Writing workflows in AiiDA

A short introduction to AiiDA's engine

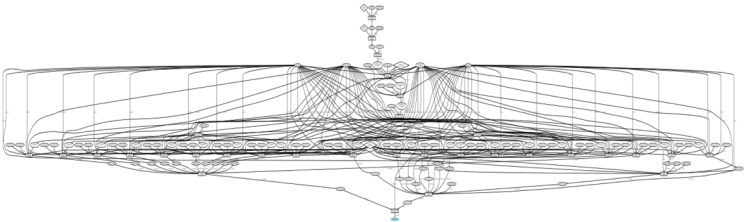
Sebastiaan Huber

AiiDA Workflow Tutorial May 22nd 2019

AiiDA

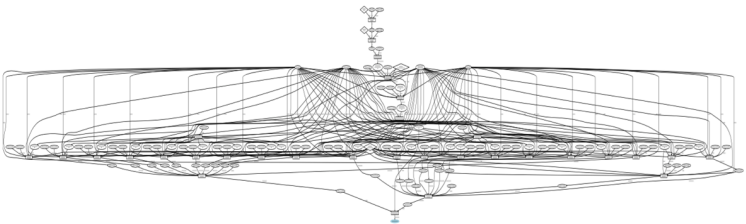
Keeping data provenance is important...

Keeping data provenance is important...



... but imagine having to manually link everything up

Keeping data provenance is important...



... but imagine having to manually link everything up

AiiDA **Automated** Interactive Infrastructure and Database

AUTOMATED PROVENANCE: CALCULATIONS

Imagine the following simple arithmetic problem:

Add two numbers and multiply the sum by third

```
def add(x, y):
    return x + y

def multiply(x, y):
    return x * y

result = multiply(add(1, 2), 3)
```

AUTOMATED PROVENANCE: CALCULATIONS

Imagine the following simple arithmetic problem:

Add two numbers and multiply the sum by third

```
def add(x, y):
    return x + y

def multiply(x, y):
    return x * y

result = multiply(add(1, 2), 3)
```

Just apply the `calcfunction` decorator...

```
from aiida.engine import calcfunction

@calcfunction
def add(x, y):
    return x + y

@calcfunction
def multiply(x, y):
    return x * y

result = multiply(add(1, 2), 3)
```

AUTOMATED PROVENANCE: CALCULATIONS

Imagine the following simple arithmetic problem:

Add two numbers and multiply the sum by third

```
def add(x, y):
    return x + y

def multiply(x, y):
    return x * y

result = multiply(add(1, 2), 3)
```

Just apply the `calcfunction` decorator...

```
from aiida.engine import calcfunction

@calcfunction
def add(x, y):
    return x + y

@calcfunction
def multiply(x, y):
    return x * y

result = multiply(add(1, 2), 3)
```

... and pass storable data types when calling

```
from aiida.engine import calcfunction
from aiida.orm import Int

@calcfunction
def add(x, y):
    return x + y

@calcfunction
def multiply(x, y):
    return x * y

result = multiply(add(Int(1), Int(2)), Int(3))
```

AUTOMATED PROVENANCE: CALCULATIONS

Imagine the following simple arithmetic problem:

Add two numbers and multiply the sum by third

```
def add(x, y):
    return x + y

def multiply(x, y):
    return x * y

result = multiply(add(1, 2), 3)
```

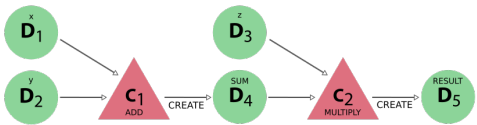
Just apply the `calcfunction` decorator...

```
from aiida.engine import calcfunction

@calcfunction
def add(x, y):
    return x + y

@calcfunction
def multiply(x, y):
    return x * y

result = multiply(add(1, 2), 3)
```

... and pass storable data types when calling

```
from aiida.engine import calcfunction
from aiida.orm import Int

@calcfunction
def add(x, y):
    return x + y

@calcfunction
def multiply(x, y):
    return x * y

result = multiply(add(Int(1), Int(2)), Int(3))
```

Provenance is automatically stored in the graph



AUTOMATED PROVENANCE: CALCULATIONS

Imagine the following simple arithmetic problem:

Add two numbers and multiply the sum by third

```
def add(x, y):
    return x + y

def multiply(x, y):
    return x * y

result = multiply(add(1, 2), 3)
```

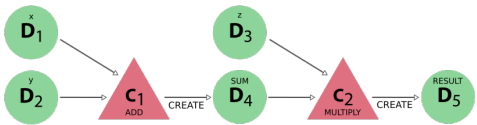
Just apply the `calcfunction` decorator...

```
from aiida.engine import calcfunction

@calcfunction
def add(x, y):
    return x + y

@calcfunction
def multiply(x, y):
    return x * y

result = multiply(add(1, 2), 3)
```

... and pass storable data types when calling

```
from aiida.engine import calcfunction
from aiida.orm import Int

@calcfunction
def add(x, y):
    return x + y

@calcfunction
def multiply(x, y):
    return x * y

result = multiply(add(Int(1), Int(2)), Int(3))
```

Provenance is automatically stored in the graph



Directed **A**cyclic **G**raph

- Directed: inputs go in, and outputs come out
- Acyclic: causality principle forbids an output being its own input

AUTOMATED PROVENANCE: EXTERNAL CODES

But not all code is well-suited as python code:

What about running codes external to AiiDA?

```
#!/bin/bash
# Read two integers from file 'aiida.in` and echo their sum
x=$(cat aiida.in | awk '{print $1}')
y=$(cat aiida.in | awk '{print $2}')
echo $(( $x + $y ))
```

AUTOMATED PROVENANCE: EXTERNAL CODES

But not all code is well-suited as python code:

What about running codes external to AiiDA?

```
#!/bin/bash
# Read two integers from file 'aiida.in` and echo their sum
x=$(cat aiida.in | awk '{print $1}')
y=$(cat aiida.in | awk '{print $2}')
echo $(( $x + $y ))
```

