Writing workflows in AiiDA

Workflows for the real-world

Sebastiaan Huber and Espen Flage-Larsen

AiiDA Workflow Tutorial May 22nd 2019

OUR WORKFLOWS SO FAR

Consider the typical workflow structure so far...

```
@workfunction
def run_eos_wf(code, pseudo_family, element):
    """Run an equation of state of a bulk crystal structure for the given element."""

    ...

    # Loop over the label and scale_factor pairs
    for label, factor in list(zip(labels, scale_factors)):

        ...

        # Launch a `PwCalculation` for each scaled structure
        calculations[label] = run(PwCalculation, **inputs)

    ...

    inputs = {
        label: result['output_parameters']
        for label, result in calculations.items()
    }
```

- Loop over some input parameters

OUR WORKFLOWS SO FAR

Consider the typical workflow structure so far...

```
@workfunction
def run_eos_wf(code, pseudo_family, element):
    """Run an equation of state of a bulk crystal structure for the given element."""

    ...

    # Loop over the label and scale_factor pairs
    for label, factor in list(zip(labels, scale_factors)):

        ...

        # Launch a `PwCalculation` for each scaled structure
        calculations[label] = run(PwCalculation, **inputs)

    ...

    inputs = {
        label: result['output_parameters']
        for label, result in calculations.items()
    }
```

- Loop over some input parameters
- Launch a calculation for each iteration

OUR WORKFLOWS SO FAR

Consider the typical workflow structure so far...

```
@workfunction
def run_eos_wf(code, pseudo_family, element):
    """Run an equation of state of a bulk crystal structure for the given element."""

    ...

    # Loop over the label and scale_factor pairs
    for label, factor in list(zip(labels, scale_factors)):

        ...

        # Launch a `PwCalculation` for each scaled structure
        calculations[label] = run(PwCalculation, **inputs)

    ...

    inputs = {
        label: result['output_parameters']
        for label, result in calculations.items()
    }
```

- Loop over some input parameters
- Launch a calculation for each iteration
- Use the results of the calculation ...

OUR WORKFLOWS SO FAR

Consider the typical workflow structure so far...

```
@workfunction
def run_eos_wf(code, pseudo_family, element):
    """Run an equation of state of a bulk crystal structure for the given element."""

    ...

    # Loop over the label and scale_factor pairs
    for label, factor in list(zip(labels, scale_factors)):

        ...

        # Launch a `PwCalculation` for each scaled structure
        calculations[label] = run(PwCalculation, **inputs)

    ...

    inputs = {
        label: result['output_parameters']
        for label, result in calculations.items()
    }
```

- Loop over some input parameters
- Launch a calculation for each iteration
- Use the results of the calculation ...
- Call it a day!

OUR WORKFLOWS SO FAR

Consider the typical workflow structure so far...

```python
@workfunction
def run_eos_wf(code, pseudo_family, element):
    """Run an equation of state of a bulk crystal structure for the given element."""

    ...

    # Loop over the label and scale_factor pairs
    for label, factor in list(zip(labels, scale_factors)):

        ...

        # Launch a `PwCalculation` for each scaled structure
        calculations[label] = run(PwCalculation, **inputs)

    ...

    inputs = {
        label: result['output_parameters']
        for label, result in calculations.items()
    }
```

- Loop over some input parameters
- Launch a calculation for each iteration
- Use the results of the calculation ...
- Call it a day!

CONTAINS ONE GLARING MISTAKE IN REASONING

OUR WORKFLOWS SO FAR

Consider the typical workflow structure so far...

```python
@workfunction
def run_eos_wf(code, pseudo_family, element):
    """Run an equation of state of a bulk crystal structure for the given element."""

    ...

    # Loop over the label and scale_factor pairs
    for label, factor in list(zip(labels, scale_factors)):

        ...

        # Launch a `PwCalculation` for each scaled structure
        calculations[label] = run(PwCalculation, **inputs)

    ...

    inputs = {
        label: result['output_parameters']
        for label, result in calculations.items()
    }
```

- Loop over some input parameters
- Launch a calculation for each iteration
- Use the results of the calculation ...
- Call it a day!

CONTAINS ONE GLARING MISTAKE IN REASONING

ASSUMES THAT CALCULATIONS **NEVER FAIL**

OUR WORKFLOWS SO FAR

Consider the typical workflow structure so far...

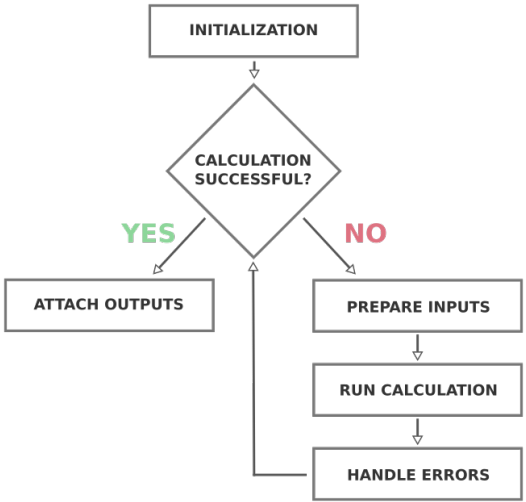
```python
@workfunction
def run_eos_wf(code, pseudo_family, element):
    """Run an equation of state of a bulk crystal structure for the given element."""

    ...

    # Loop over the label and scale_factor pairs
    for label, factor in list(zip(labels, scale_factors)):

        ...

        # Launch a `PwCalculation` for each scaled structure
        calculations[label] = run(PwCalculation, **inputs)

    ...

    inputs = {
        label: result['output_parameters']
        for label, result in calculations.items()
    }
```

- Loop over some input parameters
- Launch a calculation for each iteration
- Use the results of the calculation ...
- Call it a day!

