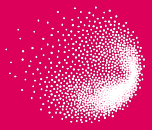


PSI Center for Scientific Computing, Theory and Data

How to write your workflow

Lessons I learned writing the `koopmans` package

Edward Linscott

LMS Seminar, 12 March 2025

A brief history of the `koopmans` code

A brief history of the `koopmans` code

- I arrived in THEOS in 2019, started learning how to run Koopmans functionals – and it was painful!

A brief history of the `koopmans` code

- I arrived in THEOS in 2019, started learning how to run Koopmans functionals – and it was painful!
- I was familiar with `ASE` so wrote some simple scripts

A brief history of the `koopmans` code

- I arrived in THEOS in 2019, started learning how to run Koopmans functionals – and it was painful!
- I was familiar with `ASE` so wrote some simple scripts
- decided this was something we wanted for everyone

A brief history of the `koopmans` code

- I arrived in THEOS in 2019, started learning how to run Koopmans functionals – and it was painful!
- I was familiar with `ASE` so wrote some simple scripts
- decided this was something we wanted for everyone
- … and having been advised not to use `AiiDA` we stuck with the `ASE` approach

A brief history of the `koopmans` code

- I arrived in THEOS in 2019, started learning how to run Koopmans functionals – and it was painful!
- I was familiar with `ASE` so wrote some simple scripts
- decided this was something we wanted for everyone
- … and having been advised not to use `AiiDA` we stuck with the `ASE` approach
- implemented `kcp.x` support in `ASE`

A brief history of the `koopmans` code

- I arrived in THEOS in 2019, started learning how to run Koopmans functionals – and it was painful!
- I was familiar with `ASE` so wrote some simple scripts
- decided this was something we wanted for everyone
- … and having been advised not to use `AiiDA` we stuck with the `ASE` approach
- implemented `kcp.x` support in `ASE`
- the first workflow was Koopmans ΔSCF for molecules; others quickly followed

A brief history of the `koopmans` code

- I arrived in THEOS in 2019, started learning how to run Koopmans functionals – and it was painful!
- I was familiar with `ASE` so wrote some simple scripts
- decided this was something we wanted for everyone
- … and having been advised not to use `AiiDA` we stuck with the `ASE` approach
- implemented `kcp.x` support in `ASE`
- the first workflow was Koopmans ΔSCF for molecules; others quickly followed
- reusability of workflows as subworkflows became important (e.g. Wannierisation)

A brief history of the `koopmans` code

- I arrived in THEOS in 2019, started learning how to run Koopmans functionals – and it was painful!
- I was familiar with `ASE` so wrote some simple scripts
- decided this was something we wanted for everyone
- … and having been advised not to use `AiiDA` we stuck with the `ASE` approach
- implemented `kcp.x` support in `ASE`
- the first workflow was Koopmans ΔSCF for molecules; others quickly followed
- reusability of workflows as subworkflows became important (e.g. Wannierisation)
- start to add tests, documentation, etc, …

A brief history of the `koopmans` code

- I arrived in THEOS in 2019, started learning how to run Koopmans functionals – and it was painful!
- I was familiar with `ASE` so wrote some simple scripts
- decided this was something we wanted for everyone
- … and having been advised not to use `AiiDA` we stuck with the `ASE` approach
- implemented `kcp.x` support in `ASE`
- the first workflow was Koopmans ΔSCF for molecules; others quickly followed
- reusability of workflows as subworkflows became important (e.g. Wannierisation)
- start to add tests, documentation, etc, …

koopmans

A brief history of the `koopmans` code

- I arrived in THEOS in 2019, started learning how to run Koopmans functionals – and it was painful!
- I was familiar with `ASE` so wrote some simple scripts
- decided this was something we wanted for everyone
- … and having been advised not to use `AiiDA` we stuck with the `ASE` approach
- implemented `kcp.x` support in `ASE`
- the first workflow was Koopmans ΔSCF for molecules; others quickly followed
- reusability of workflows as subworkflows became important (e.g. Wannierisation)
- start to add tests, documentation, etc, …

koopmans

- (more recently) substantial changes to integrate with `AiiDA` (I'll discuss this later)

… but what have I learned throughout this process?


... not quite!