Implementation is different, but running very similar...

```
from aiida.engine import run
from aiida.orm import load_code, Int
from aiida.plugins import CalculationFactory

ArithmeticAddCalculation = CalculationFactory('arithmetic.add')

inputs = {
    'code': load_code('add@localhost'),
    'x': Int(1),
    'y': Int(2),
}

run(ArithmeticAddCalculation, **inputs)
```

AUTOMATED PROVENANCE: EXTERNAL CODES

But not all code is well-suited as python code:

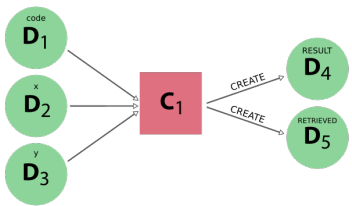
What about running codes external to AiiDA?

```
#!/bin/bash
# Read two integers from file 'aiida.in` and echo their sum
x=$(cat aiida.in | awk '{print $1}')
y=$(cat aiida.in | awk '{print $2}')
echo $(( $x + $y ))
```

Implementation is different, but running very similar...

```
from aiida.engine import run
from aiida.orm import load_code, Int
from aiida.plugins import CalculationFactory

ArithmeticAddCalculation = CalculationFactory('arithmetic.add')

inputs = {
    'code': load_code('add@localhost'),
    'x': Int(1),
    'y': Int(2),
}

run(ArithmeticAddCalculation, **inputs)
```

Generated provenance similar to that of calculation function



AUTOMATED PROVENANCE: EXTERNAL CODES

But not all code is well-suited as python code:

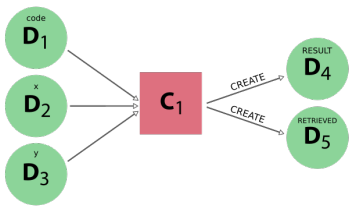
What about running codes external to AiiDA?

```
#!/bin/bash
# Read two integers from file 'aiida.in` and echo their sum
x=$(cat aiida.in | awk '{print $1}')
y=$(cat aiida.in | awk '{print $2}')
echo $(( $x + $y ))
```

Implementation is different, but running very similar...

```
from aiida.engine import run
from aiida.orm import load_code, Int
from aiida.plugins import CalculationFactory

ArithmeticAddCalculation = CalculationFactory('arithmetic.add')

inputs = {
    'code': load_code('add@localhost'),
    'x': Int(1),
    'y': Int(2),
}

run(ArithmeticAddCalculation, **inputs)
```

Generated provenance similar to that of calculation function



- Can be run on remote machines through job scheduler
- Implementation independent of job scheduler
- To change machine, just change the 'code' input
- Implementation focus of one of work groups tomorrow

AUTOMATED PROVENANCE: EXTERNAL CODES

But not all code is well-suited as python code:

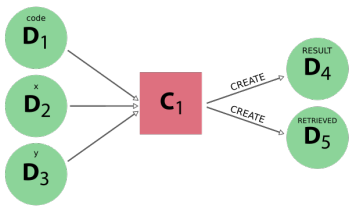
What about running codes external to AiiDA?

```
#!/bin/bash
# Read two integers from file 'aiida.in` and echo their sum
x=$(cat aiida.in | awk '{print $1}')
y=$(cat aiida.in | awk '{print $2}')
echo $(( $x + $y ))
```

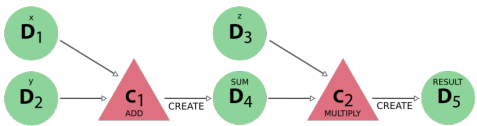
Implementation is different, but running very similar...

```
from aiida.engine import run
from aiida.orm import load_code, Int
from aiida.plugins import CalculationFactory

ArithmeticAddCalculation = CalculationFactory('arithmetic.add')

inputs = {
    'code': load_code('add@localhost'),
    'x': Int(1),
    'y': Int(2),
}

run(ArithmeticAddCalculation, **inputs)
```

Generated provenance similar to that of calculation function



Let's go back to our add-multiple example



Individual sequence of calculations recorded...

... but not the 'how' or 'why'

AUTOMATED PROVENANCE: WORKFLOWS

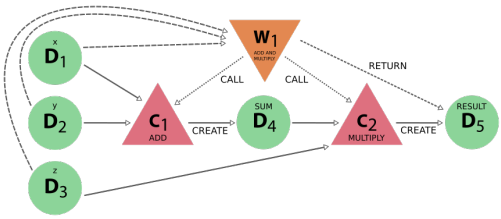
Work function can be used to store *logical* provenance

```
from aiida.engine import calcfunction, workfunction
from aiida.orm import Int

@calcfunction
def add(x, y):
    return Int(x + y)

@calcfunction
def multiply(x, y):
    return Int(x * y)

@workfunction
def add_and_multiply(x, y, z):
    sum = add(x, y)
    product = multiply(sum, z)
    return product

result = add_and_multiply(Int(1), Int(2), Int(3))
```

AUTOMATED PROVENANCE: WORKFLOWS

Work function can be used to store *logical* provenance

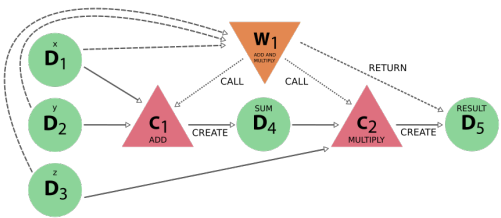
```python
from aiida.engine import calcfunction, workfunction
from aiida.orm import Int

@calcfunction
def add(x, y):
    return Int(x + y)

@calcfunction
def multiply(x, y):
    return Int(x * y)

@workfunction
def add_and_multiply(x, y, z):
    sum = add(x, y)
    product = multiply(sum, z)
    return product

result = add_and_multiply(Int(1), Int(2), Int(3))
```



