ARTICLE OPEN

Common workflows for computing material properties using different quantum engines

Sebastiaan P. Huber[1]✉, Emanuele Bosoni[2], Marnik Bercx[1], Jens Bröder[3,4], Augustin Degomme[5], Vladimir Dikan[2], Kristjan Eimre[6], Espen Flage-Larsen[7,8], Alberto Garcia[2], Luigi Genovese[5], Dominik Gresch[9], Conrad Johnston[10], Guido Petretto[11], Samuel Poncé[1], Gian-Marco Rignanese[11], Christopher J. Sewell[1], Berend Smit[12], Vasily Tseplyaev[3,4], Martin Uhrin[1], Daniel Wortmann[3], Aliaksandr V. Yakutovich[1,12], Austin Zadoks[1], Pezhman Zarabadi-Poor[13,14], Bonan Zhu[14,15], Nicola Marzari[1] and Giovanni Pizzi[1]✉

The prediction of material properties based on density-functional theory has become routinely common, thanks, in part, to the steady increase in the number and robustness of available simulation packages. This plurality of codes and methods is both a boon and a burden. While providing great opportunities for cross-verification, these packages adopt different methods, algorithms, and paradigms, making it challenging to choose, master, and efficiently use them. We demonstrate how developing common interfaces for workflows that automatically compute material properties greatly simplifies interoperability and cross-verification. We introduce design rules for reusable, code-agnostic, workflow interfaces to compute well-defined material properties, which we implement for eleven quantum engines and use to compute various material properties. Each implementation encodes carefully selected simulation parameters and workflow logic, making the implementer's expertise of the quantum engine directly available to non-experts. All workflows are made available as open-source and full reproducibility of the workflows is guaranteed through the use of the AiiDA infrastructure.

npj Computational Materials (2021)7:136 ; https://doi.org/10.1038/s41524-021-00594-6

INTRODUCTION

The use of density-functional theory (DFT) to compute the properties of systems at the atomic level has become widespread[1,2], as both the number of quantum engines that implement it and the available computational power continue to increase. However, despite its large-scale deployment both in academia and in industry, the application of DFT is still not a trivial operation. Accurate predictions require expert knowledge of not just DFT itself but also of the specific code used to perform the calculations (throughout this work we will use the terms quantum engine and code interchangeably). Although the diversity of available simulation packages improves the accuracy and reliability of results by virtue of cross-verification[3], different codes use diverse computational methods and interfaces, making it difficult even for experts to master more than just a few of them. This may result in software being used not for its applicability to a particular problem, but merely due to circumstantial reasons. Furthermore, the fact that the correct usage of DFT-based codes requires expert knowledge directly limits its application and potential for scientific discovery.

Although DFT is used to compute many material properties of varying complexity, a large percentage of all performed calculations are defined by relatively simple recipes. Therefore, in addition to implementing new functionalities and improving the accuracy of existing ones, the effort of domain and code experts should be focused on providing robust workflows with common interfaces that can be used by experts and non-experts alike. If these are designed properly such that they are reusable, they can be employed as modular blocks in building more complex workflows, e.g., in a multi-scale approach. On top of reusability, in order to guarantee that results can be validated, it is crucial that these common workflows are reproducible.

A number of workflow implementations that automatically compute a variety of material properties already exist, but they are typically implemented for a single specific quantum engine[4–7]. The Atomistic Simulation Environment (ASE)[8] initiated an effort to provide a single interface for various quantum engines, however, this remains on the level of single calculations. The recent Atomistic Simulation Recipes (ASR)[9] proposes a mechanism to extend this concept to workflows, but currently only provides implementations for GPAW[10]. In this article, we address the additional challenges that one faces when trying to develop a common-workflow interface, focusing particularly on the requirements of reusability and reproducibility, and we provide a solution based on AiiDA, an informatics infrastructure and workflow management system[11]. As a proof-of-concept, we define a common-workflow interface specifically

[1]Theory and Simulation of Materials (THEOS) and National Centre for Computational Design and Discovery of Novel Materials (MARVEL), École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland. [2]Institut de Ciència de Materials de Barcelona, ICMAB-CSIC, Bellaterra, Spain. [3]Peter Grünberg Institut and Institute for Advanced Simulation, Forschungszentrum Jülich, Jülich, Germany. [4]Department of Physics, RWTH Aachen University, Aachen, Germany. [5]CEA, IRIG-MEM-L_Sim, Univ. Grenoble-Alpes, Grenoble, France. [6]Nanotech@surfaces Laboratory, Swiss Federal Laboratories for Materials Science and Technology (Empa), Dübendorf, Switzerland. [7]SINTEF Industry, Materials Physics, Oslo, Norway. [8]Department of Physics, University of Oslo, Oslo, Norway. [9]Microsoft Station Q, University of California, Santa Barbara, CA, USA. [10]Atomistic Simulation Centre, School of Mathematics and Physics, Queen's University Belfast, Belfast, UK. [11]UCLouvain, Institut de la Matière Condensée et des Nanosciences (IMCN), Louvain-la-Neuve, Belgium. [12]Laboratory of Molecular Simulation (LSMO), Institut des sciences et ingénierie chimiques (ISIC), École Polytechnique Fédérale de Lausanne (EPFL) Valais, Sion, Switzerland. [13]Department of Chemistry, Claverton Down, University of Bath, Bath, UK. [14]The Faraday Institution, Didcot, UK. [15]Department of Chemistry, University College London, London, UK. ✉email: mail@sphuber.net; giovanni.pizzi@epfl.ch

for the optimization of solid-state structures and molecular geometries, together with its implementation in eleven quantum codes: ABINIT[12–14], BigDFT[15], CASTEP[16], CP2K[17,18], FLEUR[19], Gaussian[20], NWChem[21], ORCA[22,23], Quantum ESPRESSO[24,25], SIESTA[26,27], and VASP[28,29]. This particular common-workflow interface, referred to as the "common relax workflow" throughout this work, allows a user to optimize a structure using any of these codes without having to define code-specific parameters. The computed results are returned in a single unified format with identical units making the results directly comparable and reusable regardless of the underlying quantum engine used.

Each implementation of the common relax workflow interface provides at least three protocols ("fast", "moderate", and "precise") that allow a user to specify the desired computational accuracy in an intuitive and general way. The mapping between these levels of protocols and code-specific parameters are up to the respective code experts to define. Through these protocols, expert knowledge of appropriate numerical parameters is thus encoded directly into the workflows, reducing the risk of incorrect or unreliable results and opening up the use of the quantum engines also to non-experts. Despite the ease-of-use of the workflows, the workflow interface design (which will be discussed later) maintains full flexibility and allows users to override any of the sensible defaults provided by the protocols. Furthermore, since AiiDA tracks the full provenance graph of executed workflows, storing all parameters used in workflow steps, the appropriateness of the inputs and the correctness of the results can also be checked a posteriori.

To demonstrate the concept of modularity and potential for cross-verification, we use the common relax workflow to compute the equation of state (EOS) and the dissociation curve (DC), which are commonly computed properties for bulk compounds and diatomic molecules, respectively. Each of these properties is computed by a single workflow that exclusively leverages the common relax workflow as a modular building block, allowing any of the quantum engines to be used without specifying any code-specific parameters. The EOS and the DC are computed for a few compounds with different geometric, electronic, and magnetic properties. As we will show later, the results computed by the various quantum engines show good agreement. We stress here that the focus of this paper is not on the validation of the results, but rather on the demonstration of the feasibility of a common-workflow interface, directly enabling the reusability of complex workflows and the cross-verification of their results. We hope this will motivate readers to generalize these concepts and apply them to a broader and more complex range of problems.

The implementations of the common relax workflow interface of all quantum engines described in this paper are made available as free open-source software at https://github.com/aiidateam/aiida-common-workflows under the MIT license. In addition, all workflows, as well as the seven quantum engines with a free open-source software license, come pre-installed in the Quantum Mobile[30] virtual machine (and quantum engines with a more restrictive license can be manually installed on any computational resource and configured to be used with AiiDA). This makes it straightforward to fully reproduce all the results presented in this paper (see the Supplementary Notes for complete instructions).

RESULTS AND DISCUSSION

Reusability and reproducibility

Workflows, by definition, consist of multiple steps or multiple subprocesses that are executed in series, in parallel, or in a combination thereof, to obtain the final result. Ideally, workflows can themselves be used as modular blocks, becoming steps of higher-level workflows. To keep this process practical and tractable, workflows should be designed to be as modular and reusable as possible. Additionally, as workflows become ever more complex, so does their reproducibility. In this paper, we focus on two particular concepts that address these requirements: optional transparency and scoped provenance.

In software design, the term *transparency* is often used to mean that a consumer of an interface should not be bothered with the inner details of the implementation (the details are invisible, or transparent). In terms of computational workflows, this can be taken to mean that a useful generic turn-key solution should have a simple interface, requiring as few inputs as possible from the user. Apart from physical inputs (e.g., the initial crystal structure in a relaxation workflow) and flags to determine which type of simulation to run (e.g., relax only atomic positions or also the periodic cell), any other input that is only needed as a numerical parameter by the underlying implementation should be automatically determined by the workflow.

