Ewoks Tutorial For Developers

Introduction

This is a hand-on tutorial where concepts will be introduced in steps.

At the end of each step there will be a summary slide with an exercise which gives you time to try out the new concept.

Resources

Slides

https://ewoksfordevs.readthedocs.io

Ewoks Documentation

https://ewoks.readthedocs.io

Tutorial Resources

```
git clone https://gitlab.esrf.fr/workflow/ewokstutorials/ewoksfordevs
```

- jupyter notebook that generated these slides
- `results` directory with the results of all exercises
- links to the slides in PDF format in the README

When to use Ewoks

- Online data processing / beamline automation (Bliss, Daiquiri or MXcube)
- Make data processing *reproducible*, *repeatable* and *traceable*
- Ewoks takes care of *location* (beamline computer, Slurm, user at home, ...)
- Ewoks takes care of *integration* (beamline acquisition, upload results to the data portal, ...)
- Libraries/scripts/notebooks that are *ewoksified* can be used by all Ewoks users

Program

Part 1: Ewoks workflows

- Ewoks workflow definition
- Create workflows
- Execute workflows
- Workflows GUI

Part 2: Ewoks Tasks

- Create workflow tasks
- Use new tasks in the GUI
- Use third-party task libraries

Part 3: Ewoks Integration

- Remote execution
- Slurm execution
- Online data processing in Bliss

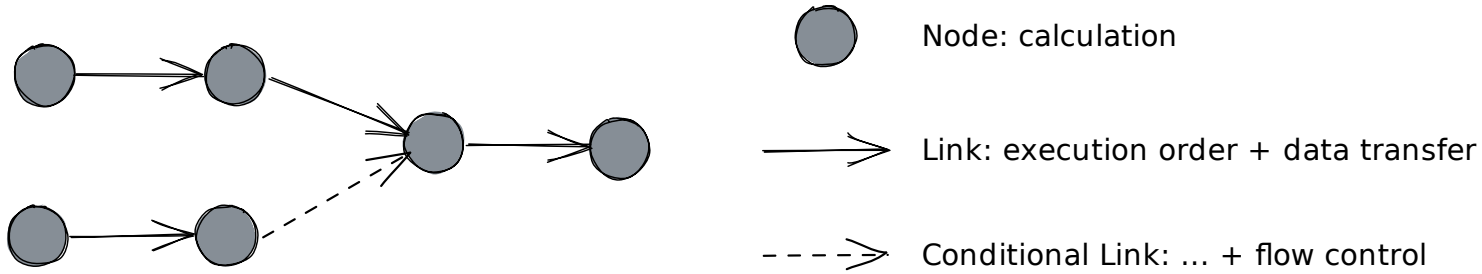
Part 4: Final remarks

- Ewoks features not covered
- Deploy software on Slurm

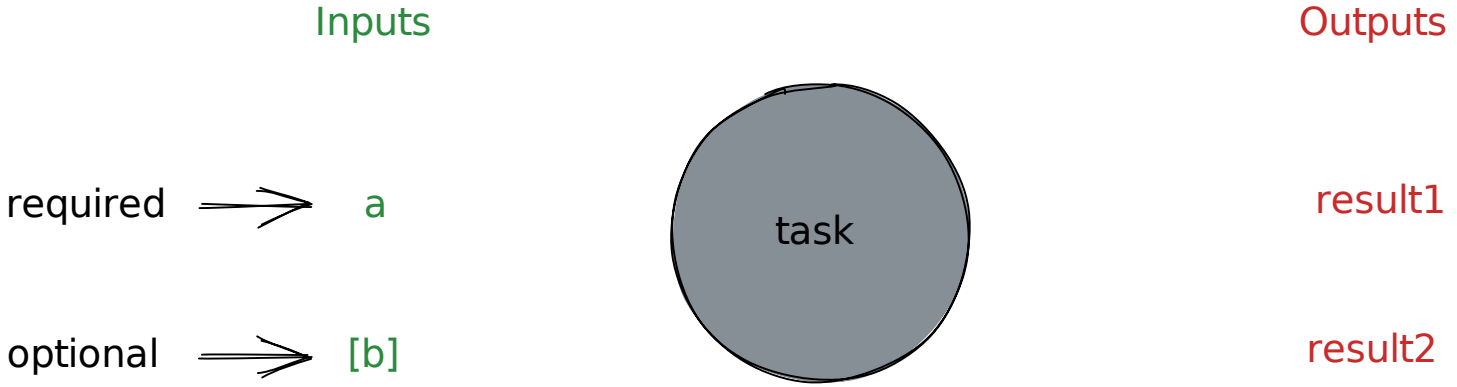
Part 1: Ewoks Workflows

- Ewoks workflow definition
- Create workflows
- Execute workflows
- Visualize workflows (GUI)