2019


TITRE DE SÉJOUR
AUFENTHALTSTITEL
PERMESSO DI SOGGIORNO

2024


CHE
AUFENTHALTSTITEL
TITRE DE SÉJOUR / PERMESSO DI SOGGIORNO

What I learned when interfacing `koopmans` with `AiiDA`

koopmans

Simple by design

koopmans

Simple by design

- local execution only

koopmans

Simple by design

- local execution only
- serial step execution (even when steps are independent!)

koopmans

Simple by design

- local execution only
- serial step execution (even when steps are independent!)
- direct access to input/output files

koopmans

Simple by design

- local execution only
- serial step execution (even when steps are independent!)
- direct access to input/output files
- simple installation

koopmans

Simple by design

- local execution only
- serial step execution (even when steps are independent!)
- direct access to input/output files
- simple installation

Powerful by design

koopmans

Simple by design

- local execution only
- serial step execution (even when steps are independent!)
- direct access to input/output files
- simple installation

Powerful by design

- remote execution

koopmans

Simple by design

- local execution only
- serial step execution (even when steps are independent!)
- direct access to input/output files
- simple installation

AiiDA

Powerful by design

- remote execution
- parallel step execution

koopmans

Simple by design

- local execution only
- serial step execution (even when steps are independent!)
- direct access to input/output files
- simple installation

AiiDA

Powerful by design

- remote execution
- parallel step execution
- outputs stored in a database

koopmans

Simple by design

- local execution only
- serial step execution (even when steps are independent!)
- direct access to input/output files
- simple installation

AiiDA

Powerful by design

- remote execution
- parallel step execution
- outputs stored in a database
- installation more involved

koopmans

Simple by design

- local execution only
- serial step execution (even when steps are independent!)
- direct access to input/output files
- simple installation

AiiDA

Powerful by design

- remote execution
- parallel step execution
- outputs stored in a database
- installation more involved

We could really benefit from a lot of these features

Goals

- be able to execute in parallel and remotely with `AiiDA`

Goals

- be able to execute in parallel and remotely with `AiiDA`
- UI should be as similar as possible

Goals

- be able to execute in parallel and remotely with `AiiDA`
- UI should be as similar as possible
- old mode of running `koopmans` should still work

Goals

- be able to execute in parallel and remotely with `AiiDA`
- UI should be as similar as possible
- old mode of running `koopmans` should still work
- minimal/no duplication of logic

Refactoring

A lot of “bad” design in `koopmans` needed refactoring

Refactoring

A lot of “bad” design in `koopmans` needed refactoring

- abstraction of many operations e.g.

```
with open(filename, 'r') as f:
        data = f.read()
```

Refactoring

A lot of “bad” design in `koopmans` needed refactoring

- abstraction of many operations e.g.

```
with open(filename, 'r') as f:
        data = f.read()
```

becomes

```
self.engine.read(filename)
```

Refactoring

A lot of “bad” design in `koopmans` needed refactoring

- abstraction of many operations e.g.

```
with open(filename, 'r') as f:
        data = f.read()
```

becomes

```
self.engine.read(filename)
```

- … and, more generally, many responsibilities are moved to the `engine` (reading/writing files, running calculations, checking the status of calculations, loading pseudopotentials, etc.)

Refactoring

- making everything more “pure”

Refactoring

- making everything more “pure” e.g. removing all reliance on shared directories

Refactoring

- making everything more “pure” e.g. removing all reliance on shared directories

```
calc_nscf.parameters.outdir = calc_scf.parameters.outdir
```

Refactoring

- making everything more “pure” e.g. removing all reliance on shared directories

```
calc_nscf.parameters.outdir = calc_scf.parameters.outdir
```

becomes

```
calc_nscf.link(calc_scf.parameters.outdir, 'tmp')
```

where

```
def link(self, src, dst):
      self.engine.link(src=src, dst=self/dst)
```

***Note*: none of this refactoring is specific to `AiiDA`!**

At the `AiiDA` end

At the AiiDA end

- conversion of calculators from ASE to AiiDA and back

At the AiiDA end

- conversion of calculators from ASE to AiiDA and back
- verdi presto for simplified AiiDA setup

At the AiiDA end

- conversion of calculators from ASE to AiiDA and back
- verdi presto for simplified AiiDA setup
- verdi dump for dumping AiiDA database to a local file structure

Writing workflows well is hard…

Writing workflows well is hard… how can it best be done?

Common Workflow Language

Basic Concepts of CWL

Basic Concepts of CWL

- “an open standard for describing how to run command line tools and connect them to create workflows”

Basic Concepts of CWL

- “an open standard for describing how to run command line tools and connect them to create workflows”
- separate `runners` are required to execute the workflows; a CWL workflow only contains the logic of the workflows

Basic Concepts of CWL

- “an open standard for describing how to run command line tools and connect them to create workflows”
- separate `runners` are required to execute the workflows; a CWL workflow only contains the logic of the workflows
- introduced in 2014; version 1.2 released in 2020