However, this transparency of the interface comes at a cost. Complex workflows often consist of multiple subprocesses, each requiring its own inputs. Oftentimes at least some of these inputs cannot be automatically determined by the main workflow, as they are circumstantial and will be dependent on *how* and *where* the workflow is run. An example is when one of the subprocesses is executed on a high-performance computing (HPC) cluster and therefore requires specific environmental settings, such as the required resources and parallelization flags. A transparent interface is closed to these inputs being set (as shown schematically in Fig. 1a) and, as such, the workflow will be tied to a very specific environment for execution. Therefore, it will not be portable and consequently not reusable. But even if the inputs of the workflow could be automatically determined, an expert user may still want to override them. Transparent interfaces precluding this level of control diminish the reusability of workflows.

The solution to the aforementioned problem is to make the interface for all workflows fully *opaque* and expose all inputs of their subprocesses. That is to say, the workflow should make it possible to define each and every input that any of its subprocesses takes, as shown in Fig. 1b. By doing so, a user has access to all the inputs of the subprocesses, whether they could have been automatically determined by the workflow or not. Certainly, there are situations where the workflow can consciously decide not to expose certain inputs, as it is part of its task to determine them based on other inputs or intermediate results.

We are now confronted with two conflicting requirements, where a workflow interface must be both *transparent* for simplicity, yet at the same time fully *opaque* for reusability. The solution is to create an interface that is *optionally transparent*, i.e., it is opaque when needed but can still be used in a transparent manner whenever possible. Exposing the inputs of subprocesses is the first crucial step towards obtaining this goal, but it is not the only one. In addition, the workflow needs to specify sensible defaults such that the interface remains simple to operate with just a minimal set of inputs. An even better solution is offered by what we refer to as *input generators*. An input generator for a workflow is a function that, based on a minimal set of essential inputs, generates the full set of inputs required by the workflow and all of its subprocesses. The advantage of this approach is that an expert user has the ability to inspect the full set of inputs that have been generated and even modify them before actually executing the workflow. This is the approach that we will take in the rest of this work.

It is commonly accepted that science is facing a reproducibility crisis in that many studies can often not be reproduced[31].

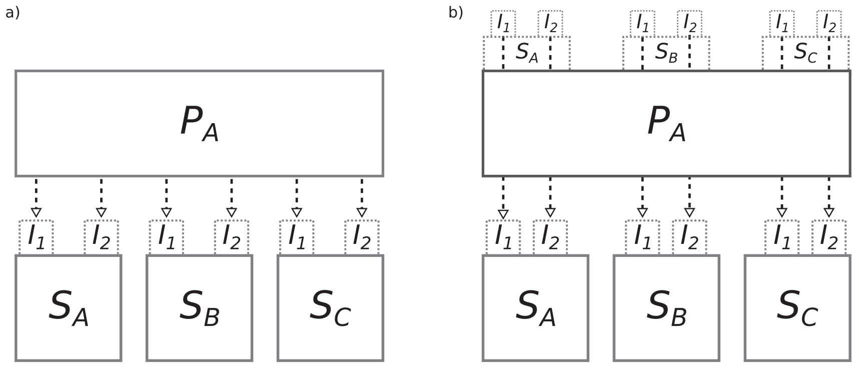

Fig. 1 Difference between a transparent and opaque workflow interface. a A schematic depiction of a process (P_A) that consists of three subprocesses (S_A, S_B, and S_C), that each requires two inputs (I_1 and I_2). In this abstract example, the top-level process takes no inputs which is just for clarity; normally the top-level process takes at least one input based on which the inputs for the subprocesses are determined. Note that, although for simplicity the same symbol is used for these inputs, they do not necessarily represent identical inputs across the subprocesses, even though in practice the names could actually overlap. Only two inputs per process are arbitrarily chosen here for illustrative purposes. The interface of P_A does not expose the inputs of its subprocesses but instead will decide them internally. This means that a user of P_A cannot customize the inputs of any of the subprocesses. **b** A schematic depiction of the same process P_A as in (**a**), but in this case exposing the inputs of its subprocesses. Since the names of the inputs can potentially overlap, inputs are exposed in namespaces to prevent name clashes. A user of P_A can now directly set the inputs through the top-level interface. If any of the inputs of the subprocesses should not be defined by the user (due to being part of the workflow's task to define it) the workflow can decide to not expose that particular input.

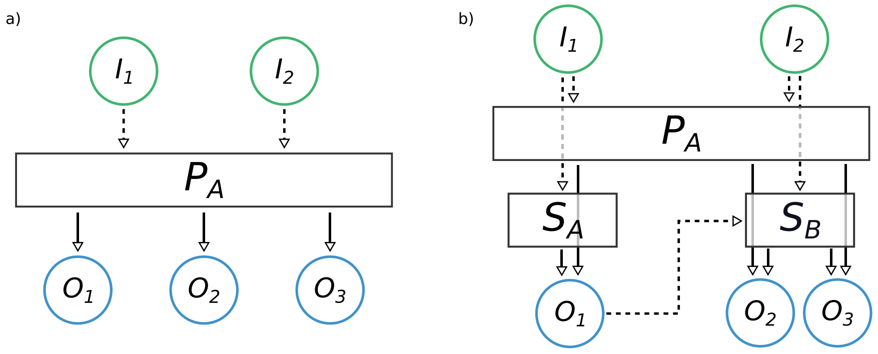

Fig. 2 Scope provenance. a Schematic provenance of a workflow (P_A) that takes two inputs (I_1 and I_2) and produces three outputs (O_1, O_2, O_3). **b** A more detailed view of the complete provenance of P_A, which actually runs two subprocesses (S_A and S_B). Input I_1 is passed by P_A to S_A, which results in O_1. This intermediate output O_1 is passed by P_A to S_B as an input, in addition to I_2, which results in the outputs O_2 and O_3. The latter are returned by P_A as the final outputs together with the O_1 intermediate result.

In recent years, guidelines have been developed to address this problem, such as the FAIR principles[32] that aim to make data, among other things, more reusable. For workflows to become FAIR as well, it is critical that they store the *provenance* of the data that they produce at each execution[33]. Concretely, this means that a workflow should store not only its own inputs and outputs but also those of all the subprocesses that it invokes. Recording the provenance of data that is produced at each step of a workflow is crucial to enable the reproducibility and intelligibility of the final result. However, full provenance is not always required. Therefore, for complex workflows that produce large provenance graphs, it becomes important to be able to investigate the provenance within different granularity levels, i.e., different scopes. We refer to the possibility of inspecting the provenance at different levels as scoped provenance, which we illustrate in Fig. 2.

The next section explains in detail how we put the concepts of optional transparency and scoped provenance into practice.

To ensure that the common workflows satisfy the requirements of optional transparency and scoped provenance, we have chosen to implement them using AiiDA[11], scalable computational infrastructure for automated reproducible workflows and data provenance. The workflows are implemented as AiiDA *work chains*[34], whose data provenance and that of all their subprocesses are automatically stored by AiiDA in a relational database. AiiDA provides an application programming interface (API) to query the provenance graph at various levels of granularity, satisfying the scoped provenance requirement. The optional transparency criterion is made possible by the design of AiiDA's workflow language specification[34]. All processes in AiiDA are implemented in Python and, most importantly, the process specification (Listing 1) is defined programmatically, allowing inspection of inputs and outputs before executing the workflow. In addition, it allows workflows to easily reuse subworkflows as modular blocks, without making their interface inaccessible, by *exposing* the inputs and outputs[34,35].

Listing 1. The definition of a process `ProcessA` implemented as a subclass of an AiiDA `WorkChain`. The process runs two subprocesses (`SubProcessA` and `SubProcessB`). The process declares an input `I_1` in its specification; in addition, the inputs of subprocesses are not redefined, but `ProcessA` simply exposes them in its own specification. The inputs of the subprocesses are exposed in separate namespaces so that inputs with same name do not shadow each other and remain all accessible (see Listing 3 for an example of how these are passed).

```
class ProcessA(WorkChain):

    @classmethod
    def define(cls, spec):
        spec.input('I_1')
        spec.expose_inputs(SubProcessA, namespace='S_A')
        spec.expose_inputs(SubProcessB, namespace='S_B')

class SubProcessA(WorkChain):

    @classmethod
    def define(cls, spec):
        spec.input('I_1')
        spec.input('I_2')

class SubProcessB(WorkChain):

    @classmethod
    def define(cls, spec):
        spec.input('I_1')
        spec.input('I_2')
```

Launching a process in AiiDA is performed by passing the process class as an argument to the `submit` function and passing the inputs as keyword arguments as shown in Listing 2.

Listing 2. Example of how `SubProcessA` is launched. The `**` marker is Python syntactic sugar to unwrap the `inputs` dictionary into keyword arguments to the `submit` function. Note that the values of the inputs are simple integers just for the clarity of the example.

```
from aiida.engine import submit
inputs = {
    'I_1': 1,
    'I_2': 2,
}
submit(SubProcessA, **inputs)
```

Listing 3 shows an example of how the top-level `ProcessA` can be launched, defining its own inputs as well as those of its subprocesses.