Ewoks workflow definition

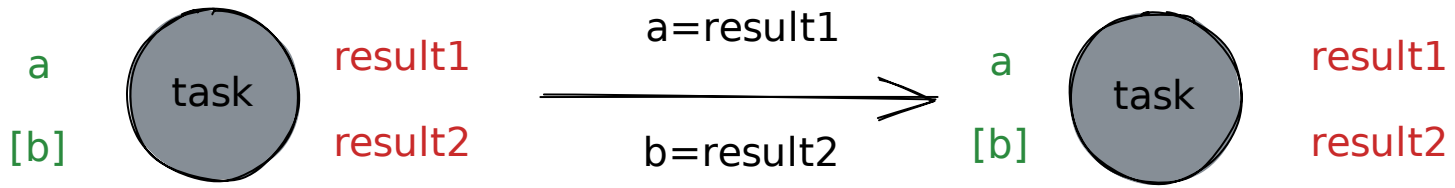


Ewoks Workflow Node

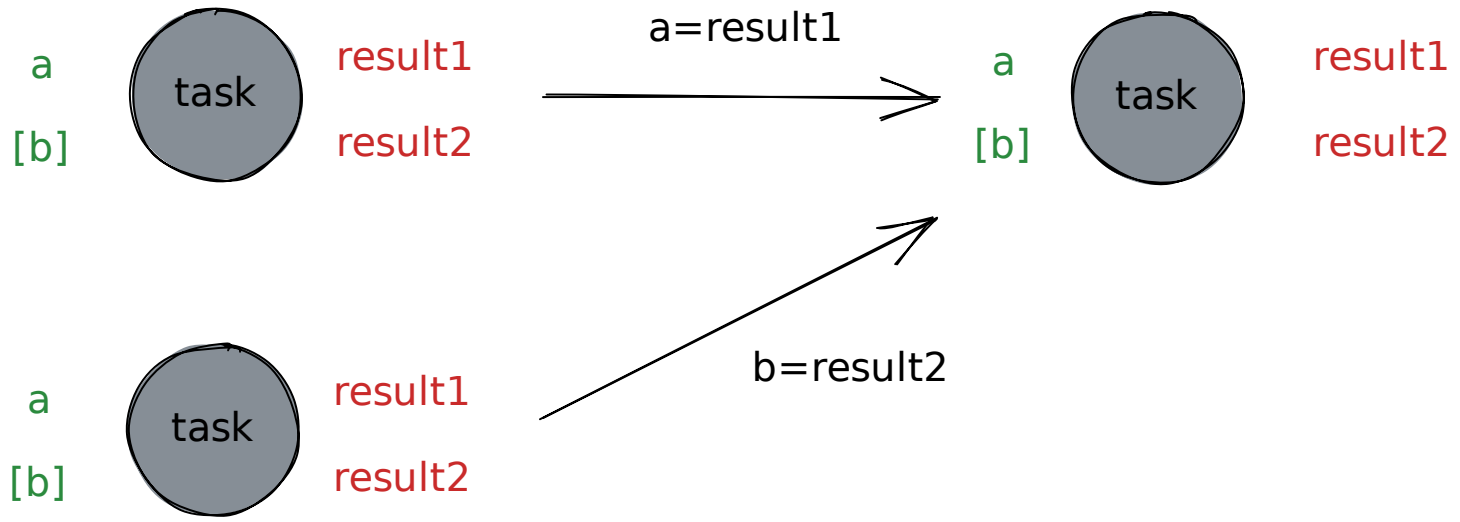


result1, result2 = task(a, b=10)

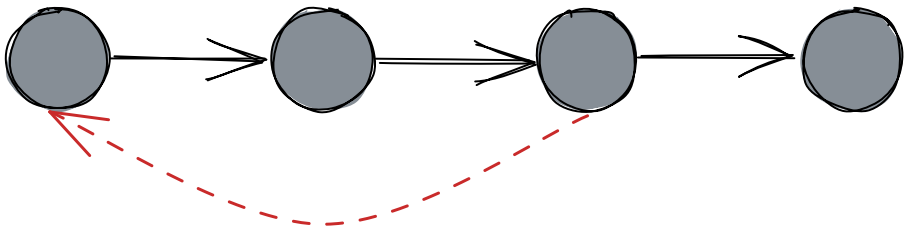
Link: data transfer



Link: data transfer

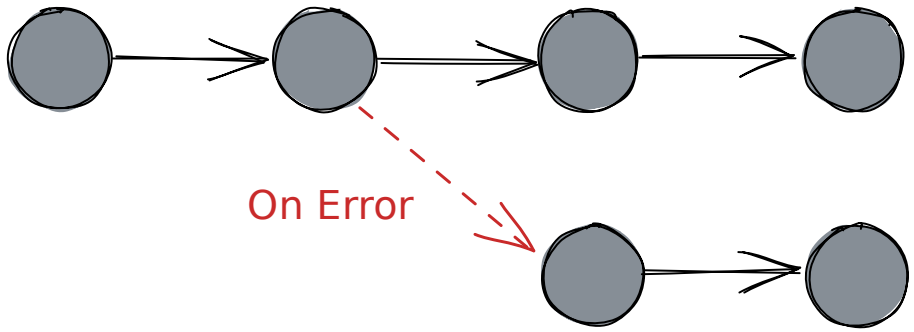


Conditional Link: flow control



result1 == 10 AND result2 == False

Conditional Link: flow control



Create Ewoks Workflows

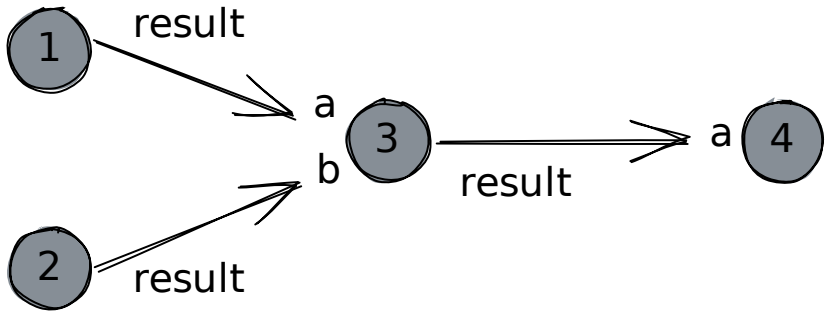
An Ewoks workflow can be created

- in Python using basic types
- in a web browser (Ewoksweb)

Other ways of creating a workflow

- in a desktop GUI (Orange)
- in Python using the `TaskGraph` class
- JSON or YAML file

Create a workflow in Python



Add two numbers

a

[b]

[delay]

result

Define workflow nodes

In [1]:
```
node1 = {
    "id": "node1",
    "task_type": "class",
    "task_identifier": "ewokscore.tests.examples.tasks.sumtask.SumTask"
}
node2 = {
    "id": "node2",
    "task_type": "class",
    "task_identifier": "ewokscore.tests.examples.tasks.sumtask.SumTask"
}
```

- *id*: unique within the context of a workflow
- *task_type*: `"class"`
- *task_identifier*: full python qualifier name

In [2]:

```
node3 = {
    "id": "node3",
    "task_type": "class",
    "task_identifier": "ewokscore.tests.examples.tasks.sumtask.SumTask"
}
node4 = {
    "id": "node4",
    "task_type": "class",
    "task_identifier": "ewokscore.tests.examples.tasks.sumtask.SumTask"
}
```

Define workflow links

In [3]:
```
link1 = {"source": "node1", "target": "node3"}
link2 = {"source": "node2", "target": "node3"}
link3 = {"source": "node3", "target": "node4"}
```

Define workflow links

In [3]:
```
link1 = {"source": "node1", "target": "node3"}
link2 = {"source": "node2", "target": "node3"}
link3 = {"source": "node3", "target": "node4"}
```

Data transfer between the nodes

In [4]:
```
link1["data_mapping"] = [
    {"source_output": "result", "target_input": "a"}
]
link2["data_mapping"] = [
    {"source_output": "result", "target_input": "b"}
]
link3["data_mapping"] = [
    {"source_output": "result", "target_input": "a"}
]
```

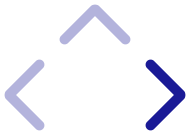
Define workflow links

In [3]:
```
link1 = {"source": "node1", "target": "node3"}
link2 = {"source": "node2", "target": "node3"}
link3 = {"source": "node3", "target": "node4"}
```

Data transfer between the nodes

In [4]:
```
link1["data_mapping"] = [
    {"source_output": "result", "target_input": "a"}
]
link2["data_mapping"] = [
    {"source_output": "result", "target_input": "b"}
]
link3["data_mapping"] = [
    {"source_output": "result", "target_input": "a"}
]
```

So a link is defined by a `source`, `target` and `data_mapping`



Define workflow inputs

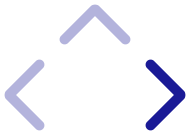
In [5]:

```
node1["default_inputs"] = [
    {"name": "a", "value": 2},
    {"name": "b", "value": 3},
]
node2["default_inputs"] = [
    {"name": "a", "value": 4},
    {"name": "b", "value": 5},
]
```