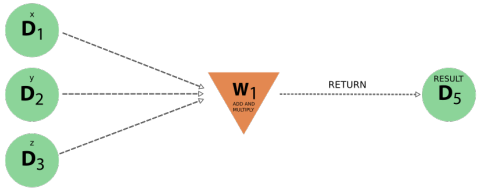
AUTOMATED PROVENANCE: WORKFLOWS

Work function can be used to store *logical* provenance

```python
from aiida.engine import calcfunction, workfunction
from aiida.orm import Int

@calcfunction
def add(x, y):
    return Int(x + y)

@calcfunction
def multiply(x, y):
    return Int(x * y)

@workfunction
def add_and_multiply(x, y, z):
    sum = add(x, y)
    product = multiply(sum, z)
    return product

result = add_and_multiply(Int(1), Int(2), Int(3))
```



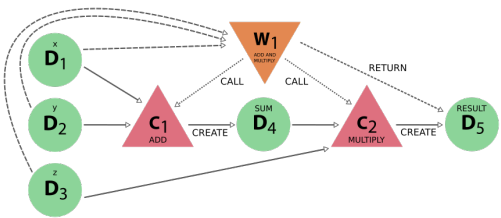
Logical provenance allows to 'hide' complexity



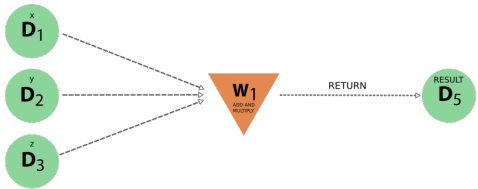
AUTOMATED PROVENANCE: WORKFLOWS

Work function can be used to store *logical* provenance

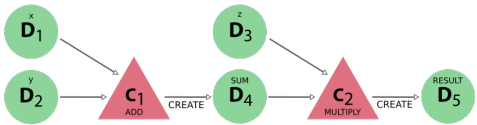
```python
from aiida.engine import calcfunction, workfunction
from aiida.orm import Int

@calcfunction
def add(x, y):
    return Int(x + y)

@calcfunction
def multiply(x, y):
    return Int(x * y)

@workfunction
def add_and_multiply(x, y, z):
    sum = add(x, y)
    product = multiply(sum, z)
    return product

result = add_and_multiply(Int(1), Int(2), Int(3))
```



Logical provenance allows to 'hide' complexity



Or by ignoring it, retrieve the original data provenance



AUTOMATED PROVENANCE: WORKFLOWS

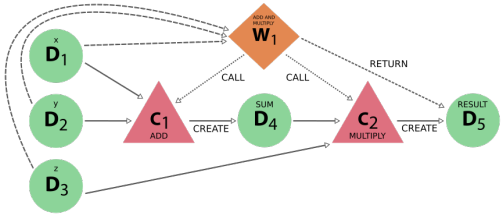
Work chains achieve the same but save progress in between steps

```python
from aiida.engine import WorkChain, calcfunction
from aiida.orm import Int

@calcfunction
def add(x, y):
    return Int(x + y)

@calcfunction
def multiply(x, y):
    return Int(x * y)

class AddAndMultiplyWorkChain(WorkChain):

    @classmethod
    def define(cls, spec):
        super(AddAndMultiplyWorkChain, cls).define(spec)
        spec.input('x')
        spec.input('y')
        spec.input('z')
        spec.outline(
            cls.add,
            cls.multiply,
            cls.results,
        )
        spec.output('result')

    def add(self):
        self.ctx.sum = add(self.inputs.x, self.inputs.y)

    def multiply(self):
        self.ctx.product = multiply(self.ctx.sum, self.inputs.z)

    def results(self):
        self.out('result', self.ctx.product)
```

AUTOMATED PROVENANCE: WORKFLOWS

Work chains achieve the same but save progress in between steps

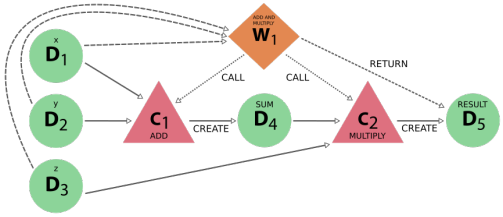
```python
from aiida.engine import WorkChain, calcfunction
from aiida.orm import Int

@calcfunction
def add(x, y):
    return Int(x + y)

@calcfunction
def multiply(x, y):
    return Int(x * y)

class AddAndMultiplyWorkChain(WorkChain):

    @classmethod
    def define(cls, spec):
        super(AddAndMultiplyWorkChain, cls).define(spec)
        spec.input('x')
        spec.input('y')
        spec.input('z')
        spec.outline(
            cls.add,
            cls.multiply,
            cls.results,
        )
        spec.output('result')

    def add(self):
        self.ctx.sum = add(self.inputs.x, self.inputs.y)

    def multiply(self):
        self.ctx.product = multiply(self.ctx.sum, self.inputs.z)

    def results(self):
        self.out('result', self.ctx.product)
```



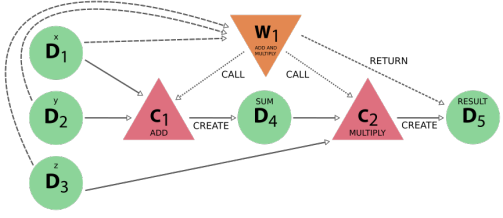
AUTOMATED PROVENANCE: WORKFLOWS

Work chains achieve the same but save progress in between steps

```python
from aiida.engine import WorkChain, calcfunction
from aiida.orm import Int

@calcfunction
def add(x, y):
    return Int(x + y)

@calcfunction
def multiply(x, y):
    return Int(x * y)

class AddAndMultiplyWorkChain(WorkChain):

    @classmethod
    def define(cls, spec):
        super(AddAndMultiplyWorkChain, cls).define(spec)
        spec.input('x')
        spec.input('y')
        spec.input('z')
        spec.outline(
            cls.add,
            cls.multiply,
            cls.results,
        )
        spec.output('result')

    def add(self):
        self.ctx.sum = add(self.inputs.x, self.inputs.y)

    def multiply(self):
        self.ctx.product = multiply(self.ctx.sum, self.inputs.z)

    def results(self):
        self.out('result', self.ctx.product)
```