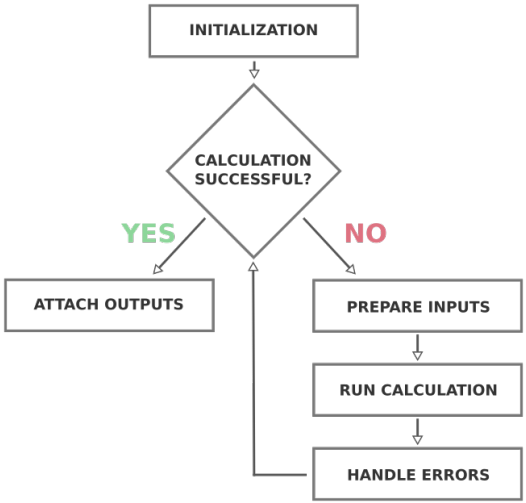
CONTAINS ONE GLARING MISTAKE IN REASONING

ASSUMES THAT CALCULATIONS **NEVER FAIL**

WHEN LIFE GIVES YOU LEMONS

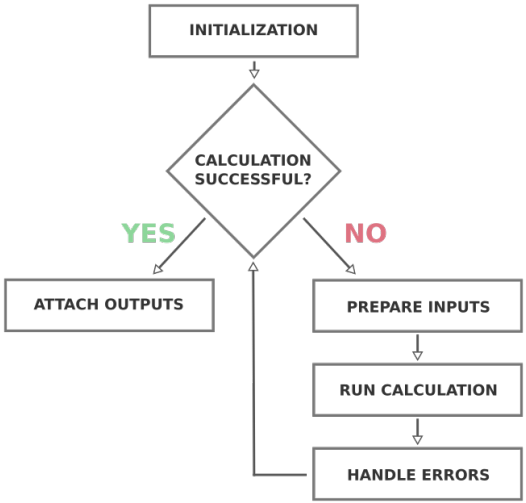


WHEN LIFE GIVES YOU LEMONS




```python
class PwBaseWorkChain(BaseRestartWorkChain):
    """Workchain to run a Quantum ESPRESSO pw.x calculation with automated error handling"""

    @classmethod
    def define(cls, spec):
        super(PwBaseWorkChain, cls).define(spec)
        spec.input('code', valid_type=orm.Code)
        ...

        spec.outline(
            cls.setup,
            while_(cls.should_run_calculation)(
                cls.prepare_calculation,
                cls.run_calculation,
                cls.inspect_calculation,
            ),
            cls.results,
        )

        ...
        spec.output('output_parameters', valid_type=orm.Dict)
```

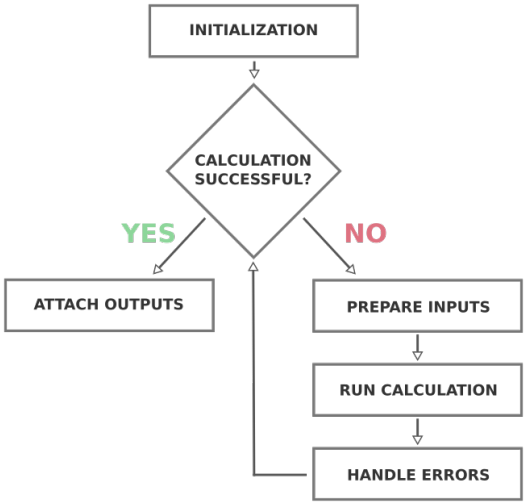
WHEN LIFE GIVES YOU LEMONS




```python
class PwBaseWorkChain(BaseRestartWorkChain):
    """Workchain to run a Quantum ESPRESSO pw.x calculation with automated error handling"""

    @classmethod
    def define(cls, spec):
        super(PwBaseWorkChain, cls).define(spec)
        spec.input('code', valid_type=orm.Code)
        ...

        spec.outline(
            cls.setup,
            while_(cls.should_run_calculation)(
                cls.prepare_calculation,
                cls.run_calculation,
                cls.inspect_calculation,
            ),
            cls.results,
        )

        ...
        spec.output('output_parameters', valid_type=orm.Dict)
```

- How would you implement `inspect_calculation`?

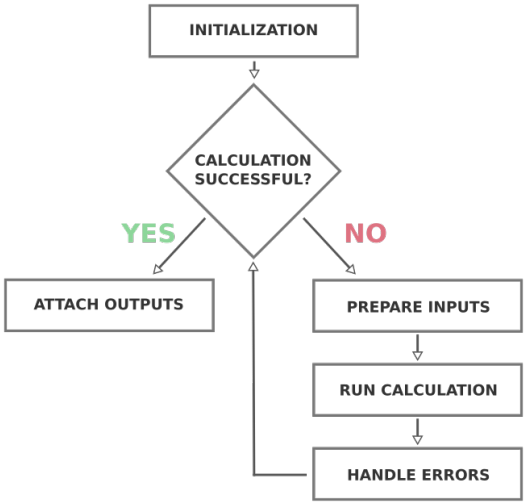
WHEN LIFE GIVES YOU LEMONS




```python
class PwBaseWorkChain(BaseRestartWorkChain):
    """Workchain to run a Quantum ESPRESSO pw.x calculation with automated error handling"""

    @classmethod
    def define(cls, spec):
        super(PwBaseWorkChain, cls).define(spec)
        spec.input('code', valid_type=orm.Code)
        ...

        spec.outline(
            cls.setup,
            while_(cls.should_run_calculation)(
                cls.prepare_calculation,
                cls.run_calculation,
                cls.inspect_calculation,
            ),
            cls.results,
        )

        ...
        spec.output('output_parameters', valid_type=orm.Dict)
```

- How would you implement `inspect_calculation`?
- How can it handle errors before the next calculation?

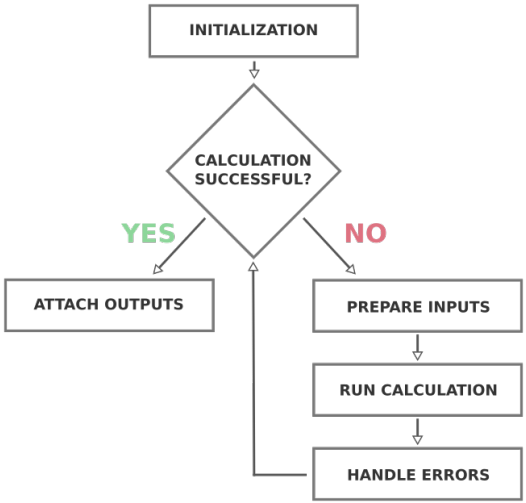
WHEN LIFE GIVES YOU LEMONS




```python
class PwBaseWorkChain(BaseRestartWorkChain):
    """Workchain to run a Quantum ESPRESSO pw.x calculation with automated error handling"""

    @classmethod
    def define(cls, spec):
        super(PwBaseWorkChain, cls).define(spec)
        spec.input('code', valid_type=orm.Code)
        ...

        spec.outline(
            cls.setup,
            while_(cls.should_run_calculation)(
                cls.prepare_calculation,
                cls.run_calculation,
                cls.inspect_calculation,
            ),
            cls.results,
        )

        ...
        spec.output('output_parameters', valid_type=orm.Dict)
```