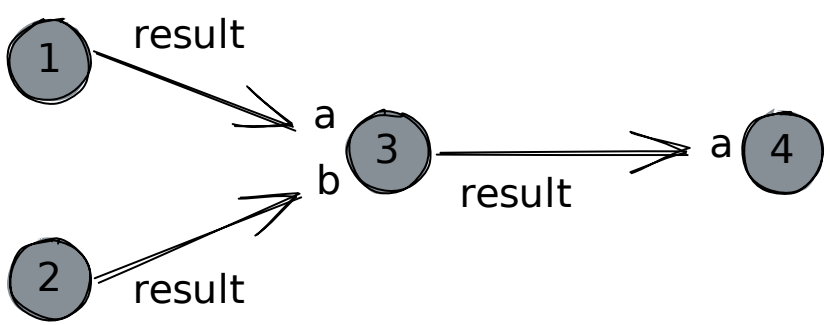
Define workflow inputs

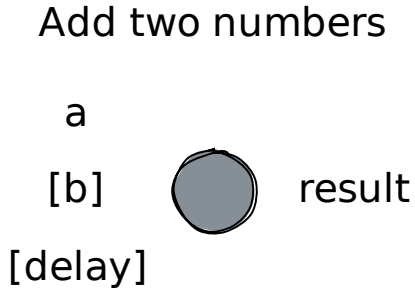
In [5]:
```
node1["default_inputs"] = [
    {"name": "a", "value": 2},
    {"name": "b", "value": 3},
]
node2["default_inputs"] = [
    {"name": "a", "value": 4},
    {"name": "b", "value": 5},
]
```

So a node is defined by an `id`, `task_type`, `task_identifier` and (optionally) `default_inputs`



Define workflow

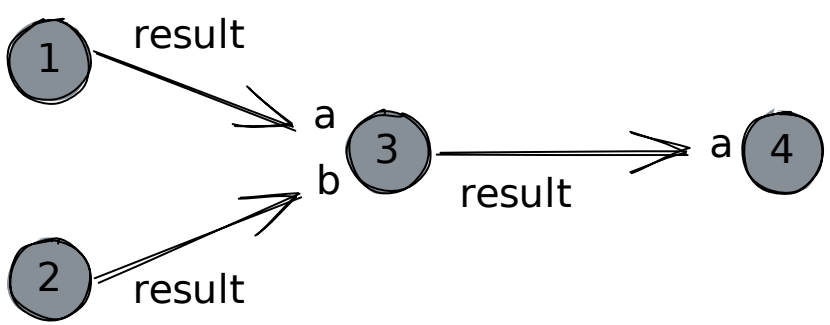


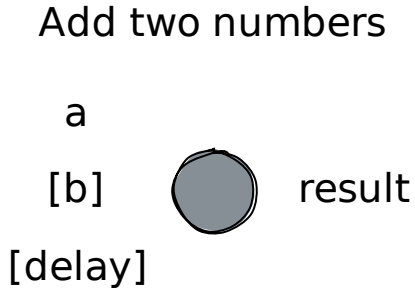


In [6]:
```
workflow = {
    "graph": {"id": "workflow1"},
    "nodes": [node1, node2, node3, node4],
    "links": [link1, link2, link3],
}
```

You have created your first Ewoks workflow!

Define workflow





In [6]:
```
workflow = {
    "graph": {"id": "workflow1"},
    "nodes": [node1, node2, node3, node4],
    "links": [link1, link2, link3],
}
```

You have created your first Ewoks workflow!

A workflow is defined by `nodes`, `links` and (optionally) `graph`.

More details: https://ewokscore.readthedocs.io/en/latest/definitions.html

Execute Ewoks workflows

```
pip install ewoks
```

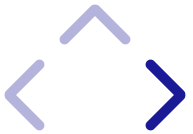
Execute Ewoks workflows

```
pip install ewoks
```

Execute workflow from python

In [7]:
```
from ewoks import execute_graph

execute_graph(workflow)
```

Out[7]:
```
{'result': 14}
```



Execute workflow from the command line

Save with the `convert_graph` function

In [8]:
```
from ewoks import convert_graph

convert_graph(workflow, "results/workflow1.json")
```

Out[8]:
```
'results/workflow1.json'
```

You can also use `json.dump` but `convert_graph` adds validation.

Execute with the `ewoks` command (the output parameters will be discussed later)

In [9]:
```
!ewoks execute results/workflow1.json --outputs=end --merge-outputs
```

```
####################################
# Execute workflow 'results/workflow1.json'
####################################

RESULTS:
{'result': 14}

FINISHED
```

✏ Exercise: make a workflow and execute it

Create a workflow with `ewokscore.tests.examples.tasks.sumtask.SumTask` tasks

```
w = {"graph": {"id": "workflow1"}, "nodes": [...], "links": [...]}
```

Execute

```
from ewoks import execute_graph
execute_graph(w)
```

Save

```
from ewoks import convert_graph
convert_graph(w, "results/workflow1.json")
```

Execute from the command line

```
ewoks execute results/workflow1.json --outputs=end --merge-outputs
```

Documentation: https://ewokscore.readthedocs.io/en/latest/definitions.html

Execution engine

In Python

```
execute_graph(workflow, engine="...")
```

On the command line

```
ewoks execute --engine=...
```

Currently the engines supported by Ewoks are: `"dask"`, `"ppf"` (for `pypushflow`) and `"orange"`.

When using an execution engine, it needs to be installed first

```
pip install "ewoks[dask,ppf,orange]"
```

engine	Loops	Conditional Links	Parallel execution	Interaction (GUI)	An Ewoks task is enough
`None`	✗	✗	✗	✗	✓
`"dask"`	✗	✗	✓	✗	✓
`"ppf"`	✓	✓	✓	✗	✓
`"orange"`	✗	✗	(✓)	✓	✗

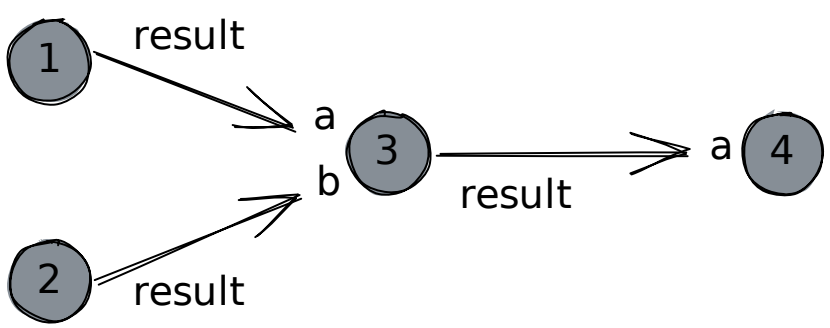
Example on how to use and configure the execution engines:

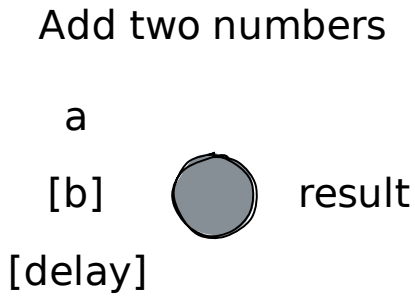
https://ewoks.readthedocs.io/en/latest/tutorials/running_workflows.html