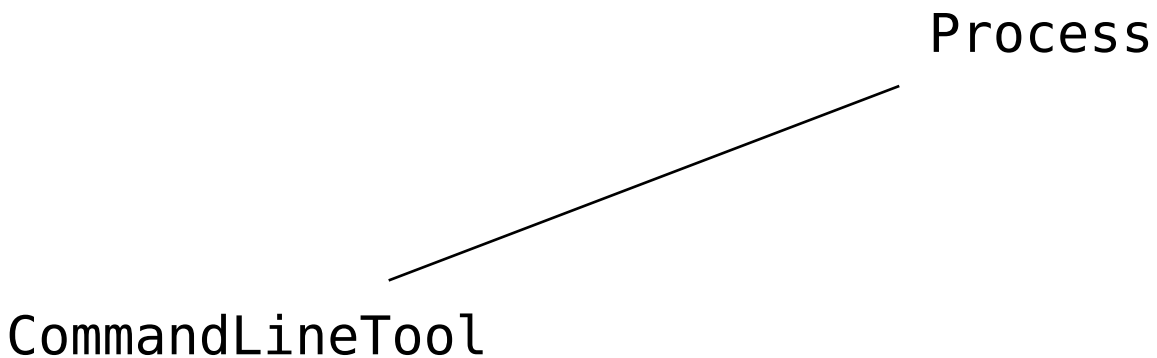
Basic Concepts of CWL

- “an open standard for describing how to run command line tools and connect them to create workflows”
- separate `runners` are required to execute the workflows; a CWL workflow only contains the logic of the workflows
- introduced in 2014; version 1.2 released in 2020
- mostly used by bioinformatics community