- How would you implement `inspect_calculation`?
- How can it handle errors before the next calculation?
- One work group will focus on error handling in work chains

WHEN LIFE GIVES YOU LEMONS




```python
class PwBaseWorkChain(BaseRestartWorkChain):
    """Workchain to run a Quantum ESPRESSO pw.x calculation with automated error handling"""

    @classmethod
    def define(cls, spec):
        super(PwBaseWorkChain, cls).define(spec)
        spec.input('code', valid_type=orm.Code)
        ...

        spec.outline(
            cls.setup,
            while_(cls.should_run_calculation)(
                cls.prepare_calculation,
                cls.run_calculation,
                cls.inspect_calculation,
            ),
            cls.results,
        )

        ...
        spec.output('output_parameters', valid_type=orm.Dict)
```

- How would you implement `inspect_calculation`?
- How can it handle errors before the next calculation?
- One work group will focus on error handling in work chains
- After you have implemented your own idea, compare with `aiida-quantumespresso`

MODULARITY IS KING

The most important part of work chains is: **modularity**

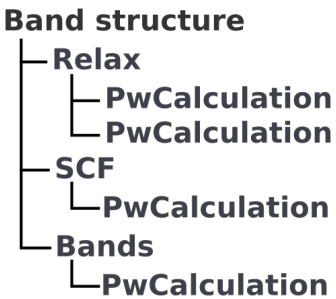
- Tackle a well-defined problem
- Create generic reusable components

MODULARITY IS KING

The most important part of work chains is: **modularity**

- Tackle a well-defined problem
- Create generic reusable components

Example: **compute the electronic band structure**

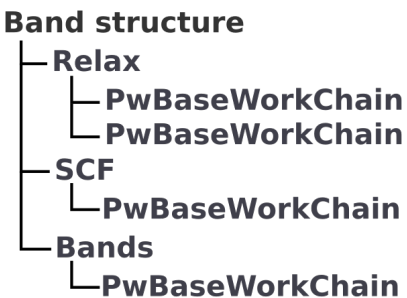


MODULARITY IS KING

The most important part of work chains is: **modularity**

- Tackle a well-defined problem
- Create generic reusable components

Example: **compute the electronic band structure**



MODULARITY IS KING

The most important part of work chains is: **modularity**

- Tackle a well-defined problem
- Create generic reusable components

Example: **compute the electronic band structure**

Band structure
- **Relax**
  - **PwBaseWorkChain**
  - **PwBaseWorkChain**
- **SCF**
  - **PwBaseWorkChain**
- **Bands**
  - **PwBaseWorkChain**

PwBaseWorkChain

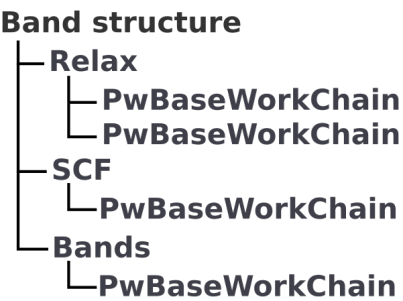
```python
class PwBaseWorkChain(BaseRestartWorkChain):
    """Workchain to run a Quantum ESPRESSO pw.x calculation with automated error handling"""

    @classmethod
    def define(cls, spec):
        super(PwBaseWorkChain, cls).define(spec)
        spec.input('code', valid_type=orm.Code)
        spec.input('structure', valid_type=orm.StructureData)
        spec.input('kpoints', valid_type=orm.KpointsData, required=False)
        spec.input('kpoints_distance', valid_type=orm.Float, required=False)
        spec.input('kpoints_force_parity', valid_type=orm.Bool, required=False)
        spec.input('parameters', valid_type=orm.Dict)
        spec.input_namespace('pseudos', required=False, dynamic=True)
        spec.input('pseudo_family', valid_type=orm.Str, required=False)
        spec.input('parent_folder', valid_type=orm.RemoteData, required=False)
        spec.input('vdw_table', valid_type=orm.SinglefileData, required=False)
        spec.input('settings', valid_type=orm.Dict, required=False)
        spec.input('options', valid_type=orm.Dict, required=False)
        spec.input('automatic_parallelization', valid_type=orm.Dict, required=False)
```

MODULARITY IS KING

The most important part of work chains is: **modularity**

- Tackle a well-defined problem
- Create generic reusable components

Example: **compute the electronic band structure**



PwBaseWorkChain

```python
class PwBaseWorkChain(BaseRestartWorkChain):
    """Workchain to run a Quantum ESPRESSO pw.x calculation with automated error handling"""

    @classmethod
    def define(cls, spec):
        super(PwBaseWorkChain, cls).define(spec)
        spec.input('code', valid_type=orm.Code)
        spec.input('structure', valid_type=orm.StructureData)
        spec.input('kpoints', valid_type=orm.KpointsData, required=False)
        spec.input('kpoints_distance', valid_type=orm.Float, required=False)
        spec.input('kpoints_force_parity', valid_type=orm.Bool, required=False)
        spec.input('parameters', valid_type=orm.Dict)
        spec.input_namespace('pseudos', required=False, dynamic=True)
        spec.input('pseudo_family', valid_type=orm.Str, required=False)
        spec.input('parent_folder', valid_type=orm.RemoteData, required=False)
        spec.input('vdw_table', valid_type=orm.SinglefileData, required=False)
        spec.input('settings', valid_type=orm.Dict, required=False)
        spec.input('options', valid_type=orm.Dict, required=False)
        spec.input('automatic_parallelization', valid_type=orm.Dict, required=False)
```