Overview on the overhead you can expect for each engine:

https://ewoks.readthedocs.io/en/latest/tutorials/benchmark.html

An example of parallel execution





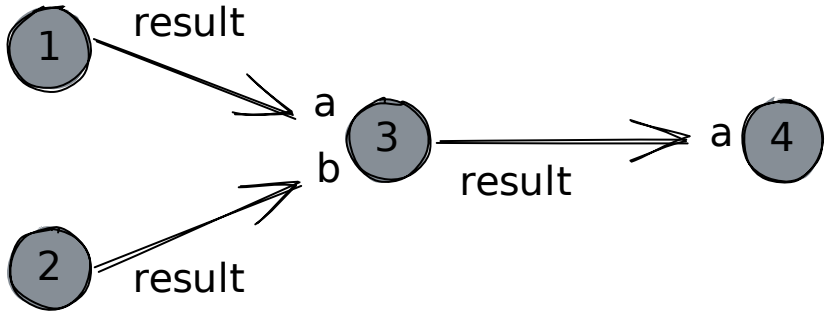
Time without parallel execution (the workflow parameters will be discussed later)

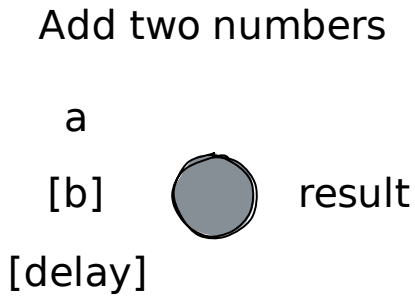
In [10]:
```
%%time
execute_graph(
    "results/workflow1.json",
    inputs=[{"name": "delay", "value": 1, "all": True}],
)
```

```
CPU times: user 3.52 ms, sys: 13.8 ms, total: 17.3 ms
Wall time: 4.04 s
```

Out[10]:
```
{'result': 14}
```

An example of parallel execution





Time with parallel execution (1 sec faster because *node1* and *node2* run in parallel)

In [11]:
```
%%time
execute_graph(
    "results/workflow1.json",
    inputs=[{"name": "delay", "value": 1, "all": True}],
    engine="ppf",
)
```

```
CPU times: user 31.8 ms, sys: 44.7 ms, total: 76.4 ms
Wall time: 3.16 s
```

Out[11]:
```
{'result': 14}
```

✏ Exercise: execute a workflow with different engines

From Python

```python
from ewoks import execute_graph

execute_graph(
    "results/workflow1.json",
    engine="ppf",
    inputs=[{"name": "delay", "value": 1, "all": True}],
    pool_type='thread' # Only required for Windows
)
```

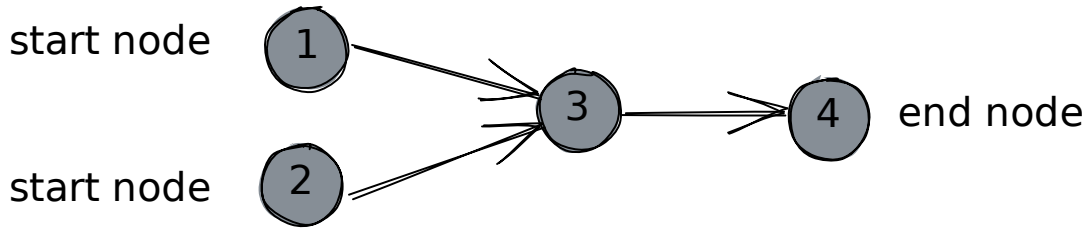
From the command line

```
ewoks execute results/workflow1.json --engine=ppf -p delay=1 \
              --inputs=all [-o pool_type=thread] # For Windows
```

See the Ewoks documentation for how to use and configure the execution engines.

ℹ *Windows users can time commands in a Powershell with* `Measure-Command {ewoks execute ...}` *instead of* `time`.

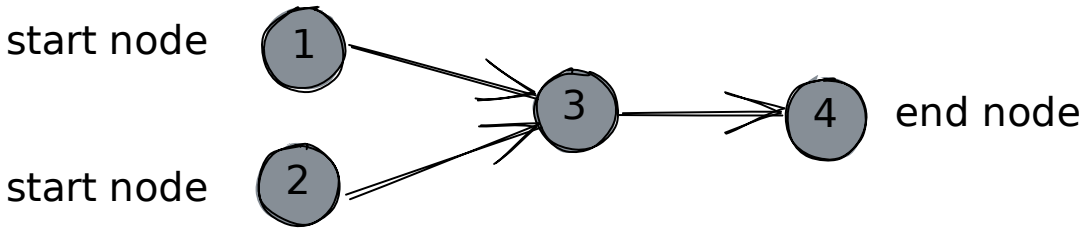
Workflow execution inputs and outputs



When providing inputs to a workflow upon execution, each argument can be passed to either

- all start nodes (default)
- all nodes
- all nodes with a specific `task_identifier`
- one specific node identified by the node `id` (or `label`)

Nodes that receive an input they don't have will silently ignore it.



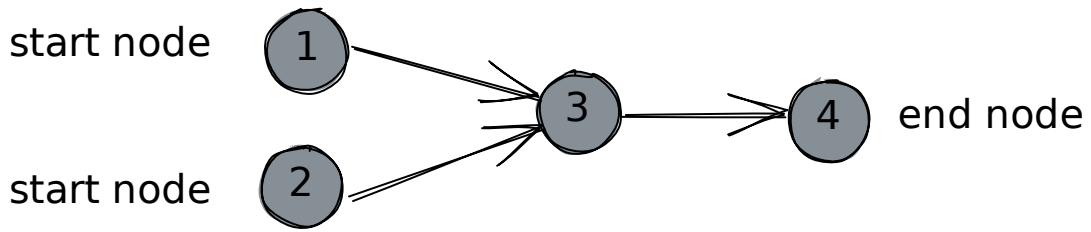
The output of the workflow is the combination of the outputs of either

- all end nodes (default in python)
- all nodes
- all nodes with a specific `task_identifier`
- one or more specific nodes identified by their node `id` (or `label`)
- none of the nodes, i.e. no output at all (default for the CLI)

You can select only specific output variables of a node or all of them (default).

Node outputs can be merged (default in python) or kept separately (default for the CLI).