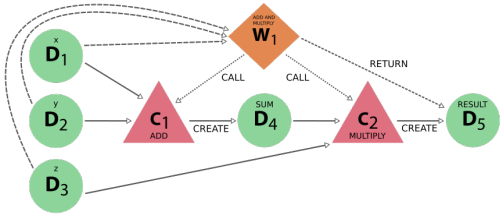
Only difference with work function solution is node type



AUTOMATED PROVENANCE: WORKFLOWS

Work chains achieve the same but save progress in between steps

```python
from aiida.engine import WorkChain, calcfunction
from aiida.orm import Int

@calcfunction
def add(x, y):
    return Int(x + y)

@calcfunction
def multiply(x, y):
    return Int(x * y)

class AddAndMultiplyWorkChain(WorkChain):

    @classmethod
    def define(cls, spec):
        super(AddAndMultiplyWorkChain, cls).define(spec)
        spec.input('x')
        spec.input('y')
        spec.input('z')
        spec.outline(
            cls.add,
            cls.multiply,
            cls.results,
        )
        spec.output('result')

    def add(self):
        self.ctx.sum = add(self.inputs.x, self.inputs.y)

    def multiply(self):
        self.ctx.product = multiply(self.ctx.sum, self.inputs.z)

    def results(self):
        self.out('result', self.ctx.product)
```



Many advantages over work function

- Can be submitted to the daemon
- Progress is saved between steps in checkpoints
- Process specification gives succinct but clear summary
- Captures the scientific knowledge and can be re-run
- Can be re-used as building block in more complex workflows

AUTOMATED PROVENANCE: LOSING PROVENANCE

Workflows cannot *create* new data.
Doing so anyway, will cause loss of provenance

Take first example: add two numbers and multiply with third...
... but now compute the product *inside* the work function

```
from aiida.engine import calcfunction, workfunction
from aiida.orm import Int

@calcfunction
def add(x, y):
    return Int(x + y)

@workfunction
def add_and_multiply(x, y, z):
    sum = add(x, y)
    product = Int(sum * z)
    return product.store()

result = add_and_multiply(Int(1), Int(2), Int(3))
```

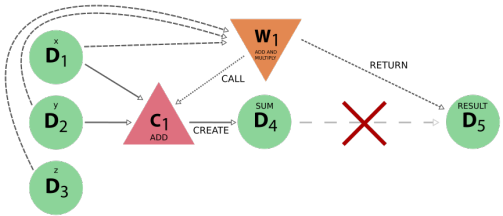
AUTOMATED PROVENANCE: LOSING PROVENANCE

Workflows cannot *create* new data.
Doing so anyway, will cause loss of provenance

Take first example: add two numbers and multiply with third...
... but now compute the product *inside* the work function

```python
from aiida.engine import calcfunction, workfunction
from aiida.orm import Int

@calcfunction
def add(x, y):
    return Int(x + y)

@workfunction
def add_and_multiply(x, y, z):
    sum = add(x, y)
    product = Int(sum * z)
    return product.store()

result = add_and_multiply(Int(1), Int(2), Int(3))
```

Provenance will miss link between sum and final result



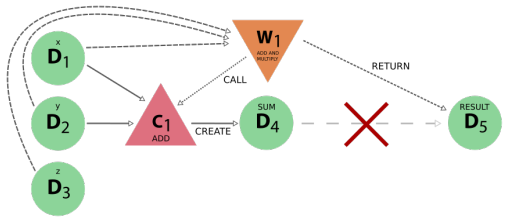
AUTOMATED PROVENANCE: LOSING PROVENANCE

Workflows cannot *create* new data.
Doing so anyway, will cause loss of provenance

Take first example: add two numbers and multiply with third...
... but now compute the product *inside* the work function

```python
from aiida.engine import calcfunction, workfunction
from aiida.orm import Int

@calcfunction
def add(x, y):
    return Int(x + y)

@workfunction
def add_and_multiply(x, y, z):
    sum = add(x, y)
    product = Int(sum * z)
    return product.store()

result = add_and_multiply(Int(1), Int(2), Int(3))
```

Provenance will miss link between sum and final result



Why don't we just give the workflows the power to **create**?

WHY A DIFFERENCE BETWEEN CALCULATIONS AND WORKFLOWS?

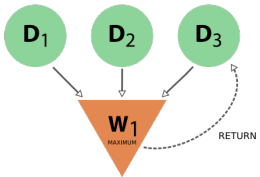
Since workflows can return, they can also return their inputs

```
from aiida.engine import workfunction
from aiida.orm import Int

@workfunction
def maximum(x, y, z):
    return sorted([x, y, z])[-1]

result = maximum(Int(1), Int(2), Int(3))
```

WHY A DIFFERENCE BETWEEN CALCULATIONS AND WORKFLOWS?

Since workflows can return, they can also return their inputs

```python
from aiida.engine import workfunction
from aiida.orm import Int

@workfunction
def maximum(x, y, z):
    return sorted([x, y, z])[-1]

result = maximum(Int(1), Int(2), Int(3))
```

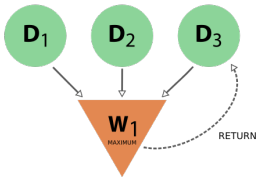


This cycle breaks the acyclicity → no more DAG

WHY A DIFFERENCE BETWEEN CALCULATIONS AND WORKFLOWS?