Listing 3. Example of how `ProcessA` is launched. The inputs of the subprocesses can be passed in dictionaries that are nested in the main inputs dictionary, where the keys correspond to the namespace in which the inputs are exposed in the process specification (Listing 1).

```
from aiida.engine import submit
inputs = {
    'I_1': 1,
    'S_A': {
        'I_1': 1,
        'I_2': 2,
    },
    'S_B': {
        'I_1': 1,
        'I_2': 2,
    }
}
submit(ProcessA, **inputs)
```

The concept of exposing inputs of subprocesses ensures that the inputs of any subprocess can be controlled from the top-level workflow, regardless of the level of nesting. This directly satisfies the requirement of providing an opaque interface for expert users that need maximal control. However, the interface quickly risks becoming complex, as multiply layered workflows will require deeply nested input dictionaries. The workflow needs to optionally provide a transparent version of the interface to enable also non-expert users to easily use the workflow.

To solve this issue for the common workflows, we implement an input generator for each workflow. Input generators are not a native AiiDA concept but are a design pattern that emerged from the needs of defining and developing common workflows. Each common workflow defines a class method `get_input_generator` that returns an instance of an object that acts as the input generator. The input generator in turn defines the class method `get_builder`, which implements the common input interface and returns an instance of a 'builder'. A builder is simply a container that wraps the generated inputs with additional information (such as the workflow class it pertains to), together with additional convenience functionality such as automatic input validation.

Since processes in AiiDA are implemented and executed directly in Python, their functionality can be easily extended. In addition, by being implemented in the same Python class as the workflow for which the inputs are generated, it is straightforward to keep the two aligned during workflow development. The `get_builder` method of the input generator takes a minimal amount of required arguments and returns a complete set of inputs for the corresponding workflow. Listing 4 shows how the input generator simplifies the usage of `ProcessA` for users, as now they only need to define a single input.

Listing 4. Example of how the launching of `ProcessA` is simplified by generating the inputs through the input generator. The `get_input_generator` class method returns an instance whose `get_builder` method can be called, to obtain a fully defined builder based on just a single input `I_1`. The builder, containing all the required inputs, can then be passed directly to the `submit` function. Since the builder also contains the process class for which it is defined, the process class itself no longer has to be explicitly passed to the `submit` function. Here, we are assuming that the inputs of the subprocess can be automatically determined by some algorithm implemented in the input generator; the number of minimally required inputs in this example is just one for simplicity.

```
from aiida.engine import submit
builder = ProcessA.get_input_generator().get_builder(I_1=1)
submit(builder)
```

An alternative approach to the problem could have been to make all subprocess inputs optional and let the workflow generate them at runtime (Fig. 1a). Although this is a valid approach, in our experience it turns out to be much less flexible in particular for experienced users, as internal parameters cannot be set from the outside. Indeed, the input generator not only makes complex workflows accessible to non-experts, but it also gives maximal flexibility to advanced users. By generating inputs before execution, they can still be modified according to the user's needs before they are passed to the workflow for execution, giving direct access to all parameters and achieving the goals of optional transparency.

The common relax workflow

As a proof of concept of the principles explained in the section, we present a common interface to a workflow that performs a geometry optimization of both molecular and extended systems, which is implemented for eleven quantum engines. Structural relaxation towards the most energetically favorable configuration is a common task in materials science, and all selected quantum engines can perform it. Nevertheless, quantum engines use a wide variety of algorithms to optimize forces on the atoms and stress on the cell. This, therefore, presents an ideal yet challenging test scenario to develop a workflow with a common interface.

The design of the interface is guided by the idea to employ *optional transparency* to create a workflow interface that is suitable both for expert and non-expert users. This interface must

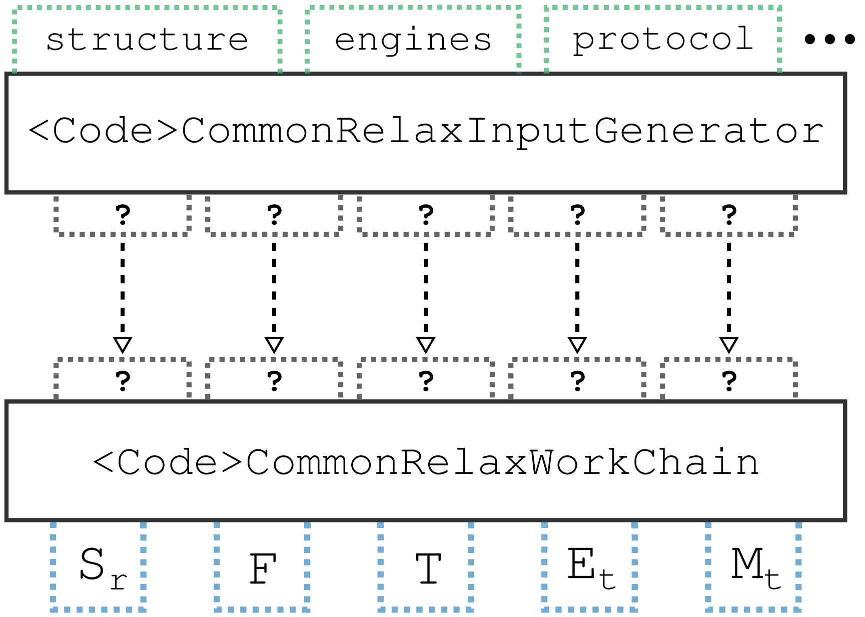

Fig. 3 Schematic diagram of the common relax workflow interface. Any implementation consists of two parts: the `<Code>CommonRelaxWorkChain` and a `<Code>CommonRelaxInputGenerator`. The `<Code>CommonRelaxWorkChain` is an AiiDA `WorkChain` that implements the logic necessary to perform the structure optimization and has an input interface that is code-specific. However, the outputs that it returns respect the schema of the common interface, where S_r is the relaxed structure, F are the forces on each atom, T is the stress on the cell, E_t is the total energy and M_t is the total magnetization of the system. Each `<Code>CommonRelaxWorkChain` provides its own `<Code>CommonRelaxInputGenerator` which, unlike the workflow, implements the common interface for the inputs (note that not all common inputs are shown for clarity). Here `structure` is the structure that is to be optimized, the `protocol` is a string that defines how the inputs are determined, and the `engine` is a dictionary that specifies what code(s) to use. The `<Code>CommonRelaxInputGenerator` translates the common inputs into the code-specific inputs that the corresponding `<Code>CommonRelaxWorkChain` expects (indicated with `?`). Since the creation of the code-specific inputs and the launching of the workflow are two separate actions, the generated inputs can be adapted at will.

be simple and general (code-agnostic) but at the same time retain full flexibility in changing code-specific parameters. Our adopted solution consists in the creation of code-specific workflows (implemented as AiiDA work chains named `<Code>CommonRelaxWorkChain`, where `<Code>` indicates the name of the underlying quantum engine), whose interface design is not restricted. A common interface is achieved by ensuring that each work chain provides also an input generator whose interface is identical for every `<Code>`, as shown in Fig. 3.

Listing 5 shows an actual code example of how the input generator can be obtained from a work chain implementation. The `get_builder` method of the input generator will transform the inputs, that respect the common interface, into the inputs that are expected by the corresponding code-specific work chain implementation. The inputs are returned in the form of a "builder" which can be directly submitted to AiiDA to start running the workflow.

Listing 5. Submission of the relaxed common workflow employing the quantum engine `<Code>`. The arguments accepted by `get_builder` are identical for every `<Code>`, establishing a common interface.

```
from aiida.engine import submit
generator = <Code>CommonRelaxWorkChain.get_input_generator()
builder = generator.get_builder(structure=...,protocol=..., ...)
submit(builder)
```

We note that not only the names but also the (Python) types of the inputs and outputs are standardized to ensure that the interface is truly generic. These are described in the next paragraphs.

The second step of the design is the identification of the minimal set of arguments for the input generator, reflecting the inputs of the most generic relaxation process. We identified three fundamental inputs that we, therefore, implemented as mandatory arguments of the `get_builder` method.

- `structure`. The structure to relax. (type: an AiiDA `StructureData` instance, the common data format to specify crystal structures and molecules in AiiDA[36]).
- `protocol`. In the context of this work, this means a single string summarizing the computational accuracy of the underlying DFT calculation and relaxation algorithm. Three protocol names are defined and implemented for each code: `'fast'`, `'moderate'`, and `'precise'`. The details of how each implementation translates a protocol string into a choice of parameters are code-dependent, or more specifically they depend on the implementation choices of the corresponding AiiDA plugin. For this work we have tried to follow these definitions: a possibly unconverged (but still meaningful) run that executes rapidly for testing (`'fast'`); a safe choice for prototyping and preliminary studies (`'moderate'`); and a set of converged parameters that might result in an expensive simulation but provides converged results (`'precise'`). The reason for not mandating the details of the protocols in the common-workflow specifications is due to the variety of basis sets, input potentials, and algorithms, requiring the specification of diverse and heterogeneous parameters in different codes. For the eleven implementations presented in this work, detailed parameter choices and the translation done for each respective code are reported in the Supplementary Methods. We note here that the choice of the exchange-correlation functional could, in the future, become an additional optional input. In this work, we decided to use the Perdew–Burke–Ernzerhof (PBE)[37] functional as the default choice. (type: a Python string).