PwRelaxWorkChain

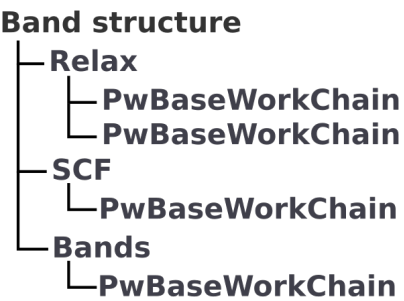
```python
class PwRelaxWorkChain(WorkChain):
    """Workchain to relax a structure using Quantum ESPRESSO pw.x"""

    @classmethod
    def define(cls, spec):
        super(PwRelaxWorkChain, cls).define(spec)
        spec.expose_inputs(PwBaseWorkChain, namespace='base', exclude=('structure',))
        spec.input('structure', valid_type=orm.StructureData)
        spec.input('final_scf', valid_type=orm.Bool, default=orm.Bool(False))
        spec.input('relaxation_scheme', valid_type=orm.Str, default=orm.Str('vc-relax'))
        spec.input('meta_convergence', valid_type=orm.Bool, default=orm.Bool(True))
        spec.input('max_meta_convergence_iterations', valid_type=orm.Int, default=orm.Int(5))
        spec.input('volume_convergence', valid_type=orm.Float, default=orm.Float(0.01))
```

MODULARITY IS KING

The most important part of work chains is: **modularity**

- Tackle a well-defined problem
- Create generic reusable components

Example: **compute the electronic band structure**



PwBaseWorkChain

```python
class PwBaseWorkChain(BaseRestartWorkChain):
    """Workchain to run a Quantum ESPRESSO pw.x calculation with automated error handling"""

    @classmethod
    def define(cls, spec):
        super(PwBaseWorkChain, cls).define(spec)
        spec.input('code', valid_type=orm.Code)
        spec.input('structure', valid_type=orm.StructureData)
        spec.input('kpoints', valid_type=orm.KpointsData, required=False)
        spec.input('kpoints_distance', valid_type=orm.Float, required=False)
        spec.input('kpoints_force_parity', valid_type=orm.Bool, required=False)
        spec.input('parameters', valid_type=orm.Dict)
        spec.input_namespace('pseudos', required=False, dynamic=True)
        spec.input('pseudo_family', valid_type=orm.Str, required=False)
        spec.input('parent_folder', valid_type=orm.RemoteData, required=False)
        spec.input('vdw_table', valid_type=orm.SinglefileData, required=False)
        spec.input('settings', valid_type=orm.Dict, required=False)
        spec.input('options', valid_type=orm.Dict, required=False)
        spec.input('automatic_parallelization', valid_type=orm.Dict, required=False)
```

PwBandsWorkChain

```python
class PwBandsWorkChain(WorkChain):
    """Workchain to compute a band structure for a given structure using Quantum ESPRESSO pw.x"""

    @classmethod
    def define(cls, spec):
        super(PwBandsWorkChain, cls).define(spec)
        spec.expose_inputs(PwRelaxWorkChain, namespace='relax', exclude=('structure',))
        spec.expose_inputs(PwBaseWorkChain, namespace='scf', exclude=('structure',))
        spec.expose_inputs(PwBaseWorkChain, namespace='bands', exclude=('structure',))
        spec.input('structure', valid_type=orm.StructureData)
        spec.input('nbands_factor', valid_type=orm.Float, default=orm.Float(1.2))
```

NESTED WORK CHAINS

What do the inputs look like of a nested work chain?

```
inputs = {
    'structure': structure,
    'relax': {
        'base': {
            'code': code,
            'pseudo_family': pseudo_family,
            'kpoints_distance': Float(0.5),
            'parameters': parameters,
        },
        'meta_convergence': Bool(False),
    },
    'scf': {
        'code': code,
        'pseudo_family': pseudo_family,
        'kpoints_distance': Float(0.2),
        'parameters': parameters,
    },
    'bands': {
        'code': code,
        'pseudo_family': pseudo_family,
        'kpoints_distance': Float(0.15),
        'parameters': parameters,
    }
}

submit(PwBandsWorkChain, **inputs)
```

NESTED WORK CHAINS

What do the inputs look like of a nested work chain?

```
inputs = {
    'structure': structure,
    'relax': {
        'base': {
            'code': code,
            'pseudo_family': pseudo_family,
            'kpoints_distance': Float(0.5),
            'parameters': parameters,
        },
        'meta_convergence': Bool(False),
    },
    'scf': {
        'code': code,
        'pseudo_family': pseudo_family,
        'kpoints_distance': Float(0.2),
        'parameters': parameters,
    },
    'bands': {
        'code': code,
        'pseudo_family': pseudo_family,
        'kpoints_distance': Float(0.15),
        'parameters': parameters,
    }
}

submit(PwBandsWorkChain, **inputs)
```

And how do we submit a sub process in a work chain?

```
def run_relax(self):
    """Run the PwRelaxWorkChain to run a relax PwCalculation."""
    inputs = AttributeDict(self.exposed_inputs(PwRelaxWorkChain, namespace='relax'))
    inputs.structure = self.ctx.current_structure

    running = self.submit(PwRelaxWorkChain, **inputs)

    self.report('launching PwRelaxWorkChain<{}>'.format(running.pk))

    return ToContext(workchain_relax=running)
```

CURRENT STATUS

We now have:

- A nice Python interface to your code (if not in Python already).

CURRENT STATUS

We now have:

- A nice Python interface to your code (if not in Python already).
- Data provenance.

CURRENT STATUS

We now have:

- A nice Python interface to your code (if not in Python already).
- Data provenance.
- Possibilities to do error handling and automatic restarts.

CURRENT STATUS

We now have:

- A nice Python interface to your code (if not in Python already).
- Data provenance.
- Possibilities to do error handling and automatic restarts.

But to be truly useful in science, we need more:

CURRENT STATUS

We now have:

- A nice Python interface to your code (if not in Python already).
- Data provenance.
- Possibilities to do error handling and automatic restarts.

But to be truly useful in science, we need more:

- Increase of productivity.

CURRENT STATUS

We now have:

- A nice Python interface to your code (if not in Python already).
- Data provenance.
- Possibilities to do error handling and automatic restarts.

But to be truly useful in science, we need more:

- Increase of productivity.
- More rigid way of performing computations, data analysis with less possibilities of shortcuts.

CURRENT STATUS

We now have:

- A nice Python interface to your code (if not in Python already).
- Data provenance.
- Possibilities to do error handling and automatic restarts.