Since workflows can return, they can also return their inputs

```python
from aiida.engine import workfunction
from aiida.orm import Int

@workfunction
def maximum(x, y, z):
    return sorted([x, y, z])[-1]

result = maximum(Int(1), Int(2), Int(3))
```



This cycle breaks the acyclicity → no more DAG

Two clearly distinct types of processes

CALCULATIONS

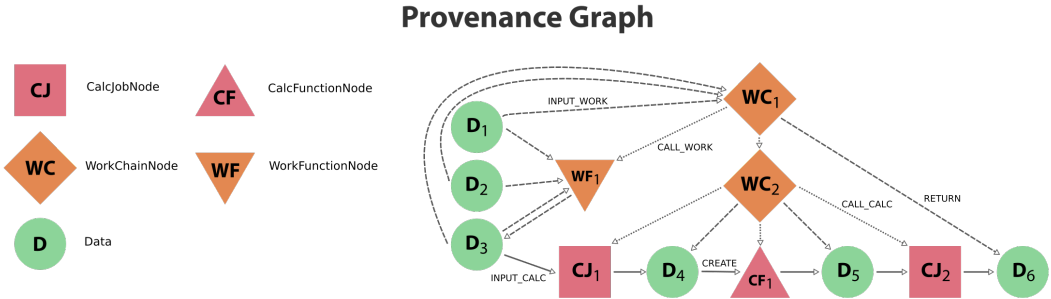
Can *create* new data

WORKFLOWS

Can *call* other processes

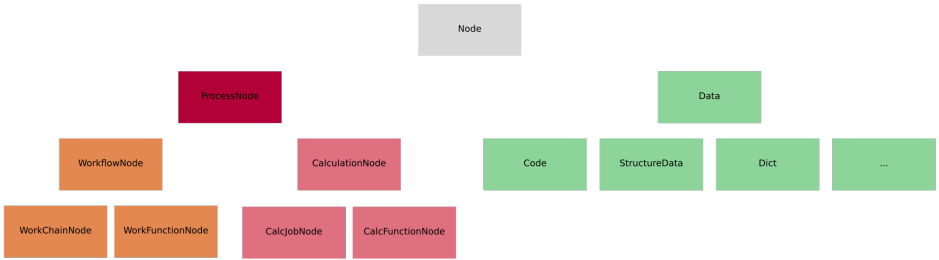
Can *return* existing data

PROVENANCE DESIGN: THE RULES



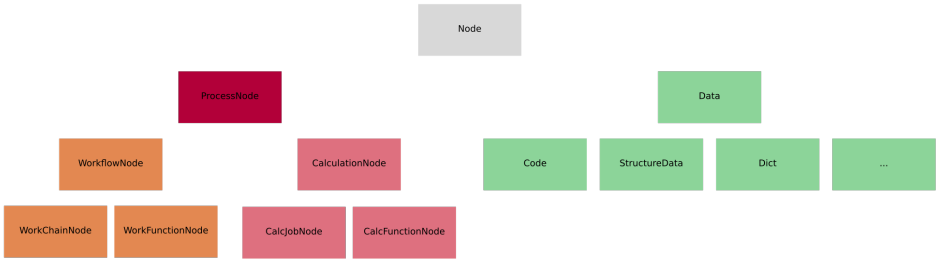
PROVENANCE DESIGN: THE RULES

ORM Hierarchy

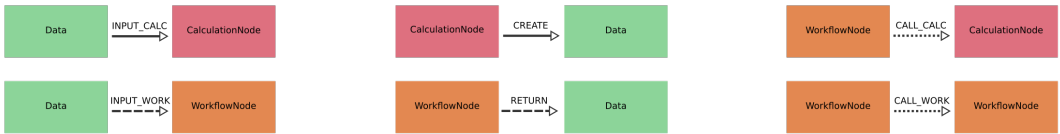


PROVENANCE DESIGN: THE RULES

ORM Hierarchy



Link Hierarchy



ENGINE: EVERYTHING IS A PROCESS

New processes to define calculations and workflows

Process class	Node class	Used for
`CalcJob`	`CalcJobNode`	Calculations performed by external codes
`WorkChain`	`WorkChainNode`	Workflows that run multiple sub processes
`FunctionProcess`	`CalcFunctionNode`	Python functions decorated with the `calcfunction` decorator
`FunctionProcess`	`WorkFunctionNode`	Python functions decorated with the `workfunction` decorator

ENGINE: EVERYTHING IS A PROCESS

New processes to define calculations and workflows

Process class	Node class	Used for
`CalcJob`	`CalcJobNode`	Calculations performed by external codes
`WorkChain`	`WorkChainNode`	Workflows that run multiple sub processes
`FunctionProcess`	`CalcFunctionNode`	Python functions decorated with the `calcfunction` decorator
`FunctionProcess`	`WorkFunctionNode`	Python functions decorated with the `workfunction` decorator

PROCESS STATE

Active	Terminated
Created	Killed
Running	Excepted
Waiting	Finished

ENGINE: EVERYTHING IS A PROCESS

New processes to define calculations and workflows

Process class	Node class	Used for
`CalcJob`	`CalcJobNode`	Calculations performed by external codes
`WorkChain`	`WorkChainNode`	Workflows that run multiple sub processes
`FunctionProcess`	`CalcFunctionNode`	Python functions decorated with the `calcfunction` decorator
`FunctionProcess`	`WorkFunctionNode`	Python functions decorated with the `workfunction` decorator

PROCESS STATE

Active	Terminated
Created	Killed
Running	Excepted
Waiting	Finished

PROCESS NODE ATTRIBUTES

Property	Meaning
`process_state`	Returns the current process state
`exit_status`	Returns the exit status, or None if not set
`exit_message`	Returns the exit message, or None if not set
`is_terminated`	Returns `True` if the process was either `Killed`, `Excepted` or `Finished`
`is_killed`	Returns `True` if the process is `Killed`
`is_excepted`	Returns `True` if the process is `Excepted`
`is_finished`	Returns `True` if the process is `Finished`
`is_finished_ok`	Returns `True` if the process is `Finished` and the `exit_status` is equal to zero
`is_failed`	Returns `True` if the process is `Finished` and the `exit_status` is non-zero

ENGINE: EVERYTHING IS A PROCESS

Four processes launchers with identical interface

`run`	Run blockingly and return result
`run_get_node`	Run blockingly and return result + node
`run_get_pk`	Run blockingly and return result + pk
`submit`	Submit to daemon and return node

ENGINE: EVERYTHING IS A PROCESS

Four processes launchers with identical interface

`run`	Run blockingly and return result
`run_get_node`	Run blockingly and return result + node
`run_get_pk`	Run blockingly and return result + pk
`submit`	Submit to daemon and return node

