AiiDA Hackathon 2020 @ CINECA - Report

This document reports on the *hackathon on plugin and workflow development for AiiDA* held at CINECA, Italy from February 17th-21st 2020 (event site).

The event featured the following presentations:

1. **AiiDA: workflow design principles and guidelines** - Sebastiaan Huber (slides)
2. **Guidelines for Code plugins and WorkChain development** - Aliaksandr Yakutovich (slides)
3. **Containerization strategies: the experience in the BigDFT code** - Luigi Genovese (slides)
4. **Unit tests and integration tests** - Giovanni Pizzi (slides)
5. **AiiDA plugin development & Contributing to Materials Cloud** - Leopold Talirz (slides)

Link to lecture recordings:
https://streaming.cineca.it/DefaultPlayer/div.php?evento=Aiida_Hackathon

The event also hosted a number of discussion sessions on topics relevant to plugin and workflow development:

1. Common API for basic workflows: structure relaxation, electronic band structure & protocols for automatic selection of inputs
2. Containerization for testing infrastructure, deployment, AiiDA lab
3. Mocking executables to test CalcJob plugins
4. Continuous integration testing of WorkChains
5. Design of new Gaussian plugin
6. How to contribute a Materials Cloud DISCOVER section
7. Guidelines for user-facing workflow documentation
8. BaseRestart WorkChain - extended functionality & reuse of error handlers
9. Input dictionaries with automatic validation & help strings
10. Task farming

For the record, the notes of all discussion sessions are reproduced below.

1. Common API for basic workflows: structure relaxation, electronic band structure & protocols for automatic selection of inputs

Present:

- Dominik Gresch
- Espen Flage-Larsen
- Jens Bröder
- Aliaksandr Yakutovich
- Emanuele Bosoni
- Daniele Ongari
- Giovanni Pizzi
- Sebastiaan Huber

Discussion:

- Two topics:
    - Make a general API for two workflows (relax and band structure), as general as possible
    - Make a general syntax to specify parameters for e.g. plane-wave DFT codes (cutoffs, kpoints, …) -> delegate to later, Espen and Sebastiaan (and more) will develop a proof of concept that works for QE and VASP.
- We work on point 1 only; goal: have a demonstration for MaX soon (a few months)
- How to define them? A workchain, or a builder, or other?
- Use case?
    - Having a general GUI that can run multiple engines for the given workflow, without needing to tune the GUI for each new engine
- Not immediately needed as a deliverable: automatic error handlers. It's ok if for some structures it crashes. But the scaffolding interface is in place and can also serve as a platform to test if the workflows are working as expected.
- Presentation of Dominik https://slack-files.com/T470FLBSQ-FU6GAE40P-7c4dfa9d38

    - How to expand attributes and convert them to actual inputs for the workchain/calculation
    - Also uses the ‘@dataclass’ construct to also specify the type of variables
    - Tool to merge inputs
    - Allow input links (input variables that are evaluated only when instance is complete and being used, e.g. “4 * PW cutoff”
    - Example on how to get a builder out
    - Possible things to do:
        - Define attributes that are common, and a class to manage explicit_inputs
        - Standardise e.g. the codes and the resources (to be discussed, as there might be different steps with different resource requirements)
- Daniele's protocol: https://aiida-lsmo.readthedocs.io/en/latest/workflows.html#multistage-work-chain
- Maybe the explicit_inputs should be machine-readable so a GUI can be created? Maybe too complex? We’re restricting which inputs to change?
- Espen: why not using already the specs of WorkChains?
  Answer: you want to have a way to change the suggested inputs.
  Espen comment: yes, but this we do already.
- GP: maybe start to have something very simple, where the only things that are standardised are (for relax, for instance) the structure, the protocol string (whose value will be plugin-specific), the code(s), and the resources for each code? Then, the GUI-automation part should just be for these 4 inputs, and allow to ‘inspect’ which values are accepted (e.g., the valid protocol values, and what they mean (e.g. human-readable docs).
- EFL prefers that workchains such as a relax work chain receive a set of unified parameters relevant for relaxation in the domain of interest. Then, at some point a translation need to happen from this set to the code specific set of input parameters. This could happen in a code specific workchain that is part of a standardized workchain stack. As such, a relaxation workchain is independent of the code it runs (but probably somewhat dependent on the domain). Parameters not directly relevant to the relaxation itself should be handled outside the relaxation workchain. Quantum ESPRESSO plugin keeps current existing lower level workflows “dumb” and passing correct inputs is up to the caller.