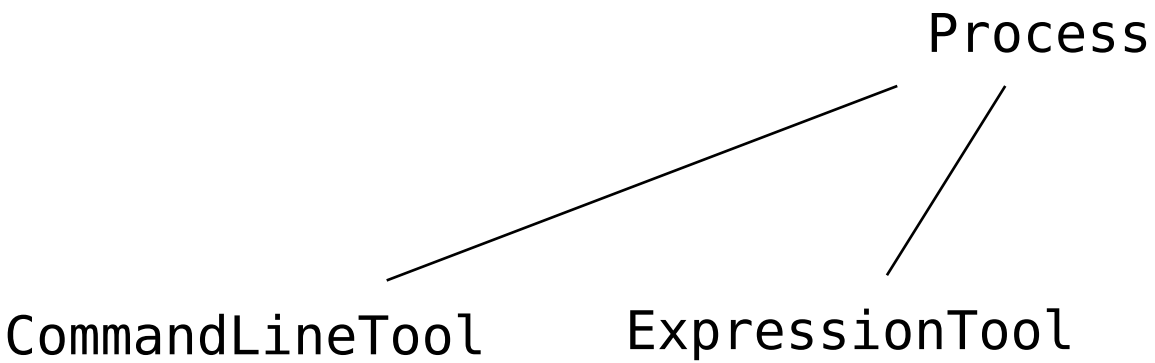
Basic Concepts of CWL

`Process`

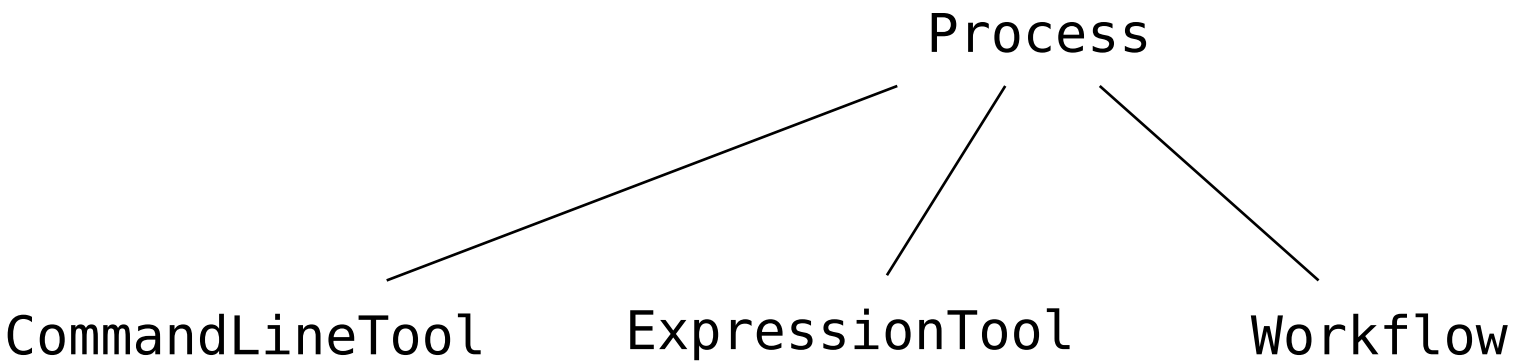
Basic Concepts of CWL



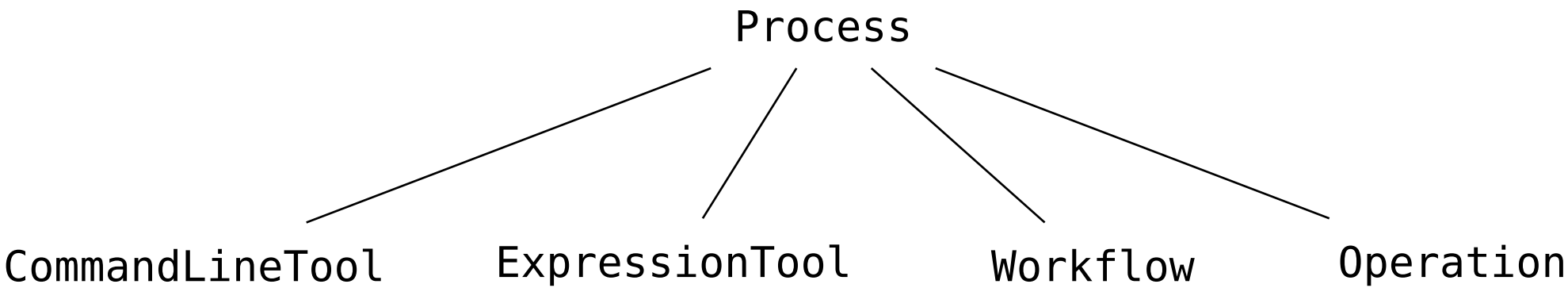
Basic Concepts of CWL



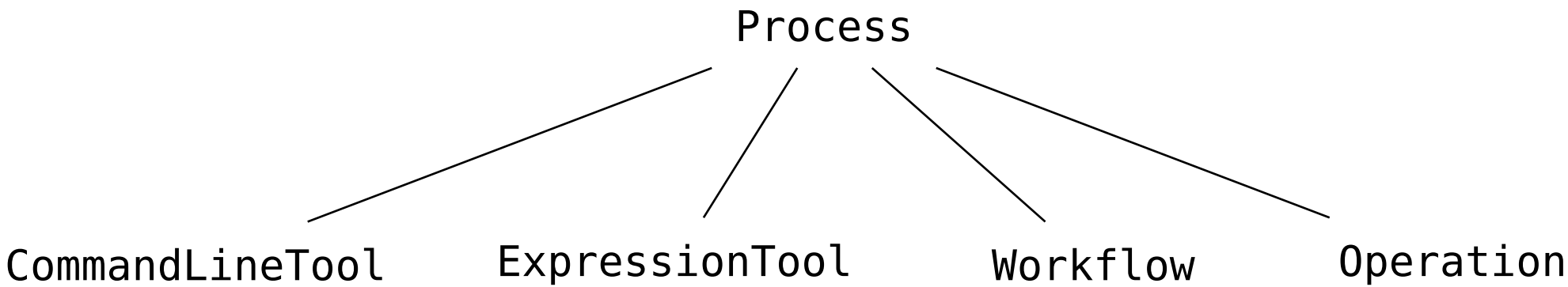
Basic Concepts of CWL



Basic Concepts of CWL



Basic Concepts of CWL



Every `Process` has `inputs` and `outputs`

CommandLineTool

PSI

echo.cwl

```yaml
cwlVersion: v1.2
class: CommandLineTool

inputs:
  message:
    type: string
    default: "Hello World"
    inputBinding:
      position: 1
outputs: []

baseCommand: echo
```

CommandLineTool

PSI

echo.cwl

```yaml
cwlVersion: v1.2
class: CommandLineTool

inputs:
  message:
    type: string
    default: "Hello World"
    inputBinding:
      position: 1
outputs: []

baseCommand: echo
```

Called via `$ cwltool echo.cwl`

ExpressionTool

uppercase.cwl

```yaml
cwlVersion: v1.2
class: ExpressionTool
requirements:
  InlineJavascriptRequirement: {}

inputs:
  message: string
outputs:
  uppercase_message: string

expression: |
  ${ return {"uppercase_message": inputs.message.toUpperCase()}; }
```

ExpressionTool

uppercase.cwl

```yaml
cwlVersion: v1.2
class: ExpressionTool
requirements:
  InlineJavascriptRequirement: {}

inputs:
  message: string
outputs:
  uppercase_message: string

expression: |
  ${ return {"uppercase_message": inputs.message.toUpperCase()}; }
```

Called via `$ cwltool uppercase.cwl --message='goes to 11'`

Workflow

PSI

echo_uppercase.cwl

```yaml
cwlVersion: v1.2
class: Workflow

requirements:
  InlineJavascriptRequirement: {}

inputs:
  message: string
outputs:
  out:
    type: string
    outputSource: uppercase/uppercase_message
```

Workflow

```yaml
steps:
  echo:
    run: echo.cwl
    in:
      message: message
    out: [out]
  uppercase:
    run: uppercase.cwl
    in:
      message:
        source: echo/out
    out: [uppercase_message]
```

Workflow

PSI

echo_uppercase.cwl

```yaml
cwlVersion: v1.2
class: Workflow

requirements:
  InlineJavascriptRequirement: {}

inputs:
  message: string
outputs:
  out:
    type: string
    outputSource: uppercase/uppercase_message
```

Workflow

```yaml
steps:
  echo:
    run: echo.cwl
    in:
      message: message
    out: [out]
  uppercase:
    run: uppercase.cwl
    in:
      message:
        source: echo/out
    out: [uppercase_message]
```