Submit to the daemon

```
from aiida import orm
from aiida.engine import submit

ArithmeticAddCalculation = CalculationFactory('arithmetic.add')
node = submit(ArithmeticAddCalculation, x=orm.Int(1), y=orm.Int(2))
```

ENGINE: EVERYTHING IS A PROCESS

Four processes launchers with identical interface

`run`	Run blockingly and return result
`run_get_node`	Run blockingly and return result + node
`run_get_pk`	Run blockingly and return result + pk
`submit`	Submit to daemon and return node

Run blockingly in local interpreter

```
from aiida import orm
from aiida.engine import run

ArithmeticAddCalculation = CalculationFactory('arithmetic.add')
result = run(ArithmeticAddCalculation, x=orm.Int(1), y=orm.Int(2))
```

ENGINE: EVERYTHING IS A PROCESS

Four processes launchers with identical interface

`run`	Run blockingly and return result
`run_get_node`	Run blockingly and return result + node
`run_get_pk`	Run blockingly and return result + pk
`submit`	Submit to daemon and return node

Run variants to get the process node or pk in addition to the result

```
from aiida import orm
from aiida.engine import run_get_node, run_get_pk

ArithmeticAddCalculation = CalculationFactory('arithmetic.add')
result, node = run_get_node(ArithmeticAddCalculation, x=orm.Int(1), y=orm.Int(2))
result, pk = run_get_pk(ArithmeticAddCalculation, x=orm.Int(1), y=orm.Int(2))
```

ENGINE: EVERYTHING IS A PROCESS

Four processes launchers with identical interface

`run`	Run blockingly and return result
`run_get_node`	Run blockingly and return result + node
`run_get_pk`	Run blockingly and return result + pk
`submit`	Submit to daemon and return node

Variants are available as attributes on `run` launcher requiring only single import

```
from aiida import orm
from aiida.engine import run

ArithmeticAddCalculation = CalculationFactory('arithmetic.add')
result = run(ArithmeticAddCalculation, x=orm.Int(1), y=orm.Int(2))
result, node = run.get_node(ArithmeticAddCalculation, x=orm.Int(1), y=orm.Int(2))
result, pk = run.get_pk(ArithmeticAddCalculation, x=orm.Int(1), y=orm.Int(2))
```

ENGINE: EVERYTHING IS A PROCESS

Four processes launchers with identical interface

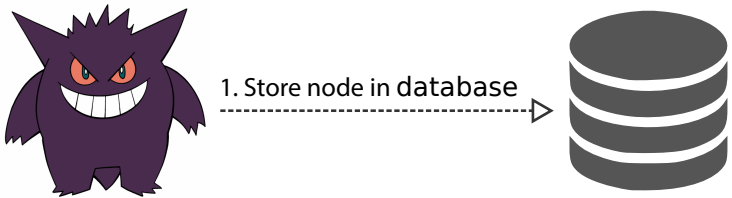
`run`	Run blockingly and return result
`run_get_node`	Run blockingly and return result + node
`run_get_pk`	Run blockingly and return result + pk
`submit`	Submit to daemon and return node

Syntactic keyword expansion for big input dictionaries

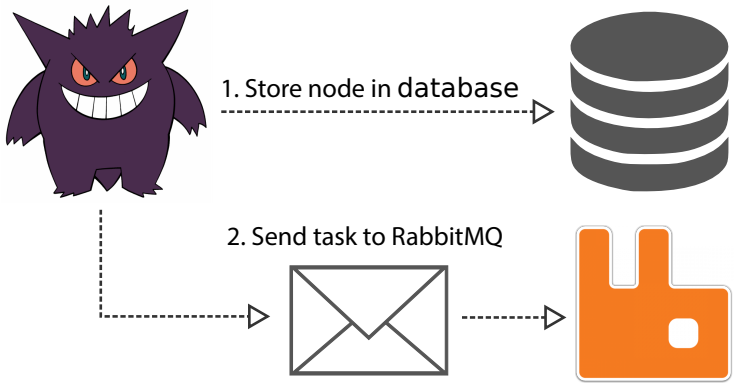
```
from aiida import orm
from aiida.engine import submit

ArithmeticAddCalculation = CalculationFactory('arithmetic.add')
inputs = {
    'x': orm.Int(1),
    'y': orm.Int(2)
}
node = submit(ArithmeticAddCalculation, **inputs)
```

What happens when we submit a process?

What happens when we submit a process?

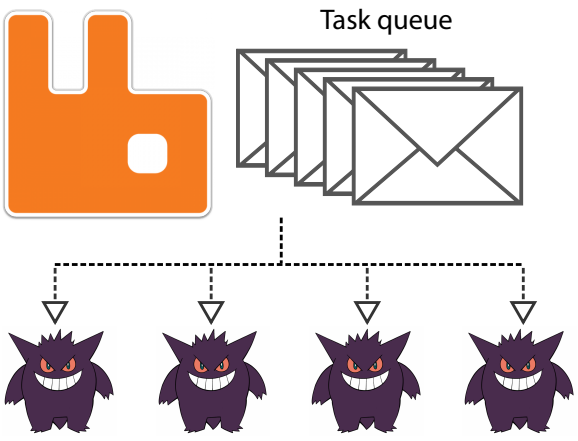


What happens when we submit a process?



What happens with those tasks?

What happens with those tasks?