But to be truly useful in science, we need more:

- Increase of productivity.
- More rigid way of performing computations, data analysis with less possibilities of shortcuts.
- More time to analyze, interpret and design new studies.

CURRENT STATUS

We now have:

- A nice Python interface to your code (if not in Python already).
- Data provenance.
- Possibilities to do error handling and automatic restarts.

But to be truly useful in science, we need more:

- Increase of productivity.
- More rigid way of performing computations, data analysis with less possibilities of shortcuts.
- More time to analyze, interpret and design new studies.
- New novel ways to traverse domains (in material science, enable multi-scale).

CURRENT STATUS

We now have:

- A nice Python interface to your code (if not in Python already).
- Data provenance.
- Possibilities to do error handling and automatic restarts.

But to be truly useful in science, we need more:

- Increase of productivity.
- More rigid way of performing computations, data analysis with less possibilities of shortcuts.
- More time to analyze, interpret and design new studies.
- New novel ways to traverse domains (in material science, enable multi-scale).
- Possibilities of utilizing different codes for different purposes, interchangeably.

CURRENT STATUS

We now have:

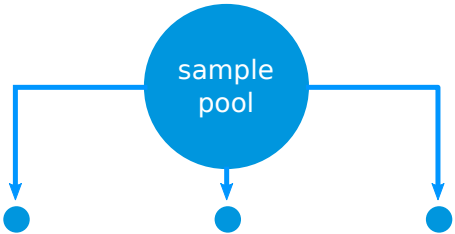
- A nice Python interface to your code (if not in Python already).
- Data provenance.
- Possibilities to do error handling and automatic restarts.

But to be truly useful in science, we need more:

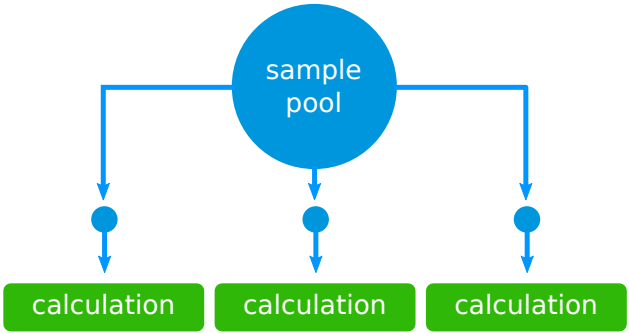
- Increase of productivity.
- More rigid way of performing computations, data analysis with less possibilities of shortcuts.
- More time to analyze, interpret and design new studies.
- New novel ways to traverse domains (in material science, enable multi-scale).
- Possibilities of utilizing different codes for different purposes, interchangeably.
- If possible, make the work to be performed independent of the code.