- `relax_type`. The type of relaxation to perform, ranging from the relaxation of only atomic coordinates to the full cell relaxation for extended systems. The complete list of supported options is: `'none'`, `'positions'`, `'volume'`, `'shape'`, `'cell'`, `'positions_cell'`, `'positions_volume'`, `'positions_shape'`. Each name indicates the physical quantities allowed to relax. For instance, `'positions_shape'` corresponds to a relaxation where both the shape of the cell and the atomic coordinates are relaxed, but not the volume; in other words, this option indicates a geometric optimization at constant volume. On the other hand, the `'shape'` option designates a situation when the shape of the cell is relaxed and the atomic coordinates are re-scaled following the variation of the cell, not following a force minimization process. The term "cell" is short-hand for the combination of `'shape'` and `'volume'`. The option `'none'` indicates the possibility to calculate the total energy of the system without optimizing the structure. Not all the described options are supported by each code involved in this work; only the options `'none'` and `'positions'` are shared by all the eleven codes. The supported options might be extended in the future. (type: a Python string).

In addition to these mandatory arguments, the computational resources must be passed to the work chain in order to make the interface transferable between different computational environments. For this task, a specific argument of `get_builder` has been designed called `engines`.

- `engines`. It specifies the codes and the corresponding computational resources for each step of the relaxation process. Typically one single executable is sufficient to perform the relaxation. However, there are cases in which two or more codes in the same simulation package are required to achieve the final goal, as, for example, in the case of FLEUR. (type: a Python dictionary).

Other inputs have been recognized as common optional features that also a non-expert user might want to have control over:

- `threshold_forces`. A real positive number indicating the target threshold for the forces in eV Å^{-1}. If not specified, the protocol specification will select an appropriate value. (type: Python float).
- `threshold_stress`. A real positive number indicating the target threshold for the stress in eV Å^{-3}. If not specified, the protocol specification will select an appropriate value. (type: Python float).
- `electronic_type`. An optional string to signal whether to perform the simulation for a metallic or an insulating system. It accepts only the `'insulator'` and `'metal'` values. This input is relevant only for calculations on extended systems. In case no such option is specified, the calculation is assumed to be metallic which is the safest assumption. Note that the specification currently provides no option that instructs the workflow to automatically determine the electronic character of the system, but this might be added in the future. This also means that when the value `'insulator'` or `'metal'` is defined, the workflow will never change the electronic type, even if the calculations seem to contradict the selected value. (type: Python string).
- `spin_type`. An optional string to specify the spin degree of freedom for the calculation. It accepts the values `'none'` or `'collinear'`. These will be extended in the future to include, for instance, non-collinear magnetism and spin-orbit coupling. The default is to run the calculation without spin polarization. (type: Python string).
- `magnetization_per_site`. An input devoted to the initial magnetization specifications. It accepts a list where each entry refers to an atomic site in the structure. The quantity is passed as the spin polarization in units of electrons, meaning the difference between spin up and spin down electrons for the site. This also corresponds to the magnetization of the site in Bohr magnetons (μ_B). The default for this input is the Python value `None` and, in case of calculations with spin, the `None` value signals that the implementation should automatically decide an appropriate default initial magnetization. The implementation of such choice is code-dependent and described in the Supplementary Methods. (type: `None` or a Python list of floats).
- `reference_workchain`. A previously performed `<Code>CommonRelaxWorkChain`. When this input is present, the interface returns a set of inputs that ensure that results of the new `<Code>CommonRelaxWorkChain` can be directly compared to the `reference_workchain`. This is necessary to create, for instance, meaningful equations of state as will be shown later. (type: a previously completed `<Code>CommonRelaxWorkChain`).

The arguments of the input generator described above fully satisfy the needs for the creation of a "ready-to-submit" `<Code>CommonRelaxWorkChain`, constructing all its necessary `inputs` (see Listing 5). These inputs are code-specific and, as discussed earlier, can be modified before submission by an expert user who is familiar with the internals of the `<Code>CommonRelaxWorkChain`.

The arguments of the `get_builder` method represent high-level parameters that describe how the geometry optimization should be performed or how the system is to be treated. Each argument has a fixed number of accepted values, but not every code implementation may necessarily support all of them, as some values might correspond to features not supported by the code. In order to be able to inspect which options are supported by a workflow implementation, the input generator offers a number of methods. An example is shown in Listing 6.

Listing 6. Call to the inspection method that returns information on the available relaxation types for the `<Code>` implementation of the common relax workflow.

```
input_gen = <Code>CommonRelaxWorkChain.get_input_generator()
input_gen.get_relax_types()
```

The `get_relax_types` method returns the supported values for `relax_type` for the corresponding workflow implementation. Inspection methods are implemented for all codes and all the arguments of `get_builder` except the threshold values, the `structure`, the `reference_workchain`, and the `magnetization_per_site`. Associated to the `engines` argument, there are the methods `get_engine_types` and `get_engine_type_schema`, which return the steps required by the relaxation and information on the code type necessary for each step of the relaxation, respectively.

The described inspection methods allow introspecting, in a fully machine-readable and automatic way, what the valid options for a particular common-workflow implementation are. This is particularly relevant to facilitate future development of a graphical user interface (GUI) for the submission of the common relax workflow. The GUI will be able to create the necessary input fields, with a list of accepted values, by programmatically introspecting the input generator interface.

To allow direct comparison and cross-verification of the results, the outputs of `<Code>CommonRelaxWorkChain` are standardized for all implementations and are defined as follows:

- `forces`. The final forces on all atoms in eV Å^{-1}. (type: an AiiDA `ArrayData` of shape $N \times 3$, where N is the number of atoms in the structure).

- `relaxed_structure`. The structure obtained after the relaxation. It is not returned if the `relax_type` is `'none'`. (type: AiiDA's `StructureData`).
- `total_energy`. The total energy in eV is associated with the relaxed structure (or initial structure in case no relaxation is performed). The total energy is not necessarily defined in a code-independent way (e.g., it does not have a common zero). We require, however, that the partial derivative of the returned energy with respect to the change of the coordinate *i* of atom *j* is always the $i-$th coordinate of the force on the atom *j*. We also stress that in general, even for calculations performed with the same code, there is no guarantee to have comparable energies in different runs if the inputs are generated with the input generator (because, for instance, the selected *k*-points depend on the input structure volume). However, in combination with the input argument `reference_workchain` mentioned earlier, energies from different relaxation runs become comparable, and their energy difference is well defined. (type: AiiDA `Float`).
- `stress`. The final stress tensor in eV $Å^{-3}$. Returned only when a variable-cell relaxation is performed. (type: AiiDA `Float`).
- `total_magnetization`. The total magnetization in μ_B (Bohr-magneton) units. Returned only for magnetic calculations. (type: AiiDA `Float`).

During the execution of the workflow, which can consist of multiple runs of the relevant quantum engine, any number of problems can occur that prevent the calculation from finishing successfully. Typical examples are problems with the quantum engine itself, such as electronic convergence not being reached, or problems related to the job scheduler, such as the allocated walltime being exceeded. AiiDA provides various tools to facilitate the writing of error handlers that can recover from these errors[34] (optionally restarting from previously run calculations). However, each workflow implementation is responsible for defining and implementing these handlers, since the errors are usually quantum-engine specific. In this way, the majority of errors that typically occur are automatically handled by the workflow and the user will not have to intervene. We stress that, thanks to the extensive provenance that is automatically stored by AiiDA, the exact errors that were encountered during a workflow and how the workflow addressed them are recorded in full detail.

As a first test case of the various implementations of the common relax workflow, we present the optimization of a simple molecular structure: ammonia. The thermodynamically stable polymorph of ammonia has a trigonal pyramidal shape, which makes the structure polar. However, ammonia also exists in a metastable planar form[38]. The optimized structure and its associated total energy have been calculated with the common relax workflow implementation for all eleven quantum engines discussed in this paper for both phases of ammonia, using the `'precise'` protocol for the input generation.