ENGINE: RELYING ON A RESILIENT AND ROBUST RABBIT

The promise of RabbitMQ

- All tasks are persisted to disk
- Each task is guaranteed to be delivered
- Each task is guaranteed to be sent to only one listener at a time
- Each task is guaranteed to be completed

ENGINE: RELYING ON A RESILIENT AND ROBUST RABBIT

The promise of RabbitMQ

- All tasks are persisted to disk
- Each task is guaranteed to be delivered
- Each task is guaranteed to be sent to only one listener at a time
- Each task is guaranteed to be completed

This ensures that:

1. We can run multiple processes in parallel independently

2. Each launched task or "process" will eventually be completed

No matter what happens

ENGINE: EVERYTHING IS A PROCESS

`verdi process`: your one-stop-shop for inspecting and interacting with processes

ENGINE: EVERYTHING IS A PROCESS

`verdi process`: your one-stop-shop for inspecting and interacting with processes

`verdi process list`: list active and terminated processes

```
(aiida_3dd) aiida@theossrv5:~/code/aiida/env/3dd$ verdi process list -l 10
     PK  Created    State      Process label     Process status
-------  ---------  ---------  ---------------  --------------------------------
1285287  2D ago     ⏵ Waiting  PwRelaxWorkChain
1285535  2D ago     ⏵ Waiting  PwRelaxWorkChain
1286517  2D ago     ⏵ Waiting  PwRelaxWorkChain
1286913  2D ago     ⏵ Waiting  PwRelaxWorkChain
1287288  2D ago     ⏵ Waiting  PwBaseWorkChain
1287486  2D ago     ⏵ Waiting  PwRelaxWorkChain
1287517  2D ago     ⏵ Waiting  PwBaseWorkChain
1287937  2D ago     ⏵ Waiting  PwRelaxWorkChain
1289927  2D ago     ⏵ Waiting  PwCalculation     Waiting for transport task: update
1291148  2D ago     ⏵ Waiting  PwBaseWorkChain

Total results: 10

Info: last time an entry changed state: 43s ago (at 11:09:50 on 2019-03-22)
```

ENGINE: EVERYTHING IS A PROCESS

`verdi process`: your one-stop-shop for inspecting and interacting with processes

`verdi process status`: tree representation of call stack

```
(aiida_3dd) aiida@theossrv5:~/code/aiida/env/3dd$ verdi process status 1338418
PwRelaxWorkChain <pk=1338418> [ProcessState.FINISHED] [3:results]
    ├── PwBaseWorkChain <pk=1338525> [ProcessState.FINISHED] [4:results]
    │   ├── create_kpoints_from_distance <pk=1338529> [ProcessState.FINISHED]
    │   └── CalcJobNode <pk=1338569> [FINISHED]
    ├── PwBaseWorkChain <pk=1341443> [ProcessState.FINISHED] [4:results]
    │   ├── create_kpoints_from_distance <pk=1341445> [ProcessState.FINISHED]
    │   └── CalcJobNode <pk=1341463> [FINISHED]
    └── PwBaseWorkChain <pk=1341701> [ProcessState.FINISHED] [4:results]
        ├── create_kpoints_from_distance <pk=1341703> [ProcessState.FINISHED]
        └── CalcJobNode <pk=1341730> [FINISHED]
```

ENGINE: EVERYTHING IS A PROCESS

`verdi process`: your one-stop-shop for inspecting and interacting with processes

`verdi process report`: complete report of log messages and scheduler stdout/stderr

```
(aiida_3dd) aiida@theossrv5:~/code/aiida/env/3dd$ verdi process report 1338418
2019-03-22 05:26:39 [553130 | REPORT]: [1338418|PwRelaxWorkChain|run_relax]: launching PwBaseWorkChain<1338525>
2019-03-22 05:28:02 [553147 | REPORT]:   [1338525|PwBaseWorkChain|run_calculation]: launching PwCalculation<1338569> iteration #1
2019-03-22 07:45:52 [554706 | REPORT]:   [1338525|PwBaseWorkChain|inspect_calculation]: PwCalculation<1338569> completed successfully
2019-03-22 07:45:52 [554707 | REPORT]:   [1338525|PwBaseWorkChain|results]: workchain completed after 1 iterations
2019-03-22 07:45:52 [554708 | REPORT]:   [1338525|PwBaseWorkChain|on_terminated]: remote folders will not be cleaned
2019-03-22 07:45:53 [554709 | REPORT]: [1338418|PwRelaxWorkChain|inspect_relax]: after iteration 1 cell volume of relaxed structure is 427.27585978
2019-03-22 07:45:54 [554710 | REPORT]: [1338418|PwRelaxWorkChain|run_relax]: launching PwBaseWorkChain<1341443>
2019-03-22 07:47:21 [554713 | REPORT]:   [1341443|PwBaseWorkChain|run_calculation]: launching PwCalculation<1341463> iteration #1
2019-03-22 08:00:47 [554854 | REPORT]:   [1341443|PwBaseWorkChain|inspect_calculation]: PwCalculation<1341463> completed successfully
2019-03-22 08:00:47 [554855 | REPORT]:   [1341443|PwBaseWorkChain|results]: workchain completed after 1 iterations
2019-03-22 08:00:47 [554856 | REPORT]:   [1341443|PwBaseWorkChain|on_terminated]: remote folders will not be cleaned
2019-03-22 08:00:48 [554857 | REPORT]: [1338418|PwRelaxWorkChain|inspect_relax]: after iteration 2 cell volume of relaxed structure is 427.275859777
2019-03-22 08:00:48 [554858 | REPORT]: [1338418|PwRelaxWorkChain|inspect_relax]: relative cell volume difference 5.54830030607e-12 smaller than convergence threshold 0.01
2019-03-22 08:00:49 [554859 | REPORT]: [1338418|PwRelaxWorkChain|run_final_scf]: launching PwBaseWorkChain<1341701> for final scf
2019-03-22 08:02:13 [554881 | REPORT]:   [1341701|PwBaseWorkChain|run_calculation]: launching PwCalculation<1341730> iteration #1
2019-03-22 08:11:19 [554931 | REPORT]:   [1341701|PwBaseWorkChain|inspect_calculation]: PwCalculation<1341730> completed successfully
2019-03-22 08:11:19 [554932 | REPORT]:   [1341701|PwBaseWorkChain|results]: workchain completed after 1 iterations
2019-03-22 08:11:19 [554933 | REPORT]:   [1341701|PwBaseWorkChain|on_terminated]: remote folders will not be cleaned
2019-03-22 08:11:20 [554934 | REPORT]: [1338418|PwRelaxWorkChain|results]: workchain completed after 2 iterations
2019-03-22 08:11:30 [554935 | REPORT]: [1338418|PwRelaxWorkChain|on_terminated]: cleaned remote folders of calculations: 1341730 1341463 1338569
```