DAILY LIFE OF A COMPUTATIONAL SCIENTIST

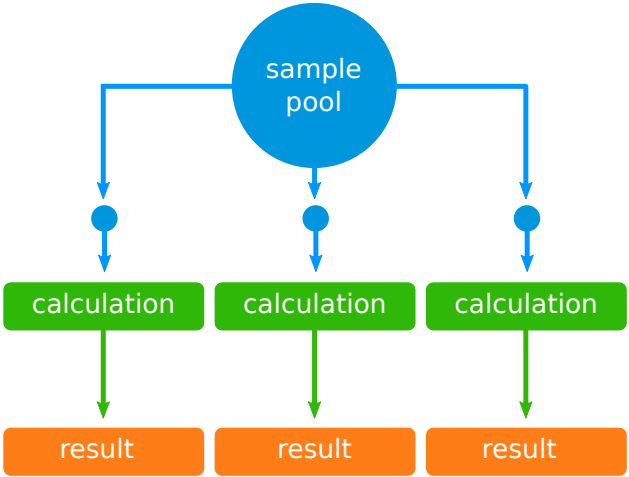
DAILY LIFE OF A COMPUTATIONAL SCIENTIST



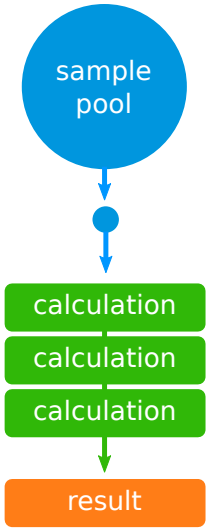
DAILY LIFE OF A COMPUTATIONAL SCIENTIST



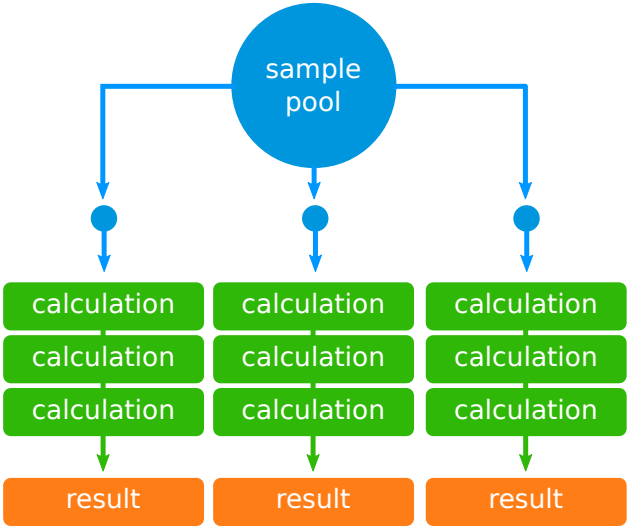
DAILY LIFE OF A COMPUTATIONAL SCIENTIST



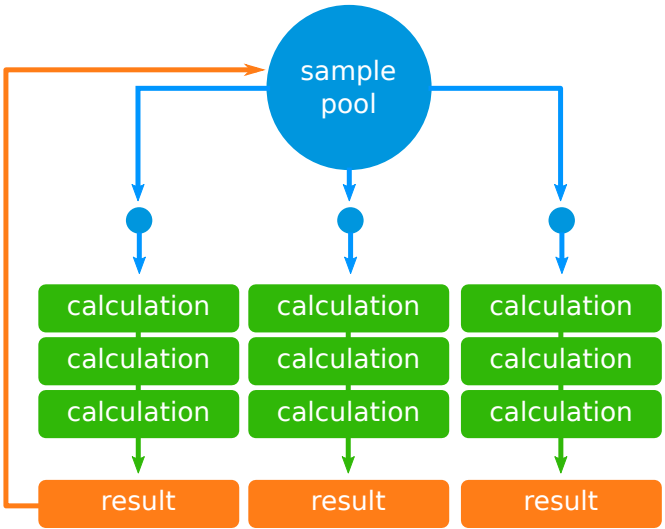
DAILY LIFE OF A COMPUTATIONAL SCIENTIST



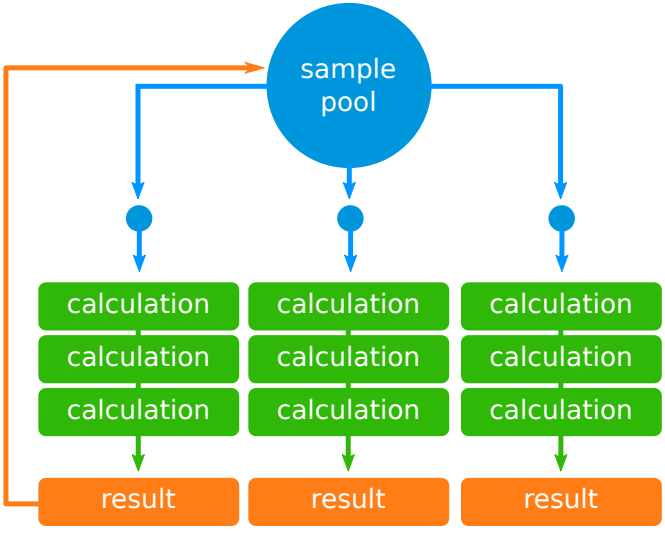
DAILY LIFE OF A COMPUTATIONAL SCIENTIST



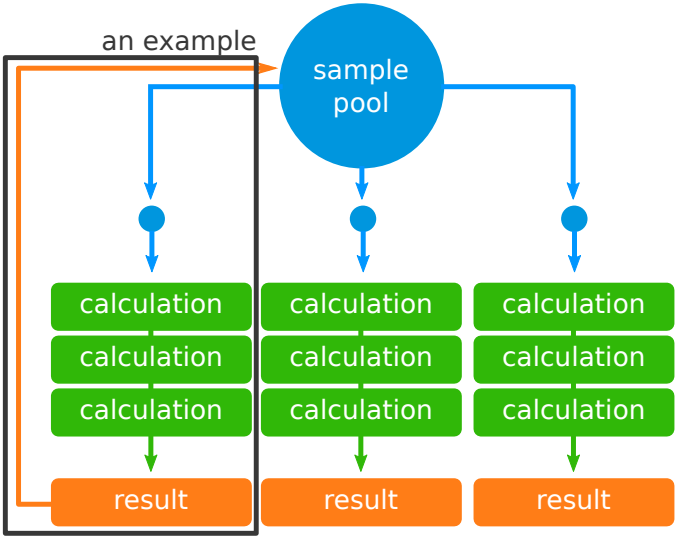
DAILY LIFE OF A COMPUTATIONAL SCIENTIST



DAILY LIFE OF A COMPUTATIONAL SCIENTIST



DAILY LIFE OF A COMPUTATIONAL SCIENTIST



AN EXAMPLE - MATERIAL SCIENCE - VASP

- We start with a structure (can of course have been generate by some other workflow).

AN EXAMPLE - MATERIAL SCIENCE - VASP

- We start with a structure (can of course have been generate by some other workflow).
- The restart workchain (currently independent of code).

AN EXAMPLE - MATERIAL SCIENCE - VASP

- We start with a structure (can of course have been generate by some other workflow).
- The restart workchain (currently independent of code).
- The VASP workchain. Handles the setup of the VASP calculation (make sure inputs etc. are passed correctly to the VASP calculation plugin). Should also auto-parallelize.

AN EXAMPLE - MATERIAL SCIENCE - VASP

- We start with a structure (can of course have been generate by some other workflow).
- The restart workchain (currently independent of code).
- The VASP workchain. Handles the setup of the VASP calculation (make sure inputs etc. are passed correctly to the VASP calculation plugin). Should also auto-parallelize.
- The verify workchain. Handles the check of physical principles outside of VASP.

AN EXAMPLE - MATERIAL SCIENCE - VASP

- We start with a structure (can of course have been generate by some other workflow).
- The restart workchain (currently independent of code).
- The VASP workchain. Handles the setup of the VASP calculation (make sure inputs etc. are passed correctly to the VASP calculation plugin). Should also auto-parallelize.
- The verify workchain. Handles the check of physical principles outside of VASP.
- The relaxation workchain. Handles relaxations of the structure.

AN EXAMPLE - MATERIAL SCIENCE - VASP

- We start with a structure (can of course have been generate by some other workflow).
- The restart workchain (currently independent of code).
- The VASP workchain. Handles the setup of the VASP calculation (make sure inputs etc. are passed correctly to the VASP calculation plugin). Should also auto-parallelize.
- The verify workchain. Handles the check of physical principles outside of VASP.
- The relaxation workchain. Handles relaxations of the structure.
- The convergence workchain. Determines convergence parameters (typically the plane wave cutoff and k-point grid).

AN EXAMPLE - MATERIAL SCIENCE - VASP

- We start with a structure (can of course have been generate by some other workflow).
- The restart workchain (currently independent of code).
- The VASP workchain. Handles the setup of the VASP calculation (make sure inputs etc. are passed correctly to the VASP calculation plugin). Should also auto-parallelize/run-time opt.
- The verify workchain. Handles the check of physical principles outside of VASP.
- The relaxation workchain. Handles relaxations of the structure.
- The convergence workchain. Determines convergence parameters (typically the plane wave cutoff and k-point grid).

AN EXAMPLE - MATERIAL SCIENCE - VASP

- We start with a structure (can of course have been generate by some other workflow).
- The restart workchain (currently independent of code).
- The VASP workchain. Handles the setup of the VASP calculation (make sure inputs etc. are passed correctly to the VASP calculation plugin). Should also auto-parallelize/run-time opt.
- The verify workchain. Handles the check of physical principles outside of VASP.
- The relaxation workchain. Handles relaxations of the structure.
- The convergence workchain. Determines convergence parameters (typically the plane wave cutoff and k-point grid).