Called via `$ cwltool echo_uppercase.cwl input.json`

Operation

p_versus_np.cwl

```yaml
cwlVersion: v1.2
class: Operation

inputs: []
outputs:
    result: bool
```

A less silly `Operation`

PSI

run_pw.cwl

```yaml
cwlVersion: v1.2
class: Operation

inputs:
   input_file: File
   pseudopotentials:
       type: array
       items:
          type: File
outputs:
   output_file: File
```

Pros and Cons

- pros:

Pros and Cons

- pros:
  - explicit

Pros and Cons

- pros:
  - explicit
  - composable

Pros and Cons

- pros:
  - explicit
  - composable
  - customisable

Pros and Cons

- pros:
  - explicit
  - composable
  - customisable
- cons:

Pros and Cons

- pros:
  - explicit
  - composable
  - customisable
- cons:
  - verbose; lots of boilerplate

Pros and Cons

- pros:
  - explicit
  - composable
  - customisable
- cons:
  - verbose; lots of boilerplate
  - complicated workflows are **very** complicated to write in CWL (e.g. `while`)

Pros and Cons

- pros:
  - explicit
  - composable
  - customisable
- cons:
  - verbose; lots of boilerplate
  - complicated workflows are **very** complicated to write in CWL (e.g. `while`)
  - `ExpressionTool` restricted to Javascript

Pros and Cons

- pros:
  - explicit
  - composable
  - customisable
- cons:
  - verbose; lots of boilerplate
  - complicated workflows are **very** complicated to write in CWL (e.g. `while`)
  - `ExpressionTool` restricted to Javascript
  - need to define custom types (see e.g. OPTIMADE, PREMISE)