ENGINE: EVERYTHING IS A PROCESS

`verdi process`: your one-stop-shop for inspecting and interacting with processes

`verdi process pause`: pause an active process

```
(aiida_dev) sphuber@theos:~$ verdi process pause 804
Success: scheduled pause Process<804>
```

ENGINE: EVERYTHING IS A PROCESS

`verdi process`: your one-stop-shop for inspecting and interacting with processes

`verdi process pause`: pause an active process

```
(aiida_dev) sphuber@theos:~$ verdi process pause 804
Success: scheduled pause Process<804>
```

`verdi process play`: resume a paused process

```
(aiida_dev) sphuber@theos:~$ verdi process play 812
Success: scheduled play Process<812>
```

ENGINE: EVERYTHING IS A PROCESS

`verdi process`: your one-stop-shop for inspecting and interacting with processes

`verdi process pause`: pause an active process

```
(aiida_dev) sphuber@theos:~$ verdi process pause 804
Success: scheduled pause Process<804>
```

`verdi process play`: resume a paused process

```
(aiida_dev) sphuber@theos:~$ verdi process play 812
Success: scheduled play Process<812>
```

`verdi process kill`: kill an active process

```
(aiida_dev) sphuber@theos:~$ verdi process kill 820
Success: scheduled kill Process<820>
```

ENGINE: EVERYTHING IS A PROCESS

`verdi process`: your one-stop-shop for inspecting and interacting with processes

`verdi process pause`: pause an active process

```
(aiida_dev) sphuber@theos:~$ verdi process pause 804
Success: scheduled pause Process<804>
```

`verdi process play`: resume a paused process

```
(aiida_dev) sphuber@theos:~$ verdi process play 812
Success: scheduled play Process<812>
```

`verdi process kill`: kill an active process

```
(aiida_dev) sphuber@theos:~$ verdi process kill 820
Success: scheduled kill Process<820>
```

`verdi process pause/play/kill`: fails if process is already terminated

```
(aiida_dev) sphuber@theos:~$ verdi process kill 812
Error: Process<812> is already terminated
```

ENGINE: ROBUSTNESS

Automatic retry for transport tasks with exponential backoff

```
(aiida_3dd) aiida@theossrv5:~/code/aiida/env/3dd$ verdi process list --paused
     PK  Created    State      Process label    Process status
-------  ---------  ---------  ---------------  ------------------------------------------------------------------------------------------
1343914  5h ago     ⏸ Waiting  PwCalculation    Pausing after failed transport task: retrieve_calculation failed 5 times consecutively

Total results: 1

Info: last time an entry changed state: 21s ago (at 16:10:08 on 2019-03-22)
```

ENGINE: ROBUSTNESS

Automatic retry for transport tasks with exponential backoff

```
(aiida_3dd) aiida@theossrv5:~/code/aiida/env/3dd$ verdi process list --paused
     PK  Created    State      Process label    Process status
-------  ---------  ---------  ---------------  ------------------------------------------------------------------------------------------
1343914  5h ago     ⏸ Waiting  PwCalculation    Pausing after failed transport task: retrieve_calculation failed 5 times consecutively

Total results: 1

Info: last time an entry changed state: 21s ago (at 16:10:08 on 2019-03-22)
```

Rate limited connections to remote clusters per daemon worker

```
(aiida_3dd) aiida@theossrv5:~$ verdi computer configure show localhost
* safe_interval  0
```

ENGINE: ROBUSTNESS

Automatic retry for transport tasks with exponential backoff

```
(aiida_3dd) aiida@theossrv5:~/code/aiida/env/3dd$ verdi process list --paused
     PK  Created    State      Process label    Process status
-------  ---------  ---------  ---------------  ------------------------------------------------------------------------------------------
1343914  5h ago     ⏸ Waiting  PwCalculation    Pausing after failed transport task: retrieve_calculation failed 5 times consecutively

Total results: 1

Info: last time an entry changed state: 21s ago (at 16:10:08 on 2019-03-22)
```

Rate limited connections to remote clusters per daemon worker

```
(aiida_3dd) aiida@theossrv5:~$ verdi computer configure show localhost
* safe_interval  0
```

Rate limited scheduler state queries per daemon worker

```
(aiida_3dd) aiida@theossrv5:~$ verdi shell

In [1]: from aiida.orm import Computer

In [2]: computer = Computer.get(name='localhost')

In [3]: computer.get_minimum_job_poll_interval()
Out[3]: 30

In [4]: computer.set_minimum_job_poll_interval(60)

In [5]: computer.get_minimum_job_poll_interval()
Out[5]: 60
```

ACKNOWLEDGMENTS

Casper
Andersen
(EPFL)

Marco
Borelli
(EPFL)

Sebastiaan
Huber
(EPFL)

Leonid
Kahle
(EPFL)

Nicola
Marzari
(EPFL)

Martin
Muhrin
(EPFL)

Elsa
Passaro
(EPFL)

Giovanni
Pizzi
(EPFL)

Aliaksandr
Yakutovich
(EPFL)

Snehal
Waychal
(EPFL)

Spyros
Zoupanos
(EPFL)

Martin Uhrin, Rico Häuselmann, Nicolas Mounet, Andrea Cepellotti, Fernando Gargiulo, Riccardo Sabatini, Rico Häuselmann, Valentin Bersier, Jocelyn Boullier, Jens Bröder, Marco Dorigo, Marco Gibertini, Dominik Gresch Eric Hontz, Daniel Marchand , Tiziano Müller, Phillippe Schwaller, Ivano E. Castelli, Ian Lee, Gianluca Prandini, Jianxing Huang, Antimo Marrazzo, Nicola Varini, Mario Zic, Vladimir Dikan, Michael Atambo, Ole Schütt, Y.-W. Fang, Philipp Rüßmann, Bonan Zhu, Andreas Stamminger, Keija Cui, Daniel Hollas, Jianxing Huang, Espen Flage-Larsen

FIN