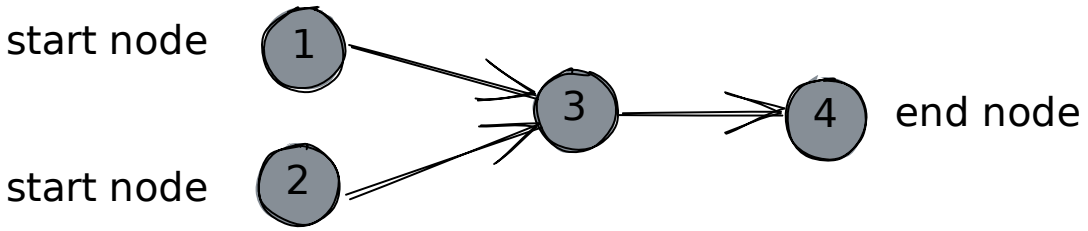
Workflow inputs



Provide the input `a=1` to `node1` and `b=1` to `node2`

In [12]:
```
execute_graph(
    workflow,
    inputs=[
        {"name": "a", "value": 1, "id": "node1"},
        {"name": "b", "value": 1, "id": "node2"},
    ],
)
```

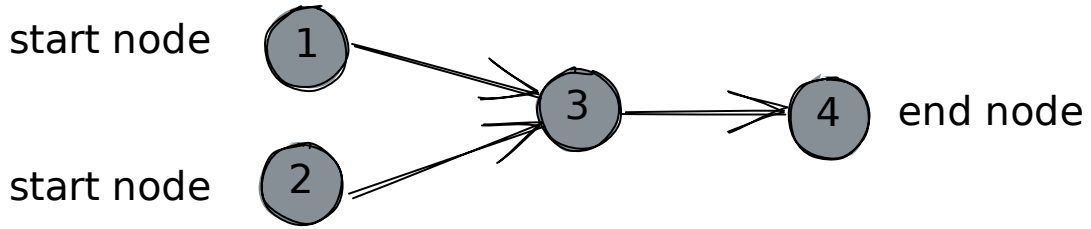
Out[12]:
```
{'result': 9}
```



Provide the input `a=1` to `node1` and `b=1` to `node2`

In [13]:
```
!ewoks execute results/workflow1.json --outputs=end --merge-outputs \
      -p node1:a=1 -p node2:b=1  # noqa E999
```

```
####################################
# Execute workflow 'results/workflow1.json'
####################################

RESULTS:
{'result': 9}

FINISHED
```

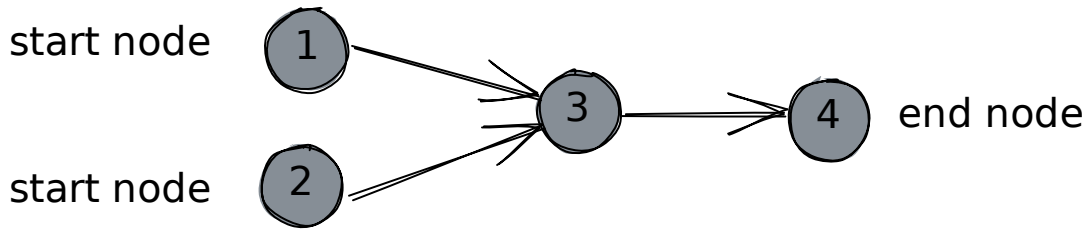


Note that *node3* can receive three types of inputs:

- data mapping (defined in links' `data_mapping` in *workflow1.json*)
- execution parameters (defined when executing the workflow)
- default inputs (defined in nodes' `default_inputs` in *workflow1.json*)

Priority:

Data mapping **>** Execution parameters **>** Default inputs



Note that *node3* can receive three types of inputs:

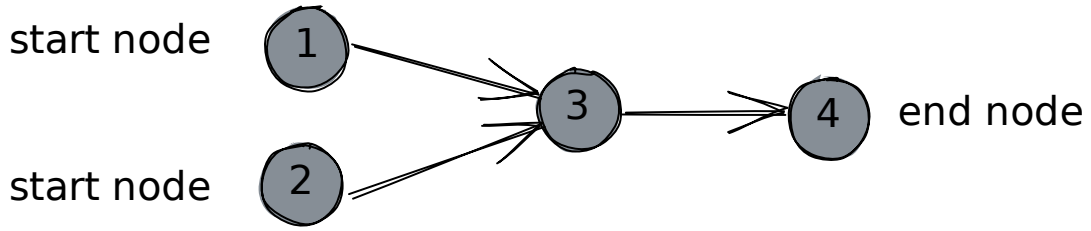
- data mapping (defined in links' `data_mapping` in *workflow1.json*)
- execution parameters (defined when executing the workflow)
- default inputs (defined in nodes' `default_inputs` in *workflow1.json*)

Priority:

Data mapping **>** Execution parameters **>** Default inputs

Ex: The default value of `a` in node 3 is overwritten by the value `a` given when executing the workflow, which is overwritten by the result of the node 1

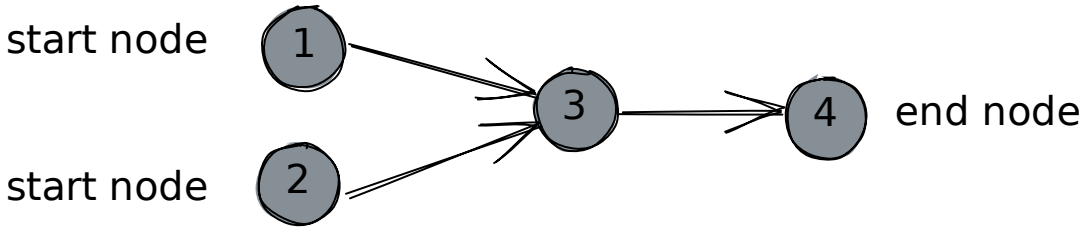
Workflow outputs



In Python, the output of a workflow is the `merged` output of all `end` nodes (in this case there is only one end node with one output variable called `result`)

In [14]:
```
execute_graph(workflow)
```

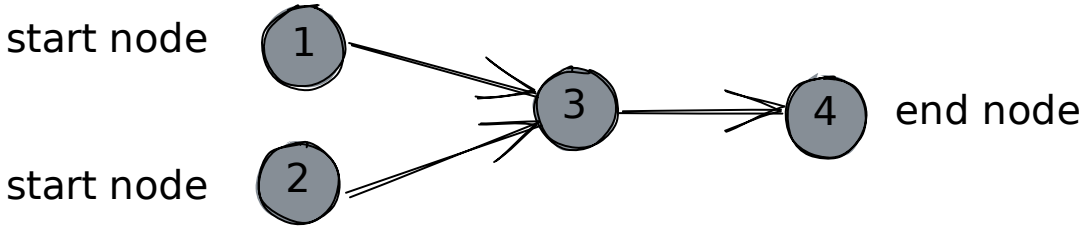
Out[14]:
```
{'result': 9}
```



On the command line, nothing is returned by default. So to obtain the same as in Python

In [15]:
```
!ewoks execute results/workflow1.json --outputs=end --merge-outputs
```

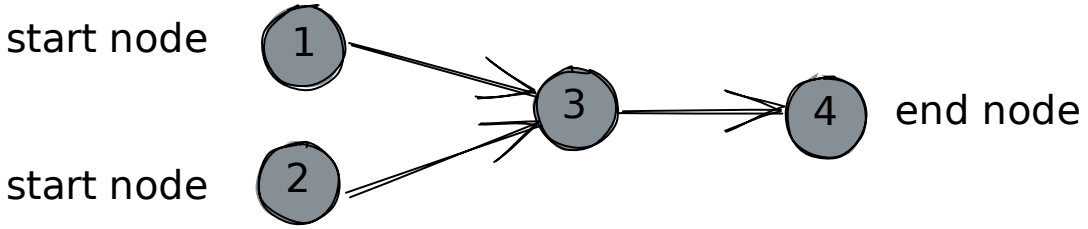
```
##################################
# Execute workflow 'results/workflow1.json'
##################################

RESULTS:
{'result': 14}

FINISHED
```