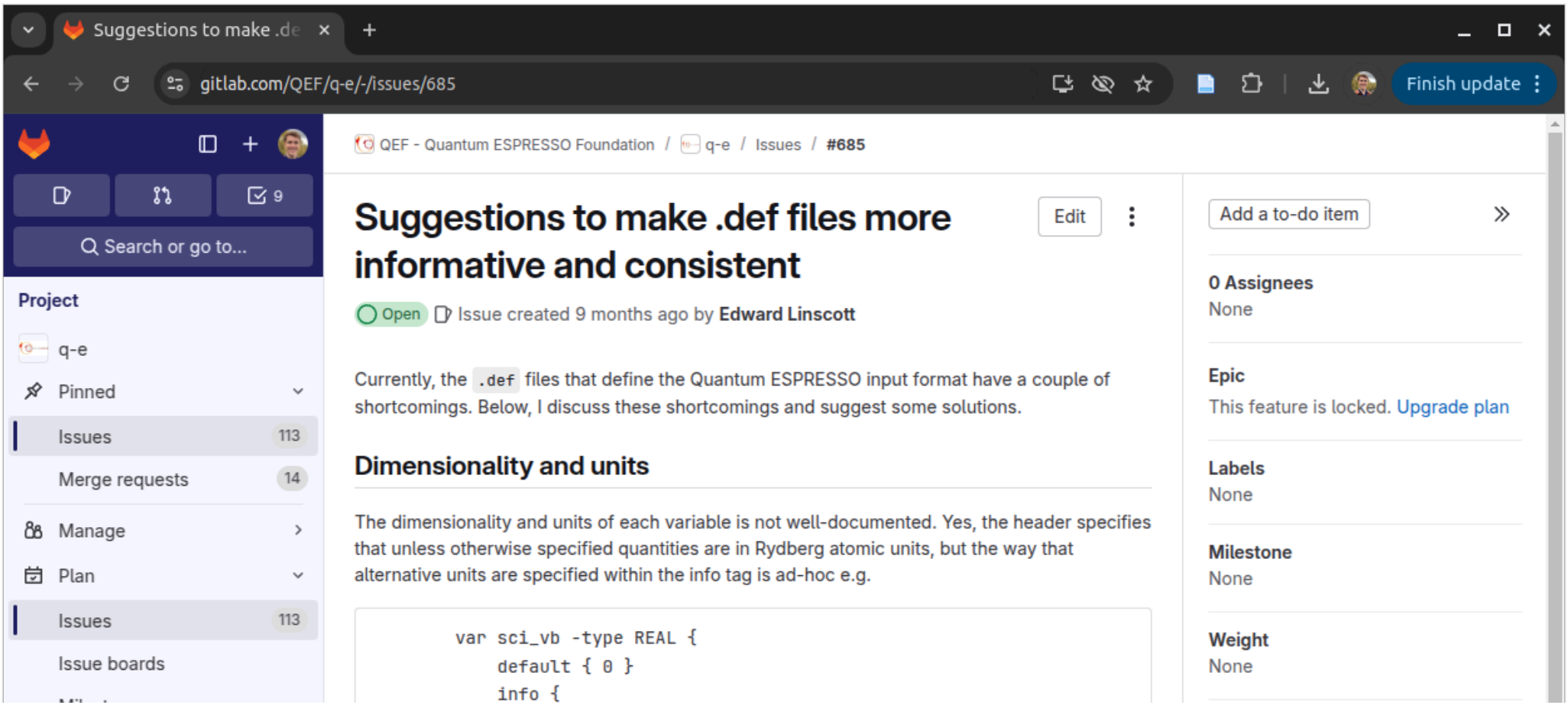
Pros and Cons

- pros:
  - explicit
  - composable
  - customisable
- cons:
  - verbose; lots of boilerplate
  - complicated workflows are **very** complicated to write in CWL (e.g. `while`)
  - `ExpressionTool` restricted to Javascript
  - need to define custom types (see e.g. OPTIMADE, PREMISE)
  - custom types do not permit defaults

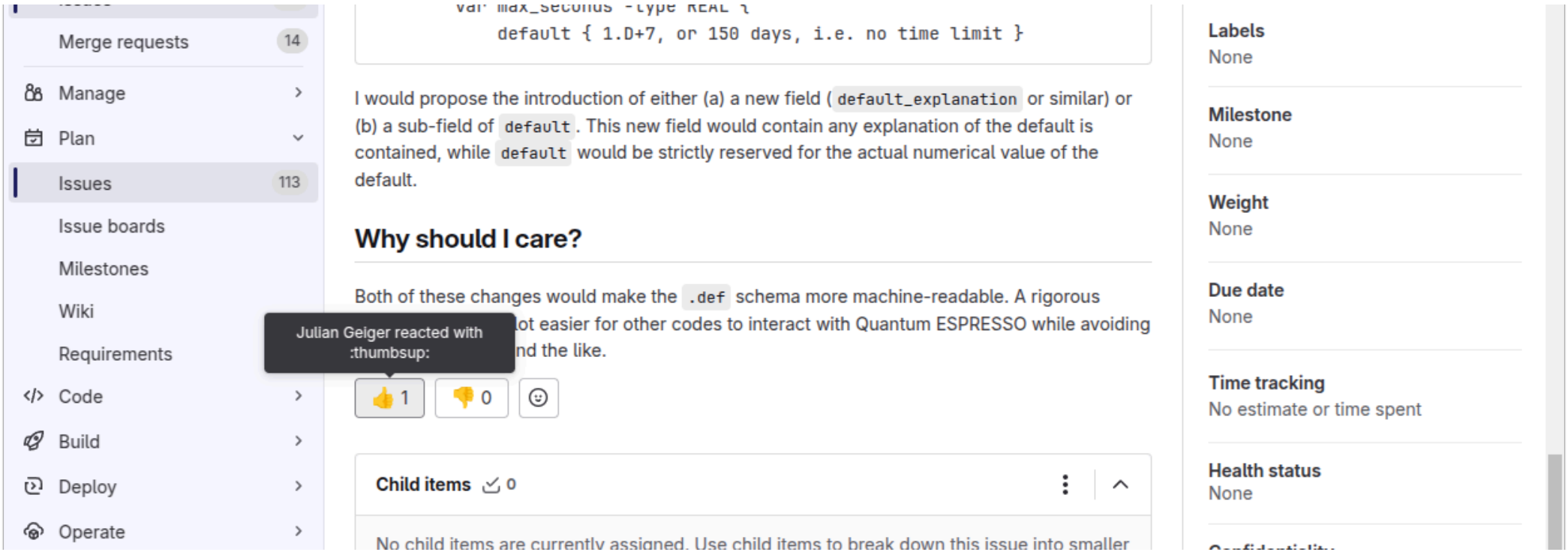
Pros and Cons

- pros:
  - explicit
  - composable
  - customisable
- cons:
  - verbose; lots of boilerplate
  - complicated workflows are **very** complicated to write in CWL (e.g. `while`)
  - `ExpressionTool` restricted to Javascript
  - need to define custom types (see e.g. OPTIMADE, PREMISE)
  - custom types do not permit defaults
  - rigorous schemas require willingness from the community

… willingness from the community?



… willingness from the community?



Building ideas from CWL into `koopmans`

Building ideas from CWL into `koopmans`

PSI

`koopmans` is slowly being refactored into “CWL-inspired” Python

```python
InputModel = TypeVar('InputModel', bound=BaseModel)
OutputModel = TypeVar('OutputModel', bound=BaseModel)

class Process(ABC, Generic[InputModel, OutputModel]):

    input_model: Type[InputModel]
    output_model: Type[OutputModel]

    def __init__(self, name: str | None = None, **kwargs):
        self.inputs: InputModel = self.input_model(**kwargs)
        self.outputs: OutputModel | None = None
        self.directory: Path | None = None

    def run(self):
        assert self.directory is not None, 'Process directory must be set before running'
        with utils.chdir(self.directory):
            self.dump_inputs()
            self._run()
            assert self.outputs is not None, 'Process outputs must be set when running'
            self.dump_outputs()

    @abstractmethod
    def _run(self):
        ...
```