The analysis of the energy difference between the planar and pyramidal configurations of ammonia is presented in Fig. 4. As mentioned in the introduction, comparing results among codes is not the focus of this paper. However, it is worth mentioning that the small discrepancies between codes in Fig. 4 are not surprising, considering that the treatment of polar molecules with codes designed for extended systems is not a trivial task. In particular, some codes always need to use periodic boundary conditions, introducing non-physical interactions among replicas in the calculation. Even for large enough simulation cells, the long-range electrostatic potential due to periodic images of the polar molecule affects the energy of the system. Strategies such as the use of improved Poisson solvers[39,40] and more sophisticated dipole corrections[41,42] can be introduced in order to circumvent this problem. Since in this paper we are focusing only on showing

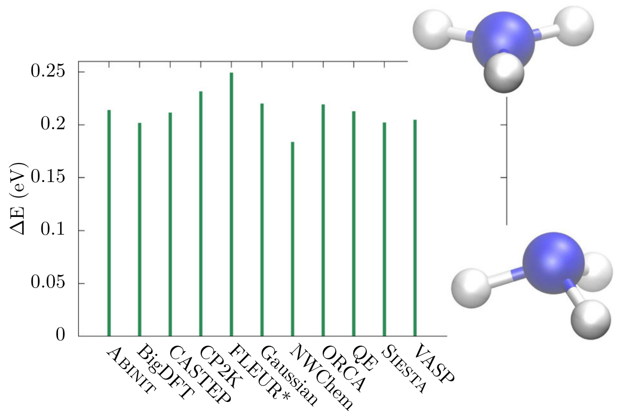

Fig. 4 Energy difference between the planar and pyramidal phases of ammonia. Energy difference (ΔE) between the planar and pyramidal phases of ammonia, calculated with the eleven quantum engines reported in the horizontal axis (QE stands for Quantum ESPRESSO). A relaxation of the structure has been performed independently by every code before computing the energies (except for FLEUR*, because for FLEUR the relaxation failed due to overlapping muffin-tin spheres, a method-specific issue requiring error handling. The energy difference for FLEUR was calculated through the common workflow without relaxation, using the relaxed output structures of Quantum ESPRESSO instead).

the concept and feasibility of common workflows, no dipole correction is considered, and the simulation box is set to a $(15\ Å)^3$ cube, without performing a proper convergence study on the cell size. However, extensions of this work can add optional flags to the input generator of the workflow to activate appropriate dipole corrections if needed and implemented by the underlying quantum engine. The data presented here also illustrates the potential of the common interface for the cross-verification of results, especially considering the variety of basis sets and algorithms of the eleven quantum engines. The present work offers the possibility to compare results from quantum-chemistry-oriented and electronic-structure codes (both pseudopotentials- and all-electrons-based) with minimum effort.

Since all workflows are implemented using AiiDA, the full provenance is automatically stored when the workflow is executed, as discussed in the Reusability and reproducibility section. Figure 5 shows two schematic provenance graphs for a relaxation workflow powered by two different quantum engines. Note that only a subselection of the total number of inputs and outputs are shown for clarity, but all subprocesses are displayed and the connections between the nodes give an idea of the internal complexity of the workflows. Notably, the figure shows how the different workflow implementations can follow considerably different logical paths while ultimately returning the same quantities according to the same common interface.

The action of taking the arguments of the common interface and transforming them in code-specific inputs (operated by the input generators) is not tracked. This is not crucial since only the code-specific inputs fully determine the calculation results. Also, we do not consider protocols as immutable objects, but rather as flexible input suggestions that an expert user might want to change.

Code-agnostic workflows

Optimizing the geometry of a solid-state structure or molecule is a common core building block of materials-science workflows. Creating a common-workflow interface for this particular task allows higher-level workflows that reuse it to become code agnostic. This makes it possible to run the workflow with any quantum engine that implements the common relax workflow

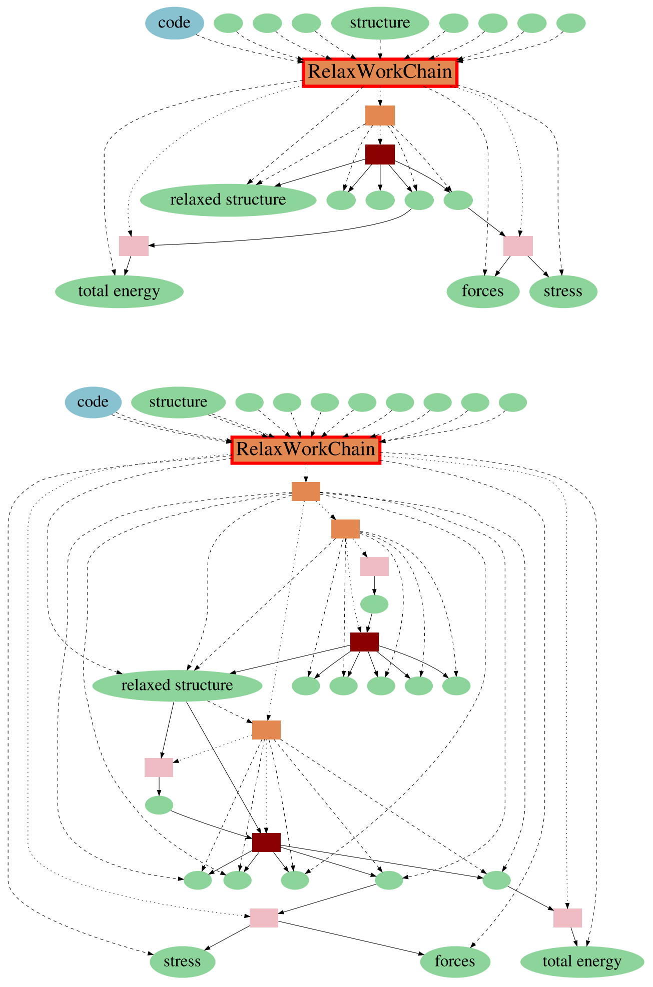

Fig. 5 Schematic provenance graph for relaxation workflows. Schematic provenance graphs for a relaxation workflow powered by two different quantum engines (top: SIESTA; bottom: Quantum ESPRESSO). The node of the `<Code>CommonRelaxWorkChain` is highlighted with the label "RelaxWorkChain" and with a red edge. All the AiiDA work chains called during the relaxation are represented by orange rectangles. The dark red rectangles are calculations, meaning calls to an external executable that performs a calculation, for instance, a call to the `pw.x` of Quantum ESPRESSO. All the ellipses represent data nodes, meaning nodes in the database that contain data, like, for instance, the initial structure, the total energy, and so on. In blue is the data node representing the code utilized for the calculations. In pink are represented calls to Python functions that modify some data in order to create others.

interface, without having to explicitly specify any input that is specific to the quantum engine. We discuss two examples of such workflows in the following sections.

An example of a workflow that uses structure optimization as a building block is the EOS workflow. The equation of the state of a solid-state system is obtained by computing the total energy of the system at various volumes. We present here the implementation of an EOS workflow that uses the common relax workflow to perform the optimization of the system at each volume and compute its total energy. This workflow serves as an example to explain how the unified interface of the common relax workflow can be used to create code-agnostic workflows. It has been named `EquationOfStateWorkflow` and its schematic representation is shown in Fig. 6.

The `EquationOfStateWorkflow` takes a structure as input (S_0 in Fig. 6) and scales the volume a number of times (N_i), with the scaled structures centered around the volume of the input structure. The workflow calls the common relax workflow for each scaled structure to compute its total energy. The common relax workflow interface is entirely accessible at the level of the inputs of the `EquationOfStateWorkflow`. This means that one can specify arguments accepted by the input generator (which are code-agnostic and are labeled `generator_inputs` in Fig. 6), but also, optionally, some code-specific `overrides` for the inputs produced by the generator. Therefore, on the one hand, by virtue of the common interface being code-agnostic, the `EquationOfStateWorkflow` is also independent of the quantum engine that is used for the underlying calculations. On the other hand, the possibility to specify explicit `overrides` should fulfill the needs of expert users and fully satisfy the optional transparency requirement for reusable workflows.

The total energies and optimized structure, as produced by the common relax workflow runs, are collected and returned by the `EquationOfStateWorkflow` as its outputs. Like the `<Code>-CommonRelaxWorkChain` itself, the code-agnostic `EquationOfStateWorkflow` is implemented as an AiiDA work chain. This provides fully automated provenance tracking of all tasks performed inside the workflow, ensuring full reproducibility of the computed results.

It should be noted that the actual logic of the `EquationOf-StateWorkflow` is slightly more complicated than depicted in Fig. 6. The common relax workflows are not all launched in parallel, but a single workflow is first performed for one of the scaled volumes. This first workflow is subsequently used as an additional input for the `reference_workchain` argument to the input generator for the common relax workflows for the remaining volumes. The input generator can use this reference to the first workflow to ensure that, if needed, parameters are kept constant between images in order for the energy differences to be meaningful. An example is the number of k-points used to sample the Brillouin zone (that is typically chosen by the input generators so as to get as close as possible to a target density, and thus is volume-dependent if a `reference_work-chain` is not specified).

The `EquationOfStateWorkflow` has been used to compute the EOS for a number of solid-state systems with varying electronic and magnetic properties: silicon (Si), aluminum (Al), germanium telluride (GeTe), and body-centered cubic (BCC) iron (Fe) both in a ferromagnetic and anti-ferromagnetic configuration. The results are shown in Figs. 7 and 8.

Figure 7 reports the EOS results for the Si, Al, and GeTe crystals. The curves for Si and Al have been obtained with all quantum engines, except ORCA and Gaussian, which are mainly specialized for non-periodic systems and the Gaussian AiiDA plugin does not yet support PBC. At each volume, the atomic positions are optimized while keeping the volume and cell shape fixed. The GeTe compound crystallizes at normal conditions in a trigonal phase (space group $R3m$)[43]. For this material, a correct calculation of the EOS requires the cell shape to be optimized at a fixed volume in order to minimize the non-hydrostatic contributions of the stress tensor. This has been achieved in the common interface setting the `relax_type` to `positions_shape`, which is only supported by five out of eleven quantum engines. This is the reason why only five curves are shown in the right panel of Fig. 7. All calculations are carried out without spin-polarization and with the `precise` protocol.