AN EXAMPLE - MATERIAL SCIENCE - VASP

- We start with a structure (can of course have been generate by some other workflow).
- The restart workchain (currently independent of code).
- The VASP workchain. Handles the setup of the VASP calculation (make sure inputs etc. are passed correctly to the VASP calculation plugin). Should also auto-parallelize/run-time opt.
- The verify workchain. Handles the check of physical principles outside of VASP.
- The relaxation workchain. Handles relaxations of the structure.
- The convergence workchain. Determines convergence parameters (typically the plane wave cutoff and k-point grid).

AN EXAMPLE - MATERIAL SCIENCE - VASP

- We start with a structure (can of course have been generate by some other workflow).
- The restart workchain (currently independent of code).
- The VASP workchain. Handles the setup of the VASP calculation (make sure inputs etc. are passed correctly to the VASP calculation plugin). Should also auto-parallelize/run-time opt.
- The verify workchain. Handles the check of physical principles outside of VASP.
- The relaxation workchain. Handles relaxations of the structure.
- The convergence workchain. Determines convergence parameters (typically the plane wave cutoff and k-point grid).

AN EXAMPLE - MATERIAL SCIENCE - VASP

- The material scientist should not need to worry about these things.

AN EXAMPLE - MATERIAL SCIENCE - VASP

- The material scientist should not need to worry about these things.
- Mostly numerics.

AN EXAMPLE - MATERIAL SCIENCE - VASP

- The material scientist should not need to worry about these things.
- Mostly numerics.
- Probably a lot of error correction, restarts etc. based on knowledge and input/output analysis. Today, knowledge of this is regarded as highly valuable competence.

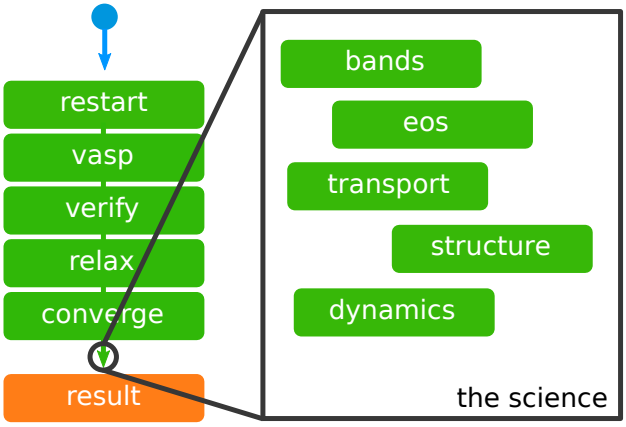
AN EXAMPLE - MATERIAL SCIENCE - VASP

- The material scientist should not need to worry about these things.
- Mostly numerics.
- Probably a lot of error correction, restarts etc. based on knowledge and input/output analysis. Today, knowledge of this is regarded as highly valuable competence.
- Minimal base workchain set for calculations with no prior knowledge.

AN EXAMPLE - MATERIAL SCIENCE - VASP

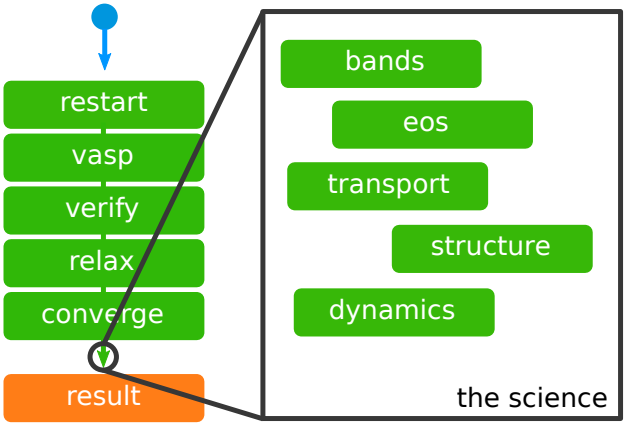
- The material scientist should not need to worry about these things.
- Mostly numerics.
- Probably a lot of error correction, restarts etc. based on knowledge and input/output analysis. Today, knowledge of this is regarded as highly valuable competence.
- Minimal base workchain set for calculations with no prior knowledge.
- Should in principle be independent of code and, in fact also the technique used.

AN EXAMPLE - MATERIAL SCIENCE - VASP



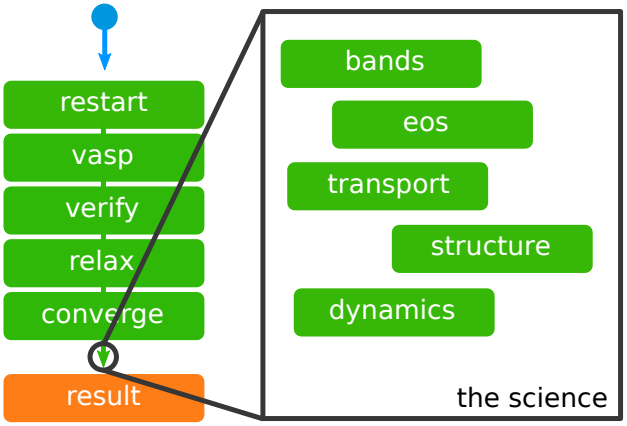
- Refactor code.

AN EXAMPLE - MATERIAL SCIENCE - VASP



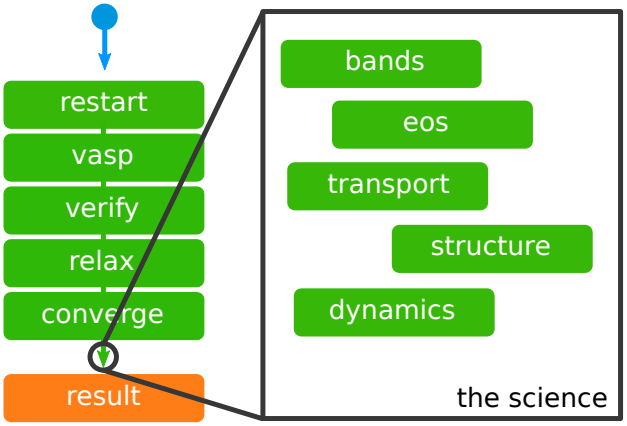
- Refactor code.
- Refactor science (lean science).