Return the output of all `end` nodes separately

In [16]:
```
execute_graph(workflow, merge_outputs=False)
```

Out[16]:
```
{'node4': {'result': 9}}
```



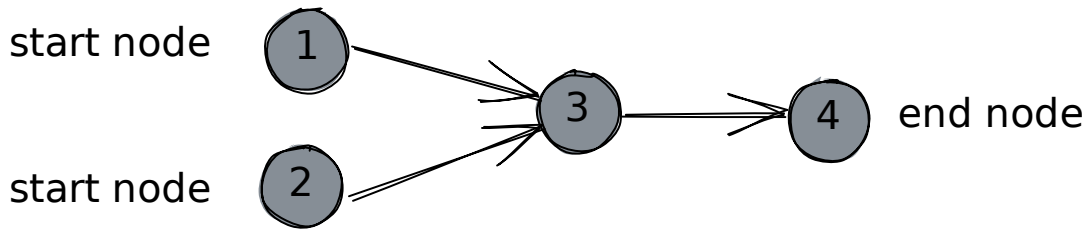
Return the output of all `end` nodes separately

In [16]:
```
execute_graph(workflow, merge_outputs=False)
```

Out[16]:
```
{'node4': {'result': 9}}
```

In [17]:
```
!ewoks execute results/workflow1.json --outputs=end
```

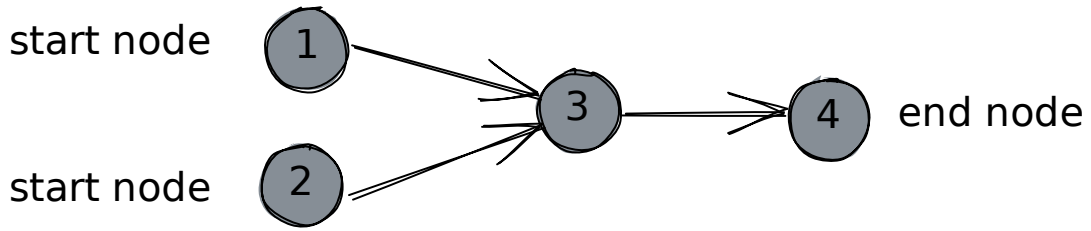
```
####################################
# Execute workflow 'results/workflow1.json'
####################################

RESULTS:
{'node4': {'result': 14}}

FINISHED
```



Output can be fully customized: for example `result` from `node2` and `node4`

In [18]:
```
execute_graph(
    workflow,
    outputs=[
        {"name": "result", "id": "node2"},
        {"name": "result", "id": "node4"},
    ],
    merge_outputs=False,
)
```

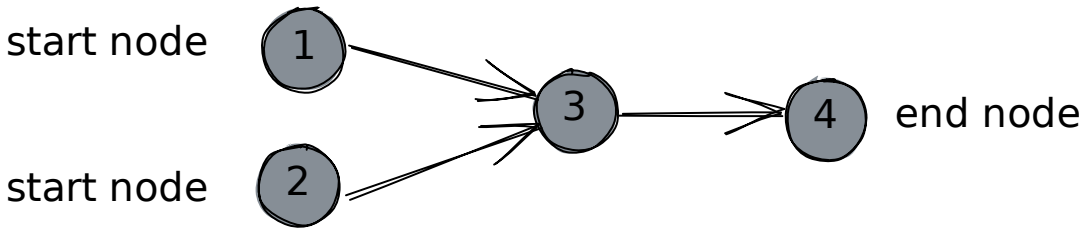
Out[18]:
```
{'node2': {'result': 5}, 'node4': {'result': 9}}
```



Return the outputs from all nodes

In [19]:
```
execute_graph(workflow, outputs=[{"all": True}], merge_outputs=False)
```

Out[19]:
```
{'node1': {'result': 4},
 'node2': {'result': 5},
 'node3': {'result': 9},
 'node4': {'result': 9}}
```



Return the outputs from all nodes

In [20]:
```
!ewoks execute results/workflow1.json --outputs=all
```

```
#####################################
# Execute workflow 'results/workflow1.json'
#####################################

RESULTS:
{'node1': {'result': 5},
 'node2': {'result': 9},
 'node3': {'result': 14},
 'node4': {'result': 14}}

FINISHED
```

✏ Exercise: execute a workflow with inputs/outputs

```
execute_graph(
    workflow,
    inputs=[
        {"name": "a", "value": 2, "id": "node1"},
        {"name": "a", "value": 3, "id": "node2"},
    ],
    outputs=[
        {"name": "result", "id": "node1"},
        {"name": "result", "id": "node4"},
    ],
    merge_outputs=False,
)
```

```
ewoks execute ... -p node1:a=2 -p node2:a=3 --outputs=all
```

Documentation: https://ewokscore.readthedocs.io/en/latest/execute_io.html

Workflow GUI

Install the Ewoks REST server (*ewoksserver*) with a frontend (*ewoksweb*)

```
pip install "ewoksserver[frontend]"
```

Workflow GUI

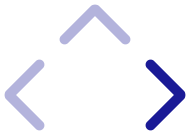
Install the Ewoks REST server (*ewoksserver*) with a frontend (*ewoksweb*)

```
pip install "ewoksserver[frontend]"
```

Start the backend, create a workflow and save it

```
ewoks-server --port 5174  # The default port is 8000

RESOURCE DIRECTORY:
/path/to/resource/directory

CELERY:
Not configured (local workflow execution)

EWOKS:
{...}

Uvicorn running on http://127.0.0.1:5174 (Press CTRL+C to quit)
```

`RESOURCE DIRECTORY` is where all workflows will be saved. `CELERY` specifies where workflows are executed (locally by default). `EWOKS` specifies where execution events are saved.