A simple `CommandLineTool`

```python
class Bin2XMLInput(IOModel):
    binary: File
    model_config = ConfigDict(arbitrary_types_allowed=True)

class Bin2XMLOutput(IOModel):
    xml: File
    model_config = ConfigDict(arbitrary_types_allowed=True)

class Bin2XMLPCommandLineTool(CommandLineTool[Bin2XMLInput, Bin2XMLOutput]):

    input_model = Bin2XMLInput
    output_model = Bin2XMLOutput

    def _pre_run(self):
        super()._pre_run()
        if not self.inputs.binary.exists():
            raise FileNotFoundError(f'`{self.inputs.binary}` does not exist')

        # Link the input binary file to the directory of this process as input.dat
        dst = self / "input.dat"
        dst.symlink_to(self.inputs.binary)

    @property
    def command(self):
        return Command(executable='bin2xml.x', suffix='input.dat output.xml')

    def _set_outputs(self):
        self.outputs = self.output_model(xml=self / "output.xml")
```

An example composite workflow

```
# Read and run the Koopmans workflow
wf = read('si.json')
wf.run_while()

# Merge the separate occ + emp projections into a set of projections that combines occ + empty
combined_proj_list = [p for block in wf.projections for p in block.projections]
combined_proj_obj = ProjectionBlocks.fromlist([combined_proj_list], ['up'], wf.atoms)

# Find the kcw.x output directory that contains the Hamiltonian to Wannierize
[kcw_ham_calc] = [c for c in wf.calculations if isinstance(c, KoopmansHamCalculator)]
kcw_outdir = File(kcw_ham_calc, kcw_ham_calc.parameters.outdir)

# Construct a Wannierize workflow for the joint occ + empty manifold
wann_wf = WannierizeBlockWorkflow.from_other(wf,
    block=combined_proj_obj[0],
    pw_outdir=kcw_outdir,
    write_tb=True,
    calculate_bands=True)
wann_wf.calculator_parameters['pw2wannier'].prefix = 'kc_kcw'
wann_wf.directory /= '02-joint-wannierize'

# Run the Wannierization
wann_wf.run_while()
```

So what?

koopmans + AiiDA

UI practically unchanged:

```
$ koopmans tio2.json
```

koopmans + AiiDA

UI practically unchanged:

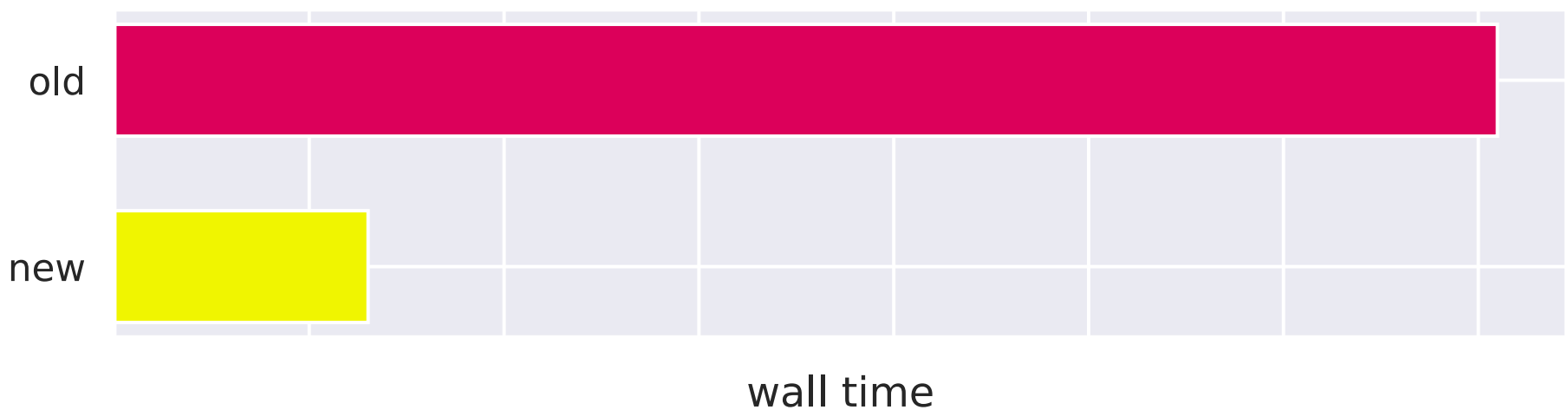
`$ koopmans tio2.json` → `$ koopmans run --engine=aiida tio2.json`

koopmans + AiiDA

UI practically unchanged:

`$ koopmans tio2.json` → `$ koopmans run --engine=aiida tio2.json`

but executed remotely and in parallel:



Outlook

Workflow-runner-agnostic workflows?