The common interface also allows calculations on magnetic systems. Figure 8 shows the EOS of BCC Fe, for both a ferromagnetic (left panel) and anti-ferromagnetic (right panel) ordering of atomic spin moments. At each volume, the atomic positions are optimized while keeping the volume and cell shape fixed. The central panel in Fig. 8 shows the total magnetization of the relaxed structure at each volume in the ferromagnetic case. The total magnetization in the anti-ferromagnetic case is zero at every volume and therefore not reported in the picture. The initial structure passed to the workflow is the same for the ferromagnetic and anti-ferromagnetic

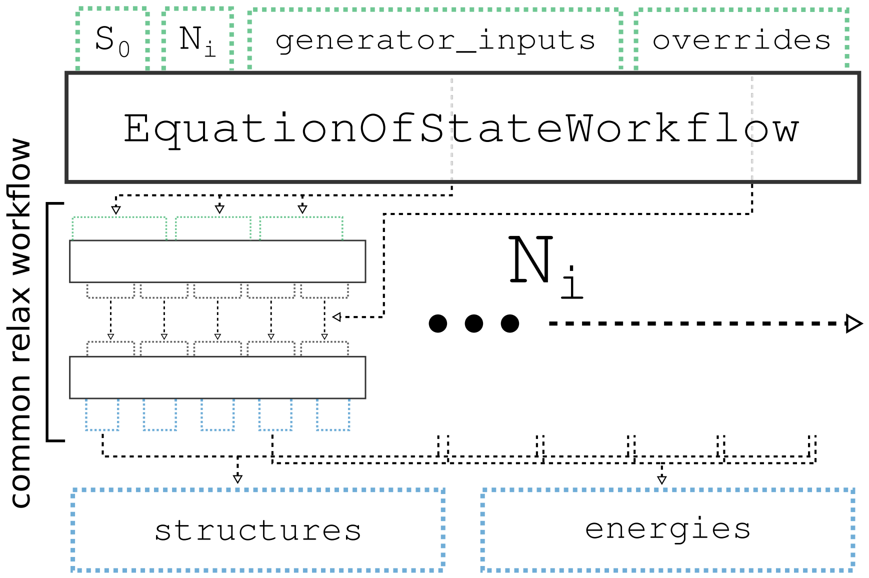

Fig. 6 Schematic diagram of the code-agnostic EOS workflow. Schematic diagram of the implementation of the code-agnostic `EquationOfStateWorkflow`. The `EquationOfStateWorkflow` takes a number of arguments: S_0 is the structure of the system at equilibrium volume and N_i are the number of volumes for which to compute the total energy. The `generator_inputs` will be passed directly to the inputs generator of the chosen common relax workflow implementation, which is called N_i times, once for each system volume. Note that the inset marked as "common relax workflow" corresponds directly to the schematic of the common relax workflow in Fig. 3. This highlights that the `EquationOfStateWorkflow` directly reuses the common relax workflow as its main building block. Which implementation of the common relax workflow is to be used is communicated to the `EquationOfStateWorkflow` by a single input, which is not shown for clarity. The `overrides` port allows an expert user to override certain inputs that are automatically determined by the generator, thus making the `EquationOfStateWorkflow` optionally transparent.

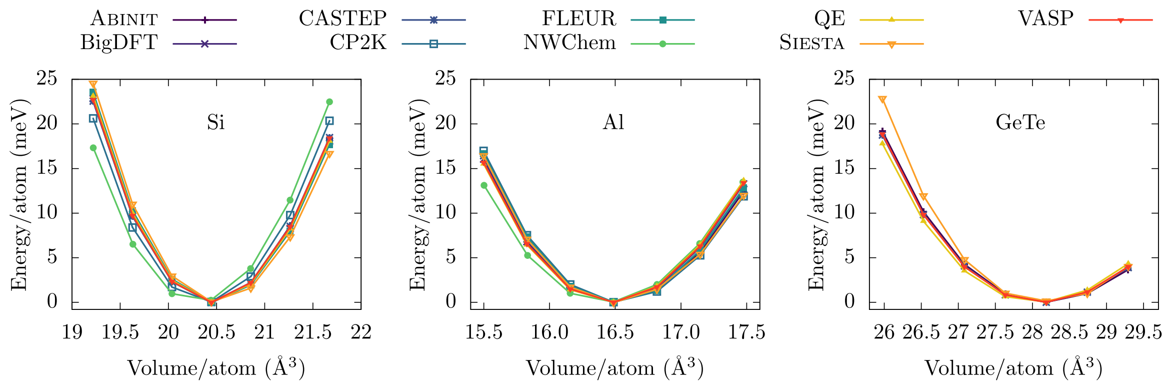

Fig. 7 EOS for Si, Al, and GeTe. Results obtained with the code-agnostic `EquationOfStateWorkflow`. For each code, the energy is shifted to set the minimum energy to zero. The EOS has been computed with all codes discussed in this work, except ORCA and Gaussian, which are mainly specialized for non-periodic systems. In addition, for GeTe, results are missing for BigDFT, CP2K, FLEUR, and NWChem (see Supplementary Table 2 for more details). The label QE stands for Quantum ESPRESSO.

configurations and it is close to the equilibrium volume of the ferromagnetic case. This explains why the volume with minimum energy for the anti-ferromagnetic case is not placed in the middle of the analyzed volumes range. It is noteworthy that the BCC structure is the most thermodynamically stable configuration only in the ferromagnetic arrangement. The results show good overall agreement among codes. However, the scope of this section is only to demonstrate the variety of systems and physical quantities that can be analyzed with the code-agnostic EOS workflow.

In a similar fashion to the EOS workflow, a code-agnostic workflow for the calculation of the dissociation curve of a diatomic molecule has been implemented. In this case, no relaxation is performed at all by the common relax workflow (accomplished by setting the `relax_type` equal to `'none'`) and it simply computes the energy of the system at various atomic distances. The same approach of the EOS workflow is used regarding the `reference_workchain` argument, meaning that the calculation at the first distance is used as a reference for the creation of inputs for the calculation at all the other distances.

Results are presented in Fig. 9 for the H_2 dissociation curve obtained with the code-agnostic workflow. An initial anti-ferromagnetic configuration has been chosen as a starting point for each energy calculation. The results show good agreement among codes. DFT is not the most appropriate method for the calculation of dissociation curves in diatomic molecules, since these systems expose well-known problems of DFT, like the delocalization error (self-interaction error) and static correlation[44]. The present test case wants to demonstrate the possibility to create code-agnostic workflows that support both electronic-structure codes and quantum-chemistry-oriented codes. In the future, the common relax workflow could be extended to allow calculations powered by different methods in addition to DFT,

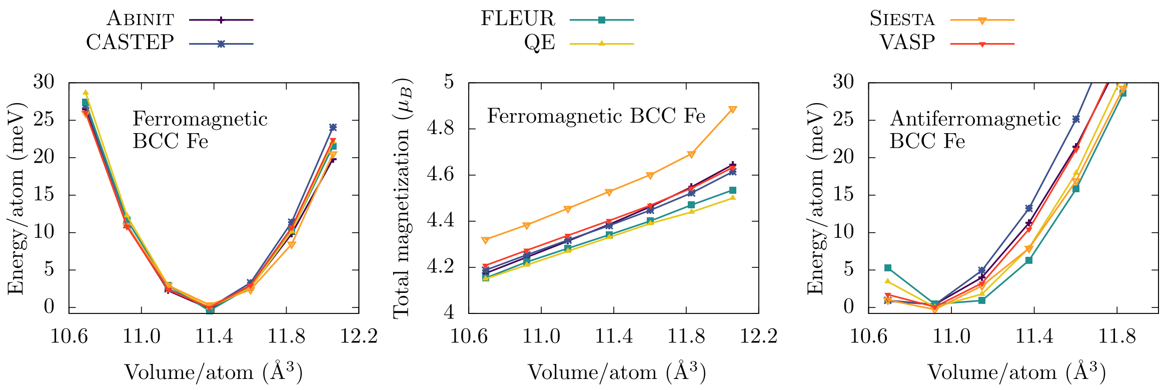

Fig. 8 EOS and total magnetization for BCC Fe. Results obtained with the code-agnostic `EquationOfStateWorkflow`. The left panel reports the EOS obtained with a ferromagnetic initialization of the atomic moments. The corresponding total magnetization is reported for each volume in the central panel. The right panel reports the EOS obtained with an anti-ferromagnetic initialization of the atomic moments. The label QE stands for Quantum ESPRESSO. Results are missing for BigDFT, CP2K, Gaussian, NWChem, and ORCA (see Supplementary Table 2 for more details).