The work chain will then simply implement the logic given a set of inputs. Higher-level workflows can then include the generation of inputs in a smart way, that might depend on other reconnaissance calculations. This is somewhat similar to the VASP plugin. EFL believes a bottom up approach when unifying workchains might be useful. E.g. start with the similar codes (e.g. QE and VASP) and try to make that work first. Then one can consider to map different domains together, possibly by reusing the domain specific workchains which is called by one that spans several domains.

General input Interface for a structural Relaxation WorkChain
Note: not all inputs valid for all codes, which should be handled by the interface

- \`structure\`
    - valid_type = (CifData, StructureData)
- \`relaxation_type\`
    - valid_type = Choice('positions', 'volume', 'shape') and combinations)
- \`thresholds.force\`
    - valid_type = float
- \`thresholds.stress\`
    - valid_type = float
- \`protocol\`
    - The key is required and the same across all codes, but values are for the time being code specific. The values *should* come however with a human readable explanation of what it entails
    - Some presets that will automatically predefine values for relaxation criterion with potential to override the presets

- EXX, spin, occupations, and all non-relaxation parameters that may be needed for the specific calculation but that are not directly relevant for the relaxation engine/workchain, should be supplied, but not as input to the *relaxation* workchain.
  They should fly though to a lower level, ultimately reaching the calculation.

Exceptions:

- KKR:
    - No relaxation -> expedited to other code

        ○ Relaxation as a functionality could/should be included in a relaxation workchain to offer such functionality to codes that do not have it

Conclusions:
- Work continues here: https://github.com/aiidateam/aiida-common-workflows

Open questions:
- How to deliver also data? (for instance pseudos to use)

2. Containerization for testing infrastructure, deployment, AiiDA lab

Present:
- Vasily Tseplayev
- Aliaksandr Yakutovich
- Jens Bröder
- (Tiziano Müller)

Related to how to containerize talk from Luigi (BigDFT) on Wednesday morning and his idea of a 'MaX-Container' for a productive AiiDA environment with all plugins. Different from quantum mobile and different target audience.

Sasha: Explained about AiiDALab, how it is deployed with docker and differences between conda and pip.
Why to also make plugin available on conda. He still had some problems with not having the right environment sourced for non login shells. Tiziano had some explanation, needs checking.

Sasha uses containers for running workchain integration tests.

Jens: In Juelich use podman, AiiDA and AiiDA-fleur is provided to Institute members within a docker container ready to use. Are working on expansion. Have to find a way to make things usable in docker and podman. Apps for Materials Cloud, AiiDA-lab and Juelich own setup.

Users should start from the same base database, some ways of customization. Have a way to allow the users to fetch certain projects or aiida exports.

Results of discussion:

- For Juelich: collaborate more on AiiDA lab, reuse work
- Plugins have to be registered on conda registry

Open questions:

- How to share data between containers? Allow like a push pull of database projects to some registry.
- For a ‘MaX container’: include AiiDA lab?

3. Mocking executables to test CalcJob plugins

Present:

- Dominik Gresch
- Tiziano Müller
- Philipp Rüssmann
- Espen Flage-Larsen
- Jens Bröder
- Leopold Talirz
- Pezhman Zarabadi-Poor

Dominik: aiida-pytest-mock-codes

- https://github.com/greschd/aiida-pytest-mock-codes/
    - Provides a pytest fixture
    - When you submit the code, it will call a command line script
    - Will hash inputs, check whether it is present in folder
    - If existing, copies pre-computed outputs
    - If not, it will run the executable specified (specified in configuration file) to generate test output data
- .aiida-pytest-mock-codes.yml: path to executable
- Mock_code_factory: returns AiiDA code instance
    - Take all inputs, hash them
    - Currently not storing inputs, just their hash

- aiida-mock-code: python click script
    - Configuration (exclude files) is added as prepend script to aiidasubmit script
      As environment variables
    - Dominik: Perhaps this could be put into the calcinfo?
- Limitations:
    - To try with scheduler and MPI etc.
- Tiziano:
    - Would be nice to get a hook in AiiDA to canonicalize inputs if necessary
    - Would prefer an optional? label in addition to the hash for the data directories (matching for label+hash)
- Dominik:
    - A "--force-regen" option would be nice
- Dominik:
    - Idea: reuse AiiDA's "get_hash" mechanism
    - Problem: if different python environments result in different inputs (to see whether this is of practical relevance)

Espen: aiida-vasp

- https://github.com/aiida-vasp/aiida-vasp
- commands/mock_vasp
- Temporary directory is placed in the AIIDA_PATH
- aiida_vasp/utils/fixtures.py
- Espen:
    - main issue for us is picking the right output depending on the input, e.g. to test the convergence workchain
    - Dominik's implementation would solve this for us as long as the sequence is predetermined. It usually is.