✏ Exercise: use the web GUI

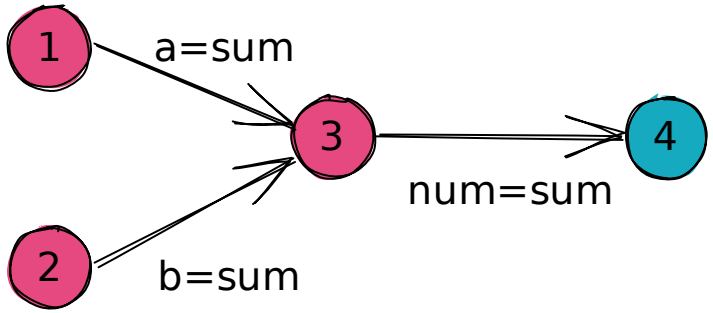
- create a workflow
- save a workflow
- download/upload a workflow
- execute a workflow

```
pip install "ewoksserver[frontend]"
ewoks-server --port 5174
```

Part 2: Ewoks Tasks

- Create workflow tasks
- Use new tasks in the GUI
- Use third-party task libraries

We will create two tasks in order to define a workflow like this



Add numbers

a

[b]

sum

Generate numbers

[start]

[stop]

numbers

[num]

Create workflow tasks

In [22]:
```
%%writefile results/tasks.py
from ewokscore import Task

class Add(
    Task,
    input_names=["a"],
    optional_input_names=["b"],
    output_names=["sum"],
):
    def run(self):
        if self.missing_inputs.b:
            self.outputs.sum = self.inputs.a
        else:
            self.outputs.sum = self.inputs.a + self.inputs.b
```

```
Overwriting results/tasks.py
```

Task attributes used in the run method:

- `inputs`: object with inputs as fields
- `outputs`: object with outputs as fields
- `missing_inputs`: object with missing inputs as fields

In [23]:
```
%%writefile -a results/tasks.py


import numpy


class Linspace1(
    Task,
    optional_input_names=["start", "stop", "num"],
    output_names=["numbers"],
):
    def run(self):
        start = self.get_input_value("start", 0)
        stop = self.get_input_value("stop", 1)
        num = self.get_input_value("num", 1)
        self.outputs.numbers = numpy.linspace(start, stop, num)
```

```
Appending to results/tasks.py
```

Task attributes used in the run method:

- `get_input_value`: get a input by its name. Optionally specify a default value
- `outputs`: object with outputs as fields

In [24]:
```
%%writefile -a results/tasks.py


class Linspace2(
    Task,
    optional_input_names=["start", "stop", "num"],
    output_names=["numbers"],
):
    """Same as Linspace1, alternative implementation"""

    def run(self):
        inputs = self.get_input_values()
        inputs.setdefault("start", 0)
        inputs.setdefault("stop", 1)
        inputs.setdefault("num", 1)
        self.outputs.numbers = numpy.linspace(**inputs)
```

```
Appending to results/tasks.py
```

Task attributes used in the run method:

- `get_input_values`: returns a dictionary with input names as keys
- `outputs`: object with outputs as fields

In [25]:

```
node1 = {
    "id": "node1",
    "task_type": "class",
    "task_identifier": "results.tasks.Add",
}
node2 = {
    "id": "node2",
    "task_type": "class",
    "task_identifier": "results.tasks.Add",
}
node3 = {
    "id": "node3",
    "task_type": "class",
    "task_identifier": "results.tasks.Add",
}
node4 = {
    "id": "node4",
    "task_type": "class",
    "task_identifier": "results.tasks.Linspace2",
}
```

Note that `task_identifier` resolves the same way a Python `import` would.

In [26]:

```
link1 = {"source": "node1", "target": "node3"}
link2 = {"source": "node2", "target": "node3"}
link3 = {"source": "node3", "target": "node4"}
link1["data_mapping"] = [
    {"source_output": "sum", "target_input": "a"}
]
link2["data_mapping"] = [
    {"source_output": "sum", "target_input": "b"}
]
link3["data_mapping"] = [
    {"source_output": "sum", "target_input": "num"}
]
```

In [27]:

```
workflow = {
    "graph": {"id": "workflow2"},
    "nodes": [node1, node2, node3, node4],
    "links": [link1, link2, link3],
}
```

Save with the `convert_graph` method

In [28]:
```
convert_graph(
    workflow,
    "results/workflow2.json",
    inputs=[{"name": "a", "value": 1}, {"name": "b", "value": 1}],
)
```

Out[28]:
```
'results/workflow2.json'
```

Note that both `convert_graph` and `execute_graph` take inputs.

The `inputs` provided to `convert_graph` override the `default_inputs` before saving the workflow.

The `inputs` provided to `execute_graph` override the `default_inputs` after loading the workflow.

Execute in Python

In [29]:
```
execute_graph("results/workflow2.json")
```

Out[29]:
```
{'numbers': array([0.        , 0.33333333, 0.66666667, 1.
])}
```

Execute from the command line

In [30]:
```
!ewoks execute results/workflow2.json --outputs=all --merge-outputs
```

```
####################################
# Execute workflow 'results/workflow2.json'
####################################

RESULTS:
{'numbers': array([0.        , 0.33333333, 0.66666667, 1.
]), 'sum': 4}

FINISHED
```

✏ Exercise: create ewoks tasks and use them in a workflow

```python
from ewokscore import Task


class Add(
    Task,
    input_names=["a"],
    optional_input_names=["b"],
    output_names=["sum"],
):
    def run(self):
        if self.missing_inputs.b:
            self.outputs.sum = self.inputs.a
        else:
            self.outputs.sum = self.inputs.a + self.inputs.b
```

Hello world: https://ewoks.readthedocs.io/en/latest/tutorials/hello_world.html

Documentation: https://ewokscore.readthedocs.io/en/latest/definitions.html#task-implementation

Use new tasks in the GUI

Create a workflow with these task classes in the Ewoksweb application (discover tasks from the python module `results.tasks`)

```
ewoks-server --port 5174
```

Use new tasks in the GUI

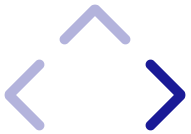
Create a workflow with these task classes in the Ewoksweb application (discover tasks from the python module `results.tasks`)

```
ewoks-server --port 5174
```

To **open** a workflow in the Ewoks desktop application

```
pip install "ewoks[orange]" pyqt5

ewoks execute results/workflow2.json --engine=orange
```

You need to create an Orange3 add-on project with widgets for all tasks in this workflow in order to **create** the workflow from scratch in the Ewoks desktop application and **save** it in OWS format.



✏ Exercise: use new tasks in a GUI

Web GUI (perform task discovery first)

```
ewoks-server --port 5174
```

Desktop GUI (only load, not create)

```
ewoks execute results/workflow2.json --engine=orange
```

⚠ *On Windows, running `pip install ewoks[orange]` may raise an error because the installation paths are too long.*

If so, you can enable long paths at `Edit group policy > Computer Configuration > Administrative Templates > System > Filesystem > Enable Win32 long paths` in your Windows system.

Use third-party task libraries

Find task libraries

To find existing task libraries: https://ewoks.esrf.fr

For example a library with tasks for image registration

```
pip install "ewoksndreg[full]"
```

Use task libraries

To use the new tasks in the web GUI, they need to be discovered by the server (in this case from module name `ewoksndreg.tasks`).