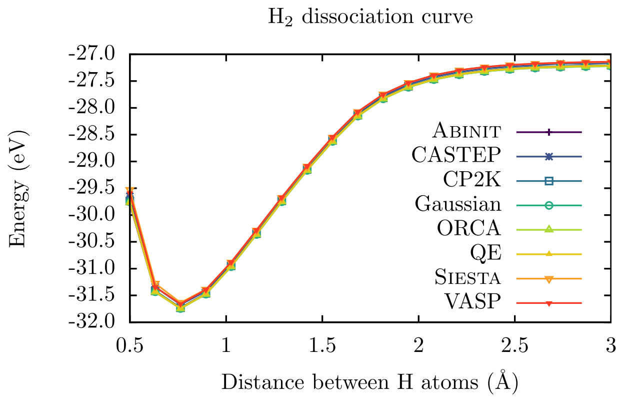

Fig. 9 Dissociation curve of the H_2 molecule. Results obtained with a code-agnostic workflow. For all codes and all volumes, the magnetization is initialized to $-1\ \mu_B$ for one atom and to $+1\ \mu_B$ for the other atom. The label QE stands for Quantum ESPRESSO. The VASP curve has been shifted by −24.89 eV. Results are missing for BigDFT, FLEUR, and NWChem (see Supplementary Table 2 for more details).

elevating the present work to a useful tool for comparing different levels of theory in the study of crystals and molecules.

We have described how it is possible for domain experts to provide robust and reusable workflows that automatically compute materials properties, in order to exploit the ever-increasing computational power and popularity of DFT-based quantum engines, with the goal of accelerating materials discovery and characterization. For the workflows to be reusable, it is critical that they have *optionally transparent* interfaces and that the full provenance of executed workflows is automatically stored. We have demonstrated a concrete implementation of these two requirements using the workflow management system of the AiiDA informatics infrastructure[11]. We defined a common interface for a workflow that optimizes the geometry of a solid-state system or molecule, that was subsequently implemented for eleven popular quantum engines, with very diverse basis-set choices and algorithms. Using this common relax workflow, we have shown how higher-level workflows can reuse it to compute relevant material properties, such as the EOS and DC, while keeping a fully code-agnostic interface. Our results show how optionally transparent common-workflow interfaces directly enable the cross-verification of results produced by different quantum engines. In addition, they empower a broader audience to use these methods in a robust way, encoding the experts' knowledge into reproducible code and hopefully stimulating new collaborations and more accurate materials-science simulations.

METHODS

Quantum-mobile implementation

The common interface and all corresponding code-agnostic workflows described in this paper allow anyone to run calculations to perform the same task with different codes, without knowing the details of each implementation. This is true assuming that the user can access a working executable of each code. The executable can be installed on the same machine as AiiDA and the common workflows, or more typically in a remote computer (HPC cluster or supercomputer), since AiiDA allows automatic connection to external machines. Compiling and installing eleven different quantum engines can be a burden even for experienced users, and even more for non-experts. As one of the goals of this work is to facilitate the access to quantum codes to a broader audience, we also make available all codes related to this project in Quantum Mobile[30] version 21.05.1, which can be downloaded here: https://quantum-mobile.readthedocs.io/en/latest/releases/versions/21.05.1.html. Quantum Mobile is an open-source virtual machine based on Ubuntu, that comes with a large number of codes, tools, and dependencies that are commonly needed to run materials-science atomistic simulations. In particular, it contains a pre-configured AiiDA installation together with the plugins interfacing AiiDA to all eleven quantum engines described here. In addition, since version 21.05.1 Quantum M obile also includes the common-workflow interfaces and implementations discussed here, released as the aiida-common-workflows package v0.1 on PyPI, which can be downloaded here: https://pypi.org/project/aiida-common-workflows/ https://pypi.org/project/aiida-common-workflows/. Crucially, Quantum Mobile also includes the executables for the following open-source quantum engines: ABINIT, BigDFT, CP2K, FLEUR, NWChem, Quantum ESPRESSO, and SIESTA (as well as a few more). Although CASTEP and ORCA provide free academic licenses, they require users to have their own license which prevents pre-installation in Quantum Mobile. The remaining three codes discussed here (CASTEP, Gaussian, and VASP) are commercial, therefore they cannot be redistributed freely without infringing their licenses. Nevertheless, a complete set of instructions is provided in the Supplementary Methods, to guide users who already have access to these codes (on any computer of their choice) to configure them with AiiDA. In this way, the common workflows (all instead available open-source in the Quantum Mobile) can be run seamlessly also for commercial codes. Thanks to this setup, common workflows with these codes can be run with almost no preliminary step required. Only a few codes require a minimal adjustment that is described in the Supplementary Methods. A detailed list of instructions on how to run the test cases presented in this manuscript in the Quantum Mobile is reported in the Supplementary Notes.

DATA AVAILABILITY

The data and the scripts used to create all the images in this work are available on the Materials Cloud Archive[45]. Note that the data includes the entire AiiDA provenance graph of each workflow execution, as well as the curated data that is extracted from that database in order to produce the images.

CODE AVAILABILITY

The source code of the common workflows is released under the MIT open-source license and is made available on GitHub (github.com/aiidateam/aiida-common-workflows). It is also distributed as an installable package through the Python Package Index (pypi.org/project/aiida-common-workflows).

Received: 11 May 2021; Accepted: 14 July 2021;
Published online: 19 August 2021

REFERENCES

1. Burke, K. Perspective on density functional theory. *J. Chem. Phys.* **136**, 150901 (2012).
2. Jones, R. Density functional theory: its origins, rise to prominence, and future. *Rev. Mod. Phys.* **87**, 897 (2015).
3. Lejaeghere, K. Reproducibility in density functional theory calculations of solids. *Science* **351**, aad3000 (2016).
4. Curtarolo, S. AFLOW: an automatic framework for high-throughput materials discovery. *Comp. Mat. Sci.* **58**, 218 (2012).
5. Kirklin, S. et al. The Open Quantum Materials Database (OQMD): assessing the accuracy of DFT formation energies. *npj Comput. Mater.* **1**, 15010 (2015).
6. Mathew, K. Atomate: a high-level interface to generate, execute, and analyze computational materials science workflows. *Comp. Mat. Sci.* **139**, 140 (2017).
7. Armiento, R. In *Machine Learning Meets Quantum Physics* 377–395 (Springer International Publishing, 2020).
8. Larsen, A. H. The atomic simulation environment—a Python library for working with atoms. *J. Phys. Cond. Matt.* **29**, 273002 (2017).
9. Gjerding, M. et al. Atomic simulation recipes—a Python framework and library for automated workflows. Preprint at https://arxiv.org/abs/2104.13431 (2021).
10. Enkovaara, J. Electronic structure calculations with GPAW: a real-space implementation of the projector augmented-wave method. *J. Phys. Cond. Matt.* **22**, 253202 (2010).
11. Huber, S. P. et al. AiiDA 1.0, a scalable computational infrastructure for automated reproducible workflows and data provenance. *Sci. Data* **7**, 300 (2020).
12. Gonze, X. Recent developments in the ABINIT software package. *Comput. Phys. Commun.* **205**, 106 (2016).
13. Gonze, X. The Abinitproject: impact, environment and recent developments. *Comput. Phys. Commun.* **248**, 107042 (2020).
14. Romero, A. H. ABINIT: overview and focus on selected capabilities. *J. Chem. Phys.* **152**, 124102 (2020).
15. Ratcliff, L. E. Flexibilities of wavelets as a computational basis set for large-scale electronic structure calculations. *J. Chem. Phys.* **152**, 194110 (2020).
16. Clark, S. J. et al. *First principles methods using CASTEP.Z. Kristallogr. Cryst. Mater.* **220**, 567–570 (2005).
17. Hutter, J., Iannuzzi, M., Schiffmann, F. & VandeVondele, J. cp2k: atomistic simulations of condensed matter systems. *Wiley Interdiscip. Rev. Comput. Mol. Sci.* **4**, 15 (2013).
18. Kühne, T. D. CP2K: an electronic structure and molecular dynamics software package—Quickstep: efficient and accurate electronic structure calculations. *J. Chem. Phys.* **152**, 194103 (2020).
19. https://www.flapw.de. *The Jülich FLAPW code family*.
20. Frisch, M. J. et al. *Gaussian 09 Revision D.01* (2016).
21. Aprà, E. NWChem: past, present, and future. *J. Chem. Phys.* **152**, 184102 (2020).
22. Neese, F. The ORCA program system. *Wiley Interdiscip. Rev. Comput. Mol. Sci.* **2**, 73 (2011).
23. Neese, F. Software update: the ORCA program system, version 4.0. *Wiley Interdiscip. Rev. Comput. Mol. Sci.* **8**, e1327 (2017).
24. Giannozzi, P. Quantum ESPRESSO: a modular and open-source software project for quantum simulations of materials. *J. Phys. Cond. Matt.* **21**, 395502 (2009).
25. Giannozzi, P. Advanced capabilities for materials modelling with Quantum ESPRESSO. *J. Phys. Cond. Matt.* **29**, 465901 (2017).
26. Soler, J. M. The SIESTA method forab initioorder-Nmaterials simulation. *J. Phys. Cond. Matt.* **14**, 2745 (2002).
27. García, A. Siesta: Recent developments and applications. *J. Chem. Phys.* **152**, 204108 (2020).
28. Kresse, G. & Furthmüller, J. Efficient iterative schemes forab initiototal-energy calculations using a plane-wave basis set. *Phys. Rev. B* **54**, 11169 (1996).
29. Kresse, G. & Joubert, D. From ultrasoft pseudopotentials to the projector augmented-wave method. *Phys. Rev. B* **59**, 1758 (1999).
30. Talirz, L. et al. Materials cloud, a platform for open computational science. *Sci. Data* **7**, 299 (2020).
31. Baker, M. 1,500 scientists lift the lid on reproducibility. *Nature* **533,** 452–454 (2016).
32. Wilkinson, M. D. et al. The FAIR guiding principles for scientific data management and stewardship. *Sci. Data* **3**, 160018 (2016).
33. Goble, C. FAIR computational Workflows. *Data Intell.* **2**, 108 (2020).
34. Uhrin, M., Huber, S. P., Yu, J., Marzari, N. & Pizzi, G. Workflows in AiiDA: Engineering a high-throughput, event-based engine for robust and modular computational workflows. *Comp. Mat. Sci.* **187**, 110086 (2021).
35. Gresch, D., Wu, Q., Winkler, G. W., Häuselmann, R., Troyer, M. & Soluyanov, A. A. Automated construction of symmetrized Wannier-like tight-binding models from ab initio calculations. *Phys. Rev. Mat.* **2**, 103805 (2018).
36. Pizzi, G., Cepellotti, A., Sabatini, R., Marzari, N. & Kozinsky, B. AiiDA: automated interactive infrastructure and database for computational science. *Comp. Mat. Sci.* **111**, 218 (2016).
37. Perdew, J. P., Burke, K. & Ernzerhof, M. Generalized gradient approximation made simple. *Phys. Rev. Lett.* **77**, 3865 (1996).
38. Ghosh, D. C., Jana, J. & Biswas, R. Quantum chemical study of the umbrella inversion of the ammonia molecule. *Int. J. Quantum Chem.* **80**, 1 (2000).
39. Cerioni, A., Genovese, L., Mirone, A. & Sole, V. A. Efficient and accurate solver of the three-dimensional screened and unscreened Poisson's equation with generic boundary conditions. *J. Chem. Phys.* **137**, 134108 (2012).
40. Castro, A., Rubio, A. & Stott, M. J. Solution of Poisson's equation for finite systems using plane-wave methods. *Can. J. Phys.* **81**, 1151 (2003).
41. Bengtsson, L. Dipole correction for surface supercell calculations. *Phys. Rev. B* **59**, 12301 (1999).
42. Makov, G. & Payne, M. C. Periodic boundary conditions inab initiocalculations. *Phys. Rev. B* **51**, 4014 (1995).
43. Goldak, J., Barrett, C. S., Innes, D. & Youdelis, W. Structure of alpha GeTe. *J. Chem. Phys.* **44**, 3323 (1966).
44. Cohen, A. J., Mori-Sanchez, P. & Yang, W. Insights into current limitations of density functional theory. *Science* **321**, 792 (2008).
45. Huber, S. P. et al. Common workflows for computing material properties using different quantum engines. *Materials Cloud Archive* **2021.73**. https://doi.org/10.24435/materialscloud:nz-01 (2021).
46. Bröder, J., Wortmann, D. and Blügel, S. Using the AiiDA-FLEUR package for all-electron abinitio electronic structure data generation and processing in materials science. In *Extreme Data Workshop 2018 Proceedings, IAS Series* Vol. 40, 43–48 (Forschungszentrum Jülich, Jülich, 2019).