Philipp & Jens:

- Looks nice, no particular additional requirements
- Note: probably don't want to do this on the AiiDA
- Tiziano: be careful about file names
- Philipp: Currently no way to tell why a calculation was not found

Results of discussion

- pytest-mock-codes is a good starting point for all code developers present (tight binding, vasp, fleur, zeo++)
- We plan to start using it (as a plugin to begin with, can move into aiida-core later if desired)

Open questions / to do:

- Todo: test with schedulers / MPI
- How to provide a way for canonicalizing inputs
    - Example: by default, fleur puts a serial number of the calculation into the first line (can be turned off, however)
- (optional) Would be nice to be able to tell which is the "closest" calculation that was found in the results repository

4. Continuous integration testing of WorkChains

Present:

- Dominik
- Philipp
- Espen
- Jens
- Leopold

Discussion

- One way: use mock executables (see discussion under 3.)
- Alternative idea: use caching to test workflows
- Difference to mock executables: no longer testing CalcJob plugin
- Needed :Pytest fixture for loading export files
- Fixture (Calc-classes):
    - Try to import DB file
      Note: Also needs to *rehash* the loaded nodes, so that they get the _aiida_hash extra
    - PSEUDOCODE
      ```
      with enable_caching:
          yield # test runs
      ```
    - if "new" then create export file
      ```
        # potentially clear file repository
      ```
- note: depend on clear_database for this fixture

- Importing export file:
  with cached_calcs('workchain')
- Comment: Perhaps instead of importing one file per test, you might want to have a test session-level fixture that imports all files (?)
  To decide after practical tests
- Philipp is testing this approach and it seems to work
- Not straightforward to use this mechanisms to test sub-workchains, since you can't easily export the provenance of subworkchains in isolation (may require running them manually)

Results of discussion:

- Using caching to test WorkChains seems doable.
  Basic idea: run once, create an export file.
  When running tests, import export file and run tests with caching enabled (Philipp has already been testing it)

Open questions / to do:

- fix for configuration of caching (https://github.com/aiidateam/aiida-core/pull/3785)
- Write a fixture that automates loading of export files & the creation of export files (if they don't exist yet)

5. Design of new Gaussian plugin

Present:

- Samuel Poncé
- Aliaksandr Yakutovich
- Leopold Talirz
- Kristjan Eimre
- Carl Pignedoli
- Pezhman Zarabadi-Poor

Discussion:

- Docs on design guidelines: https://aiida.readthedocs.io/projects/aiida-core/en/latest/developer_guide/plugins/basics.html#design-guidelines
- Gaussian input files:
  - Usual calculations have a single input file that defines the job & the structure
    - Input file can specify basis set
  - Pezhman: Input file can define multi-step jobs
  - Sasha: you can also have a checkpoint file to restart
- Kristjan: Gaussian is not made for large-scale calculations
- Output files:
  - Log file (cclib will work only on this)
  - Documentation about CCLIB: https://cclib.github.io/
  - Checkpoint file <= binary, not portable
- Kristjan: was looking into cclib
- Get basis sets from: https://www.basissetexchange.org/
- Representing input structures
  - Sasha: NWchem used structuredata
  - Note: we currently don't support z-matrix format
  - Potential problem: molecules defined "across" periodic boundary conditions
  - Kristjan: StructureData might want to inherit from the MoleculeData
- Storage of input data in AiiDA:
  - Input parameters:
    - % section: potentially autogenerated from "resources"
      - Provide a way to override
    - Route (#): parameters['route']
    - Title section: parameters['title']
    - Charge multiplicity
    - Molecule specification => StructureData
    - Basis sets & PPs: Perhaps take inspiration from here: https://github.com/dev-zero/aiida-gaussian-datatypes
- Storage of output data in AiiDA:
  - Log file: retrieve, parse using cclib
    - Goes into Dict, potentially StructureData
  - Checkpoint file: no
- Add small plugin for cubegen

- Issue: Gaussian uses shared-memory parallelization
  => Need to specify multiple cores, but withmpi=False
- Input validation: One feature that we need to think about implementing would be validating the keywords in the calculation plugin. This requires constructing separate libraries for different keywords in Gaussian. Some of them like keywords for method (different functionals) and internal basisset are better to be constructed earlier.
  The other keywords for controlling the SCF, optimization, etc. can be added gradually on the usage and occurrence by user and developers.