AN EXAMPLE - MATERIAL SCIENCE - VASP



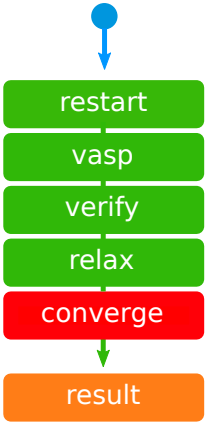
- Refactor code.
- Refactor science (lean science).
- Try to make the scientific workchains reusable, preferable independent of code.

AN EXAMPLE - MATERIAL SCIENCE - VASP



- Refactor code.
- Refactor science (lean science).
- Try to make the scientific workchains reusable, preferable independent of code.
- Plugin developers should unite and try to merge the codes within one class.

AN EXAMPLE - MATERIAL SCIENCE - VASP - THE CONVERGENCE WORKCHAIN



- Convergence tests should always be done (how many are actually doing this regularly?). For plane wave DFT codes, this typically covers the plane wave cutoff and k-point sampling.

So let us have a look at a bit more complex workchain outline.

AN EXAMPLE - MATERIAL SCIENCE - VASP - THE CONVERGENCE WORKCHAIN

- Convergence tests should always be done (how many are actually doing this regularly?). For plane wave DFT codes, this typically covers the plane wave cutoff and k-point sampling.
- Should be able to perform convergence tests on the relevant parameters (of course not always possible due to limited resources).

So let us have a look at a bit more complex workchain outline.

AN EXAMPLE - MATERIAL SCIENCE - VASP - THE CONVERGENCE WORKCHAIN

- Convergence tests should always be done (how many are actually doing this regularly?). For plane wave DFT codes, this typically covers the plane wave cutoff and k-point sampling.
- Should be able to perform convergence tests on the relevant parameters (of course not always possible due to limited resources).
- Should not harden the result more than necessary (waste computational resources, bad for the environment).

So let us have a look at a bit more complex workchain outline.

AN EXAMPLE - MATERIAL SCIENCE - VASP - THE CONVERGENCE WORKCHAIN

- Convergence tests should always be done (how many are actually doing this regularly?). For plane wave DFT codes, this typically covers the plane wave cutoff and k-point sampling.
- Should be able to perform convergence tests on the relevant parameters (of course not always possible due to limited resources).
- Should not harden the result more than necessary (waste computational resources, bad for the environment).
- Necessary to also test relative convergence, e.g. say:

$$\Delta E_g = E_g^0 - E_g^1$$

So let us have a look at a bit more complex workchain outline.

AN EXAMPLE - MATERIAL SCIENCE - VASP - THE CONVERGENCE WORKCHAIN


```
spec.outline(
    cls.initialize,
    if_(cls.run_conv_calcs)(
        while_(cls.run_pw_conv_calcs)(
            cls.init_pw_conv_calc,
            cls.init_next_workchain,
            cls.run_next_workchain,
            cls.results_pw_conv_calc
        ),
        cls.analyze_pw_conv,
        while_(cls.run_kpoints_conv_calcs)(
            cls.init_kpoints_conv_calc,
            cls.init_next_workchain,
            cls.run_next_workchain,
            cls.results_kpoints_conv_calc
        ),
```

AN EXAMPLE - MATERIAL SCIENCE - VASP - THE CONVERGENCE WORKCHAIN


```
spec.outline(
    cls.initialize,
    if_(cls.run_conv_calcs)(
        while_(cls.run_pw_conv_calcs)(
            cls.init_pw_conv_calc,
            cls.init_next_workchain,
            cls.run_next_workchain,
            cls.results_pw_conv_calc
        ),
        cls.analyze_pw_conv,
        while_(cls.run_kpoints_conv_calcs)(
            cls.init_kpoints_conv_calc,
            cls.init_next_workchain,
            cls.run_next_workchain,
            cls.results_kpoints_conv_calc
        ),
```

AN EXAMPLE - MATERIAL SCIENCE - VASP - THE CONVERGENCE WORKCHAIN


```
    cls.init_disp_conv,
    while_(cls.run_pw_conv_disp_calcs)(
        cls.init_pw_conv_calc,
        cls.init_next_workchain,
        cls.run_next_workchain,
        cls.results_pw_conv_calc
    ),
    if_(cls.analyze_pw_after_disp)(
        cls.analyze_pw_conv,
    ),
    while_(cls.run_kpoints_conv_disp_calcs)(
        cls.init_kpoints_conv_calc,
        cls.init_next_workchain,
        cls.run_next_workchain,
        cls.results_kpoints_conv_calc
    ),
```

AN EXAMPLE - MATERIAL SCIENCE - VASP - THE CONVERGENCE WORKCHAIN


```
            cls.init_comp_conv,
            while_(cls.run_pw_conv_comp_calcs)(
                cls.init_pw_conv_calc,
                cls.init_next_workchain,
                cls.run_next_workchain,
                cls.results_pw_conv_calc
            ),
            if_(cls.analyze_pw_after_comp)(
                cls.analyze_pw_conv,
            ),
            while_(cls.run_kpoints_conv_comp_calcs)(
                cls.init_kpoints_conv_calc,
                cls.init_next_workchain,
                cls.run_next_workchain,
                cls.results_kpoints_conv_calc
            ),
            cls.analyze_conv,
            cls.store_conv,
        ),
```

AN EXAMPLE - MATERIAL SCIENCE - VASP - THE CONVERGENCE WORKCHAIN


```
cls.init_converged,
cls.init_next_workchain,
cls.run_next_workchain,
cls.verify_next_workchain,
cls.results,
cls.finalize
)
```

AN EXAMPLE - MATERIAL SCIENCE - VASP - EXPOSING INPUTS/OUTPUTS

- The workchain specifications

  ```
  spec.input
  spec.output
  ```

  can potentially yield a lot of boiler plate code that is hard to maintain for nested workchains.
- AiiDA to the rescue

  ```
  spec.expose_inputs
  spec.export_outputs
  ```
- For instance if we want to define the same outputs in the converge workchain as defined in the relax workchain we do

  ```
  spec.export_outputs(relax)
  ```

  in the define section of the converge workchain.
- Can also exclude certain input/output in case that is for instance modified in that particular workchain. Check the manual.