ACKNOWLEDGEMENTS

This work is supported by the MARVEL National Centre of Competence in Research (NCCR) funded by the Swiss National Science Foundation (grant agreement ID 51NF40-182892) and by the European Union's Horizon 2020 research and innovation program under Grant Agreement No. 824143 (European MaX Centre of Excellence "Materials design at the Exascale") and Grant Agreement No. 814487 (INTERSECT project). We thank M. Giantomassi and J.-M. Beuken for their contributions in adding support for `PseudoDojo` tables to the `aiida-pseudo` (https://github.com/aiidateam/aiida-pseudo) plugin. We also thank X. Gonze, M. Giantomassi, M. Probert, C. Pickard, P. Hasnip, J. Hutter, M. Iannuzzi, D. Wortmann, S. Blügel, J. Hess, F. Neese, and P. Delugas for providing useful feedback on the various quantum engine implementations. S.P. acknowledges support from the European Unions Horizon 2020 Research and Innovation Programme, under the Marie Skłodowska-Curie Grant Agreement SELPH2D No. 839217 and computer time provided by the PRACE-21 resources MareNostrum at BSC-CNS. E.F.-L. acknowledges the support of the Norwegian Research Council (project number 262339) and computational resources provided by Sigma2. P.Z.-P. thanks to the Faraday Institution CATMAT project (EP/S003053/1, FIRG016) for financial support. KE acknowledges the Swiss National Science Foundation (grant number 200020-182015). G.Pi. and K.E. acknowledge the swissuniversities "Materials Cloud" (project number 201-003). Work at ICMAB is supported by the Severo Ochoa Centers of Excellence Program (MICINN CEX2019-000917-S), by PGC2018-096955-B-C44 (MCIU/AEI/FEDER, UE), and by GenCat 2017SGR1506. B.Z. thanks to the Faraday Institution FutureCat project (EP/S003053/1, FIRG017) for financial support. J.B. and V.T. acknowledge support by the Joint Lab Virtual Materials Design (JLVMD) of the Forschungszentrum Jülich.

AUTHOR CONTRIBUTIONS

G.Pi. and N.M. conceived the idea of a common interface among quantum engines and supervised the project. S.P.H., E.B., and G.Pi. coordinated the project, implemented the interface design for the common relax workflow and the code-agnostic workflow implementations that use it as a modular block. S.P.H., M.U., and D.G. conceived the design of reusable modular workflow interfaces through programmatic process specification and implemented it in AiiDA. C.J.S. created the Quantum Mobile virtual machine to include the AiiDA plugins containing the implementation of the common relax workflow interface for all quantum engines, as well as an installation of the quantum engines for those that are open-source and can be freely redistributed. S.P., Z.P., and G.Pe. developed the ABINIT implementation of the common relax workflow, which relies on the `aiida-abinit` plugin developed and maintained by S.P., A.Z., and G.Pe., and the work was supervised by G.-M.R. A.D. developed the BigDFT implementation of the common relax workflow which relies on the `BigDFTCommonRelaxWorkChain` and `BigDFTBasexorkChain` of the `aiida-bigdft` plugin package, supervised by L.G. B.Z. developed the CASTEP implementation of the common relax workflow and its protocols which relies on the `CastepCommonRelaxWorkChain` of the package `aiida-castep` developed by the same author. A.V.Y. developed the CP2K implementation of the common relax workflow and its protocols which rely on the `Cp2kBaseWorkChain` of the `aiida-cp2k` package, developed by A.V.Y. and others, and the work was supervised by B.S. J.B. developed the FLEUR implementation of the common relax workflow and its protocols, which rely on the `FleurBaseCommonRelaxWorkChain`, `FleurCommonRelaxWorkChain`, `FleurScfWorkChain`, and `FleurBaseWorkChain` of the `aiida-fleur` package[46], developed by J.B., V.T., D.W., and others, under the supervision of D.W. K.E. developed the Gaussian implementation of the common relax workflow, which relies on the `GaussianBaseWorkChain` of the package `aiida-gaussian`, developed by K.E. and P.Z.-P. C.J. developed the NWChem implementation of the common relax workflow and its protocols, which relies on the `NwchemBaseWorkChain` of the package `aiida-nwchem`, developed by C.J. and others. P.Z.-P. developed the ORCA implementation of the common relax workflow which relies on the `OrcaBaseWorkChain` of the `aiida-orca` package developed by the same author. S.P.H. developed the Quantum ESPRESSO implementation of the common relax workflow with the support of M.B., which relies on the `PwCommonRelaxWorkChain` and `PwBaseWorkChain` of the `aiida-quantumespresso` package, developed by S.P.H., M.B., G.Pi., and others. E.B. developed the SIESTA implementation of the common relax workflow, which relies on the `SiestaBaseWorkChain` of the package `aiida-siesta`[27], developed by A.G., V.D., and E.B. E.F.-L. developed the VASP implementation of the common relax workflow, which relies on the `CommonRelaxWorkChain`, `VerifyWorkChain`, and `VaspWorkChain` of the AiiDA plugin `aiida-vasp`, developed by E.F.-L. and others.

COMPETING INTERESTS

The authors declare no competing interests.

ADDITIONAL INFORMATION

Supplementary information The online version contains supplementary material available at https://doi.org/10.1038/s41524-021-00594-6.

Correspondence and requests for materials should be addressed to S.P.H. or G.P.

Reprints and permission information is available at http://www.nature.com/reprints