Open questions:

- How to represent non-periodic structures?
    - Do we simply use StructureData with periodic = [False, False, False]?
      Potential problem: molecules defined "across" periodic boundary conditions
    - Or do we create a more chemistry-oriented class
        - E.g support for z-matrix, SMILES, ...

Relevant links/ info:

- Example Gaussian input

# HF/6-31G(d) *Route section*

water energy *Title section*

0 1 *Molecule specification*

O -0.464 0.177 0.0

H -0.464 1.137 0.0

H 0.441 -0.143 0.0

- pyGauss: https://github.com/chrisjsewell/PyGauss NOTE: only for visualization (seemingly).

- Besides cclib, the other possibility is using pymatgen (capabilities for both writing input file & parsing output): https://pymatgen.org/pymatgen.io.gaussian.html
- code snippet for reading the coordinates as xyz and write the gaussian input file:

```
import pymatgen as mg
import pymatgen.io.gaussian as mgaus
mol = mg.Molecule.from_file('./test.xyz')
inp = mgaus.GaussianInput(
   mol, charge=0, spin_multiplicity=1,
   title='MY test',
   functional='PBE1PBE',
   basis_set='aug-cc-pvdz',
   route_parameters={
      'Symmetry':'None',
      'Output':'WFX'},
   input_parameters={'output.wfx':None},
   link0_parameters={
      '%chk':'mychk.chk',
      '%mem':'30gb',
      '%nprocshared':'16'
      },
   dieze_tag='#P'
   )
inp.write_file('./test.gjf', cart_coords=True)
```

6. How to contribute a Materials Cloud DISCOVER section

Present:

- Leopold Talirz
- Jens Bröder
- Philipp Rüssmann
- Pezhman Zarabadi-Poor
- Emanuele Bosoni

- Alberto Garcia

Discussion:

- Basic instructions: https://www.materialscloud.org/discover/help
- AiiDA code in structure-property-visualizer has not been tested in a long time
- First containerized discover section that directly queries the AiiDA database: https://github.com/lsmo-epfl/discover-curated-cofs
    - Runs inside docker container
    - Visualization with panel / bokeh
    - Communicates with AiiDA database sitting on database server *outside* container
    - AiiDA file repository mounted *inside* container (but not part of the docker image)
- Open question: How to define the interface between discover container and the external services (AiiDA DB, file repository)
- Current solution (to be updated):
    - File repository mounted at /app/.aiida/repository/{{ aiida_profile }}
    - AiiDA config file with correct variables (see template [1]) copied into container image when building the Dockerfile
- This makes things difficult on our side
- Proposed improvement:
    - Materials Cloud will pass the following environment variables to the container:
        - AIIDA_PROFILE
        - AIIDADB_ENGINE
        - AIIDADB_PASS
        - AIIDADB_NAME
        - AIIDADB_HOST
        - AIIDADB_BACKEND
        - AIIDADB_PORT
        - AIIDADB_REPOSITORY_URI
        - AIIDADB_USER
        - default_user_email
    - We add support to verdi to read these properties from the environment

    - You can easily test things locally by writing them into a .env file (we also add support for loading this from a .env file)
      Using https://pypi.org/project/python-dotenv/
    - Materials Cloud will mount file repository under /app/.aiida/repository/{{ aiida_profile }}
- Dokku: http://dokku.viewdocs.io/dokku/

Conclusions:
- To get started with discover sections at the moment:
    - Look at the curated-cofs section
      https://github.com/lsmo-epfl/discover-curated-cofs
      (Leopold will update structure-property-visualizer when time)
    - Don't worry about the docker setup when getting started - first get it to run locally without docker

Open Questions / to do:
- To do (Materials Cloud Archive): record AiiDA version compatible with an archive record export in machine-readable form
- To do (Materials Cloud EXPLORE): containerize explore sections in order to be able to run EXPLORE sections with different AiiDA versions (needs to match AiiDA version of DISCOVER section)
- To do (aiida-core): can we have AiiDA read environment variables AIIDA_…?