A lot of the pain in `koopmans` + `AiiDA` derives from wanting to have two workflow runners:

- `localhost` (the original `koopmans` implementation)
- `AiiDA`

[1]but still requires schemas, so not truly general

[2]S. P. Huber *et al. npj Comput Mater* **7**, 1–12 (2021)

Workflow-runner-agnostic workflows?

A lot of the pain in `koopmans` + `AiiDA` derives from wanting to have two workflow runners:

- `localhost` (the original `koopmans` implementation)
- `AiiDA`

Can we write runner-agnostic workflows?

[1]but still requires schemas, so not truly general

[2]S. P. Huber *et al. npj Comput Mater* **7**, 1–12 (2021)

Workflow-runner-agnostic workflows?

A lot of the pain in `koopmans` + `AiiDA` derives from wanting to have two workflow runners:

- `localhost` (the original `koopmans` implementation)
- `AiiDA`

Can we write runner-agnostic workflows?

- yes! CWL the most rigorous way[1], but `koopmans` (kind of) achieves this

[1]but still requires schemas, so not truly general

[2]S. P. Huber *et al. npj Comput Mater* **7**, 1–12 (2021)

Workflow-runner-agnostic workflows?

A lot of the pain in `koopmans` + `AiiDA` derives from wanting to have two workflow runners:

- `localhost` (the original `koopmans` implementation)
- `AiiDA`

Can we write runner-agnostic workflows?

- yes! CWL the most rigorous way[1], but `koopmans` (kind of) achieves this

Do we want to write runner-agnostic workflows?

[1]but still requires schemas, so not truly general

[2]S. P. Huber *et al. npj Comput Mater* **7**, 1–12 (2021)

Workflow-runner-agnostic workflows?

A lot of the pain in `koopmans` + `AiiDA` derives from wanting to have two workflow runners:

- `localhost` (the original `koopmans` implementation)
- `AiiDA`

Can we write runner-agnostic workflows?

- yes! CWL the most rigorous way[1], but `koopmans` (kind of) achieves this

Do we want to write runner-agnostic workflows?

- probably not…

[1]but still requires schemas, so not truly general

[2]S. P. Huber *et al. npj Comput Mater* **7**, 1–12 (2021)

Workflow-runner-agnostic workflows?

A lot of the pain in `koopmans` + `AiiDA` derives from wanting to have two workflow runners:

- `localhost` (the original `koopmans` implementation)
- `AiiDA`

Can we write runner-agnostic workflows?

- yes! CWL the most rigorous way[1], but `koopmans` (kind of) achieves this

Do we want to write runner-agnostic workflows?

- probably not…

(*cf.* Do we want to write calculator-agnostic workflows?

- yes! See Common Workflows[2])

[1] but still requires schemas, so not truly general

[2] S. P. Huber *et al. npj Comput Mater* **7**, 1–12 (2021)

If not, what should we do?

What do people want?

If not, what should we do?

What do people want?

- quick to write

If not, what should we do?

What do people want?

- quick to write
- understandable

If not, what should we do?

What do people want?

- quick to write
- understandable
- customisable

If not, what should we do?

What do people want?

- quick to write
- understandable
- customisable

What should we do when writing code?

If not, what should we do?

What do people want?

- quick to write
- understandable
- customisable

What should we do when writing code?

- always have inputs and outputs rigorously defined – think functionally!

If not, what should we do?

What do people want?

- quick to write
- understandable
- customisable

What should we do when writing code?

- always have inputs and outputs rigorously defined – think functionally!
- be modular wherever possible (e.g. separate responsibilities)

If not, what should we do?

What do people want?

- quick to write
- understandable
- customisable

What should we do when writing code?

- always have inputs and outputs rigorously defined – think functionally!
- be modular wherever possible (e.g. separate responsibilities)
- be abstract wherever possible

If not, what should we do?

What do people want?

- quick to write
- understandable
- customisable

What should we do when writing code?

- always have inputs and outputs rigorously defined – think functionally!
- be modular wherever possible (e.g. separate responsibilities)
- be abstract wherever possible
- make it easy for people to tweak and combine workflows

If not, what should we do?

What do people want?

- quick to write
- understandable
- customisable

What should we do when writing code?

- always have inputs and outputs rigorously defined – think functionally!
- be modular wherever possible (e.g. separate responsibilities)
- be abstract wherever possible
- make it easy for people to tweak and combine workflows
  - (should `AiiDA` be able to read and dump `.cwl` files?)

If not, what should we do?

What do people want?

- quick to write
- understandable
- customisable

What should we do when writing code?

- always have inputs and outputs rigorously defined – think functionally!
- be modular wherever possible (e.g. separate responsibilities)
- be abstract wherever possible
- make it easy for people to tweak and combine workflows
  - (should `AiiDA` be able to read and dump `.cwl` files?)
- for `AiiDA` , continue efforts to simplify


Fin

References

S. P. Huber *et al.* Common workflows for computing material properties using different quantum engines. *npj Comput Mater* **7,** 1–12 (2021).