Example AiiDA configuration template

```
{
  "CONFIG_VERSION": {
    "CURRENT": 3,
    "OLDEST_COMPATIBLE": 3
  },
  "default_profile": "{{ aiida_profile }}",
  "profiles": {
    "{{ aiida_profile }}": {
      "AIIDADB_ENGINE": "postgresql_psycopg2",
      "AIIDADB_PASS": "{{ aiida_db_pass }}",
      "AIIDADB_NAME": "{{ aiida_db_name }}",
      "AIIDADB_HOST": "{{ aiida_db_host }}",
      "AIIDADB_BACKEND": "django",
      "AIIDADB_PORT": "{{ aiida_db_port }}",
      "default_user_email": "leopold.talirz@epfl.ch",
      "AIIDADB_REPOSITORY_URI": "file:///app/.aiida/repository/{{
aiida_profile }}",
      "AIIDADB_USER": "mcloud",
      "options": {}
    }
  }
}
```

7. Guidelines for user-facing workflow documentation

Present:

- Daniele Ongari
- Alberto Garcia
- Emanuele Bosoni
- Vasily Tseplayev
- Leopold Talirz
- Carlo Pignedoli
- Sebastiaan Huber
- Aliaksandr Yakutovich

Discussion:

- Question 1: Where to put user-facing workflow documentation (including the science)?
    - Docstring?
    - Special format in docstring?
    - Some new workflow attribute?
    - README?
    - Readthedocs?
    - "Literate programming" for python?
      Would be great to have aYou document as you go,
- Question 2: Guidelines / good examples of how to describe the science going on in a workflow
- Daniele:

    - Suggestion: 3 lines in docstring + link to RTD
      (link hardcoded in the docstring?)
    - Describe what workchain can do & cannot do
    - Some redundancy is perhaps unavoidable
- Sasha:
    - Do not place this information in the README of the github repo
- Alberto: Can we put latex equations?
  Philipp: yes
- Alberto: It could be good to put the docstring in the database for persistence so that someone can import it and have the documentation
  Leopold: We should revisit this once we have deduplication of nodes in the AiiDA DB
- Jens: for linking within sphinx
  https://kevin.burke.dev/kevin/sphinx-interlinks/
- Sebastiaan: Suggest format RST for this docstring?
  RST is widely used in docstrings, see e.g. click, tornado
- Leopold: Use DOI-links wherever possible

Conclusions:

- Suggestion: For user-facing turn-key workflows, include a short documentation describing the science inside the workflow docstring (a few sentences), and a longer version in the documentation of your plugin package.
- Suggestion: Describe what your WorkChain *can* do, but also what it *cannot* do
- Suggestion: In the docstring, use plain text or ReStructuredText (On AiiDA lab etc. we will support parsing RST)
- Where to look for first versions of this in practice:

  Sasha: will use this in aiida-cp2k
  Carlo: will start using this in the nanoribbon-surfaces app
  Emanuele/Alberto: use it for the general relax workflow
  Pezhman: aiida-porous-materials
  Vasily: aiida-fleur
  Sebastiaan: aiida-qe
  Daniele: use it in aiida-lsmo

Open questions:

- How to best link from the short description in the docstring back to the documentation
  (hard coded link or is there something better maintainable?)

8. BaseRestart WorkChain - extended functionality & reuse of error handlers

Present:

- Espen Flage-Larsen
- Sebastiaan Huber
- Aliaksandr Yakutovich
- Alberto Garcia
- Philipp Rüssmann
- Vasily Tseplayev
- Carlo Pignedoli
- Giovanni Pizzi
- Daniele Ongari
- Leopold Talirz

Disclaimer: the `BaseRestartWorkChain` is nothing more than a base implementation of a `WorkChain` that can be easily reused to prevent copy-pasting and a lot of template code. The basic concept is that if you run a process in AiiDA (e.g. a WorkChain or CalcJob), it can fail. The errors causing the failure need to be handled and the process restarted until it is successful. This simple logical flow is exactly what is implemented in the `BaseRestartWorkChain`. By subclassing it, instead of the normal `WorkChain` one does not have to implement all this logic over and over again. Of course the error handling is sub process specific and so cannot be defined by the BRWC but has to be done by the sub class. The most logical solution is that these are implemented as instance methods, just as the outline steps. The `inspect_process` then only needs a way to collect (i.e. detect which defined methods are "process handlers") these handler methods and call them in a specific order. A python decorator is a nice concept of syntactic sugar to accomplish exactly this. It needs to be restated again that a process handler is nothing more than just another work chain instance method that is called during the logical outline.

Motivation: The BaseRestartWorkChain (BRWC) provides a nice mechanism to insert multiple functions that allow performing atomic checks on the results and, if discovered, handle problems (ProcessHandler). This mechanism can be seen as an alternative way to build calculation-specific work chains. Or as a supplement for fine-tuning behaviour or possibly reuse a work chain with minimal modification for other purposes. In addition, the handler concept, in particular, error handlers are a well-known concept in process control and conceptually it can be easier for users to relate to. However, currently one can only add process_handlers that are defined in the same folder with the code-specific BRWC.

(*It is not clear what your work chain architecture is: do you have all "feature" work chains calling the BRWC?, or do you have a "base" work chain that calls the BRWC and is itself called by feature work chains?)*

(Also, from the point of view of specific methods/physics: Can't the PBE functional be specified as a parameter to a more general "DFTGeoOpt" work chain?)

For example, if I was to build a DftPbeGeoOptWorkChain that, according to its name, does geometry optimization using PBE functional - I should introduce multiple checks that are quite specific to the methods in use (DFT convergence, geometry convergence) and fixes to those problems that are also quite specific. The problem with this approach is that you have to build an additional work chain for a slight modification of a method (different DFT functional, geo or cell optimization, optimization method).

One approach is to isolate the relaxation optimization as much as possible and rely on e.g. changes to the DFT functional in other calling, or lower level workchains. In this case all the logic would reside in the workchains themselves.

Another alternative approach to this is to use BRWC with a preselection of ProcessHandlers that is suitable for the current calculation. In this way, one can have a `process_handlers` module and use only the handlers that are needed.

- Now in aiida_core 1.1.0

- The mechanism to dress work chain with some kind of handler in place (e.g. register_process_handler dresser which you put on a defined function)
    - Avoids prefixing
    - Place these functions in the work chain they are used
    - Can use in other WorkChains if BaseRestartWorkChain is subclassed

Can we generalize where/how the handler functions are defined and included?

- Should avoid import mechanisms due to its additive character unless you want to use the same handlers for all the work chains?
- Should we allow run time injection and how do we do that?
- Should we put the process handlers in the input as a spec?
- Should we validate the handlers?
- Do we want to allow injection of handlers?
    - Good arguments against this concept with regards to the damage that can be done with respect to reproducibility
    - We only want injection in the class where the handlers are defined since it is difficult to generalize
- If we allow injection, in order to respect provenance, should we use the input inspection and store the source code of the process handlers?
    - Store source code in the repository of the work chain node?
    - Store source code with e.g. SinglefileData?
    - Entry point system?
- How about parameters to handlers?
- Should be careful with extending the handler concept too much as it will at some point take the job of what the work chain should do. What is and what is not the role of the handler?
- Comment Espen: The concept of process_handlers can also be directly embedded in a workchain (i.e. hard coded). At some point, depending on the allowed complexity of the process_handlers, it will intersect the executing role of the workchain. This is not a bad thing as it gives the plugin developers flexibility.
- Make sure not to include too much functionality at first due to backwards compatibility.

Results of discussion:

- Handler definition should be placed in the acting class (less flexible, but safer)

- There should be a way to enable/disable the defined handlers
  An input port will be added that takes a `Dict` node where keys are the method name of process handlers and the value is a boolean, where `True` means it should be enabled and `False` should disable. This overrides the value of the decorator in the source code.
  (PR https://github.com/aiidateam/aiida-core/pull/3786)
- The `process_handler` decorator will take another keyword `enabled` which is `True` by default, but can be set to `False` in the source code. This will cause it to be skipped in the `inspect_process`
- There is no clear line between tasks for process handlers and tasks to be handled by the rest of the workchain. Process handlers, however, provide a standardized API for for enabling/disabling parts of the logic of a workchain.

9. Input dictionaries with automatic validation and help strings

Present:

- Sebastiaan Huber
- Leopold Talirz
- Philipp Rüssmann
- Vasily Tseplayev
- Daniele Ongari
- Pezhman Zarabadi-Poor
- Tiziano Müller

Discussion

- Question 1: validator for Dict or sub-class of Dict?
- Question 2: what about auto documentation?
- Dict builder to enable tab completion
-
- kkr: class given a dictionary of parameters will validate it and write the input file
  Seb: better do validation in the valid_type/validator rather than in prepare_for_submission
- Tiziano: pydantic - Use data classes to write a schema
  or marshmallow

Both options are significantly heavier/expressive than voluptuous; there doesn't seem to be a great middleground

- another difference to voluptuous: pydantic cannot define a nested schemas directly (have to do it via references)
- Pezhman: What should be validated?
  Leo: Main goal is to save users personal and computational time
  Philipp: Even rare errors are worth validating
  Tiziano: Validate that the user is not doing something that goes against assumptions of the plugin
- Note: both pydantic and process builder support defining validator functions
  - pydantic requires python 3.7
  - Leo: has been changing quite rapidly over the last few months
- Tiziano: in python you can have dynamic classes via the __dict__ attribute
  Data classes don't have this (attributes are fixed at declaration time), which saves memory
  Pydantic uses data classes
- From September onwards, AiiDA could take advantage of data classes
  Seb: we should state that we follow the python EOL of the python versions themselves: https://devguide.python.org/#branchstatus
- Pydantic: since this is using types, editors can already do suggestions / type validation

Examples

- simple voluptuous example aiida-diff
  https://github.com/aiidateam/aiida-diff/blob/master/aiida_diff/data/__init__.py
  - n = DiffParameters(dict={'key': 123}) # will fail
    n = DiffParameters(dict={'key': True}) # works
    print(n)
    n.schema
- zeo++ implementation with voluptuous:
  https://github.com/ltalirz/aiida-zeopp/blob/master/aiida_zeopp/data/parameters.py
- aiida process builder:
  https://github.com/aiidateam/aiida-core/blob/develop/aiida/engine/processes/builder.py
  - use schema to get `PortNamespace` and reuse this

- Example using AiiDA's Process builder

The following is a quick mockup of how we can use our own PortNamespace functionality with the builders that we already use for process builders. The only change required in AiiDA-core is to catch an exception in the ProcessBuilderNamespace.__setattr__ if the `serializer` attribute does not exist on the port, but this is three extra lines.
https://github.com/aiidateam/aiida-core/blob/develop/aiida/engine/processes/builder.py#L86

```
from aiida.orm import Data
from plumpy import InputPort
from aiida.engine.processes.ports import PortNamespace
from aiida.engine.processes.builder import
ProcessBuilderNamespace


class Dict(Data):

     _schema = PortNamespace()

     @classmethod
     def spec(cls):
     cls._schema['restart'] = InputPort('restart',
valid_type=str, help='Restart mode')
     cls._schema['cutoff'] = InputPort('cutoff',
valid_type=int, help='Cutoff energy')
     return cls._schema

     @classmethod
     def get_builder(cls):
     return ProcessBuilderNamespace(cls.spec())


builder = Dict.get_builder()

builder.restart??
builder.restart = 5   # Will Raise
builder.cutoff = 10
```

```
# Syntax options for creating the actual node
Dict(dict=builder)     # Option 1a  1 votes
Dict(builder=builder)  # Option 1b  5 votes
builder._get_node()    # Option 2a
builder.get_node()     # Option 2b
Dict.from_builder(builder)  # Option 3  votes
~~builder                   # Option 3~~
```

- Example of steering workflow control depending on which inputs are passed https://github.com/aiidateam/aiida-quantumespresso/blob/develop/aiida_quantumespresso/workflows/pw/bands.py

Conclusions

- With a small fix, the Port namespaces should be easy to use for a "dict builder"
    - The builder provides tab-autocompletion
    - The builder provides help strings (with ?)
    - The builder provides validation
    - May allow to easily reuse the aiida sphinx extension to document validated input dictionaries
- Where to look for examples of this:
    - Sebastiaan plans to give it a shot with aiida-qe when he finds time
    - Leopold plans to give it a shot with aiida-zeopp
- For the moment, pydantic requires python 3.7 and is therefore not an option for aiida-core (did not look into marshmallow in detail).
  Of course, plugins are still free to use pydantic/marshmallow
- Leopold will update the AiiDA roadmap for dropping python 3.5/3.6 support
  Once we switch to python >= 3.6, we should start looking into using data classes (e.g. for ports)
- Vote in the plenum on syntax for creating the node from the builder favored:
  Dict(builder=builder)

10. Task farming

Present:

- Giovanni Pizzi
- Pezhman Zarabadi-Poor
- Leopold Talirz

Challenge:

One of the current challenges on the way of high-throughput screening studies using AiiDA is the lack of task farming functionality in AiiDA, i.e. to pack multiple runs inside a single submission script. This becomes an efficiency issue while using serial codes such as RASPA and Zeo++, or parallel codes which do not scale very well with the number of processors.
In the case of serial codes, if the HPC does not provide the ability of reserving only one cpu in a computation node, either we need to reserve a full node for a single processor job or not use the resources at that center.

One current non-AiiDA solution for this challenge is using job arrays in combination with GNU Parallel. As an example, you may check the following script which does the job:

```
#!/bin/bash

# Defining the PBS parameters
#PBS -l select=1:ncpus=24
#PBS -l walltime=48:00:00
#PBS -A OPEN-16-53
#PBS -q qprod
#PBS -N proj12_widom
#PBS -J 1-10720:24
#PBS -o proj12_widom.log
#PBS -e proj12_widom.err

# Loading RASPA module
ml RASPA/RASPA-2.0.37-dev
```

```
### Setting up GNU Parallel and Job Array
[ -z "$PARALLEL_SEQ" ] && \
{ module add parallel ; exec parallel -a $PBS_O_WORKDIR/numtasks $0 ; }
IDX=$(($PBS_ARRAY_INDEX + $PARALLEL_SEQ - 1))
TASK=$(sed -n "${IDX}p" $PBS_O_WORKDIR/tasklist)
[ -z "$TASK" ] && exit

##### Running RASPA on the input files
cd $TASK
simulate < simulation.input
```

Briefly, we generate all the input files at once and write the absolute path of each job directory to the tasklist file. We put the number of tasks which we want to run on each node, in the numtasks. Once we submit the script to PBS, it will put all the jobs in the queue and run them.

Technologies used:

- job arrays - PBS feature (-J …, $PBS_ARRAY_INDEX, $PARALLEL_SEQ) allows to specify how many jobs to run per node
  Pezhman: does not support
- module parallel - can handle non-CPU-intensive jobs that can share the same core with others
  Note: in the example above, this is not actually used; you could remove "module parallel" and nothing would change
- Documentation for capacity computing (Pezhman's supercomputer): https://docs.it4i.cz/general/capacity-computing/

Possible Implementation in AiiDA:

Possible Solution:

Designing a switch like TASK_FARMING that if set to True, AiiDA would generate the inputs, upload them to the frontend, but do not submit the script until the whole upload procedure is finished.

Obstacles:

How to monitor the jobs? Job arrays go by the same Job ID and arrays would be present as range with job ID, for instance: Job ID as the output of qstat would be something like: 9781379[].isrv5. The full list of jobs can be obtained by for instance: qstat -u $USER -t. This already makes things complicated as we need to distinguish/group jobs which are running on the same node. This is for the case if we want to retrieve node-by-node. One solution would be just waiting for all Jobs to be finished and retrive.

Discussion:

- Questions:
  - Which other schedulers support task arrays?
  - Pezhman: slurm should support it
- Pezhman: When you do qstat, you see one job
- As a first implementation, we could just wait until everything is finished
- Giovanni:
  - Start uploading with ".pre_submit"
  - The question is: how do you trigger the actual job submission?
  - Get new node (special JobCalculation)
    - To decide whether it has to be in the graph or not
    - involves
  - When it is finished, it will also change the state of the pre-submitted JobCalcs to "finished"
  - Problem: Often you may not know when it is "time to actually submit" but you
    Will need some way to "group" jobs saying that "they can be packed together"
  - From the point of view of AiiDA, someone else (the metascheduler) will pack them
- Requirements for meta-scheduler:
  - should not live inside AiiDA
  - Provides an interface we can call
  - Some clever logic for packing things
  - Define rules on what can be packed and what can't
  - (difficult) if you submit 20 and you can pack 50, after some timeout they should anyway be submitted…

        Can this be done without a daemon? E.g. just with manual trigger to "flush"
    - (difficult) Concurrency - multiple task farming jobs being submitted simultaneously
- Use sqlite?
- Set a date for a skype call next week

- Giovanni: Have a look at myqueue
    - https://myqueue.readthedocs.io/en/latest/
    - https://gitlab.com/myqueue/myqueue/
    - Lives on the supercomputer
- Pezhman: myqueue could be useful to translate between schedulers but does not seem to support task farming itself (? to be understood)

Conclusions:
- A basic implementation of task farming seems possible:
  a user "pre-submits" jobs that are then submitted only once a given limit is reached
- Set a date for a skype call next week; possibly involve Kevin Jablonka as well

Open Questions:
- How to trigger actual job submission?
- Are there meta-schedulers around that we could use for this task?