We did it manually in the previous exercise but the `--rediscover-tasks` arg allows to automatically discover tasks in the current Python env:

```
ewoks-server --port 5174 --rediscover-tasks
```

The desktop GUI always re-discovers all tasks when starting the application

```
ewoks-canvas
```

✏ Exercise: create an image alignment workflow in Orange and load it in the web app or vice versa

Install the task library and create an OWS file

```
pip install "ewoksndreg[full]"
ewoks-canvas
```

Convert to JSON format

```
ewoks convert ...
```

Discover `ewoksndreg.tasks` and load the JSON file

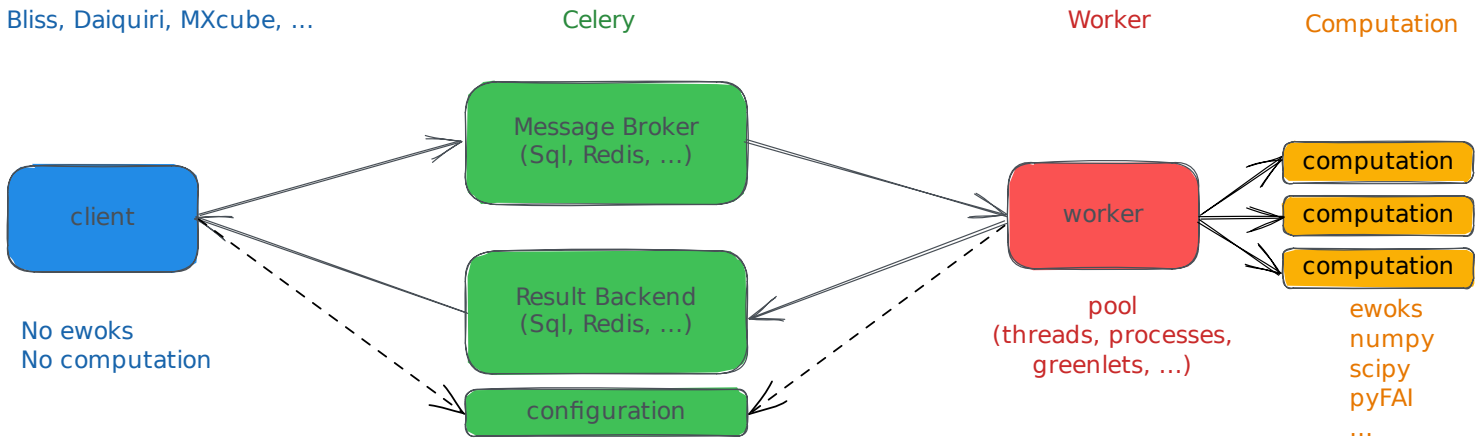
```
ewoks-server --port 5174
```

Part 3: Ewoks Integration

- Remote execution
- Slurm execution
- Online data processing in Bliss

Remote execution

Supported through `ewoksjob` with `celery` as the underlying technology.



Client environment (use either *sql* or *redis*)

```
pip install "ewoksjob[sql,redis]" ewoks
```

Client environment (use either *sql* or *redis*)

```
pip install "ewoksjob[sql,redis]" ewoks
```

Worker environment (use either *sql* or *redis*)

```
pip install "ewoksjob[sql,redis,worker]"
```

Client environment (use either *sql* or *redis*)

```
pip install "ewoksjob[sql,redis]" ewoks
```

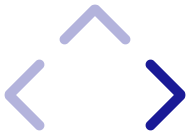
Worker environment (use either *sql* or *redis*)

```
pip install "ewoksjob[sql,redis,worker]"
```

Monitoring environment (does not work with `sql` client/worker)

```
pip install "ewoksjob[monitor]"
```

Documentation: https://ewoksjob.readthedocs.io



Celery configuration

Both client and worker need this information to communicate with the message broker and the result backend

In [31]:
```
%%writefile results/celeryconfig_sql.py
broker_url = "sqla+sqlite:///celery.db"
result_backend = "db+sqlite:///celery_results.db"

result_serializer = "pickle"
accept_content = [
    "application/json",
    "application/x-python-serialize",
]
result_expires = 600
task_remote_tracebacks = True
```

```
Overwriting results/celeryconfig_sql.py
```

If you want monitoring to work, install Redis (or RabbitMQ)

```
apt install redis-server
conda install redis-server
```

and start a server in a separate terminal

```
redis-server
```

The first fields of the Celery configuration become

```
broker_url = "redis://localhost:6379/3"
result_backend = "redis://localhost:6379/4"
```

Start and test a worker

Start a worker in the worker environment

```
ewoksjob --config=results.celeryconfig_sql worker
```

You should get

```
-------------- celery@lindenolf v5.3.0b1 (dawn-chorus)
--- ***** -----
-- ******* ---- Linux-5.4.0-144-generic-x86_64 ...
- *** --- * ---
- ** ---------- [config]
- ** ---------- .> app:          ewoks:0x7ff87d66a700
- ** ---------- .> transport:    sqla+sqlite:///celery.db
- ** ---------- .> results:      sqlite:///celery_results.db
- *** --- * --- .> concurrency: 4 (prefork)
-- ******* ---- .> task events: OFF (enable -E to monitor tasks)
--- ***** -----
 -------------- [queues]
                .> celery  exchange=celery(direct) key=celery
```

Warning: Windows users should use `ewoksjob ... worker --pool=threads` (prefork is not supported)

With Redis

Start the worker in the worker environment

```
ewoksjob --config=results.celeryconfig_redis worker
```

Start the monitor in the monitoring environment

```
ewoksjob --config=results.celeryconfig_redis monitor

[I 230320 11:44:13 command:162] Visit me at http://localhost:5555
[I 230320 13:17:17 command:170] Broker: redis://localhost:6379/3
```

Submit a test workflow in the client environment.

Define the environment variable

```
export EWOKS_CONFIG_URI=results.celeryconfig_sql
```

replacing `sql` by `redis` if needed.

Then in Python,

```
from ewoksjob.client import submit_test
result = submit_test().get()
```

✏ Exercise: setup and test remote execution

Documentation: https://ewoksjob.readthedocs.io/

Environment variable for client and worker

```
export EWOKS_CONFIG_URI=results.celeryconfig_sql  # Linux and macOS
set EWOKS_CONFIG_URI=results.celeryconfig_sql  # Windows
```

Worker

```
ewoksjob worker  # Linux and macOS
ewoksjob worker --pool=threads  # Windows
```

Client

```
from ewoksjob.client import submit_test

submit_test().get()
```

Submit a workflow from Python

This is a local execution (blocking call)

```
from ewoks import execute_graph

result = execute_graph(*args, **kwargs)
```

This is the equivalent for a remote execution (non-blocking call)

```
from ewoksjob.client import submit

future = submit(args=args, kwargs=kwargs)
result = future.get(timeout=None)
```

Submit a workflow from the command line

Submit and wait for the results

```
ewoks submit results/workflow2.json --outputs=end --wait inf
```

Submit without waiting

```
ewoks submit results/workflow2.json

Workflow 'results/workflow2.json' submitted (ID: d3d20d06-20da-4861-
8ddf-6cf70c0bab0b)
```

Retrieve the result from the job ID in Python

```
from ewoksjob.client import get_future

future = get_future('d3d20d06-20da-4861-8ddf-6cf70c0bab0b')
print(future.get())
{'task6': {'result': 16}}
```

✏ Exercise: submit a workflow remotely

Documentation: https://ewoksjob.readthedocs.io/

```
export EWOKS_CONFIG_URI=results.celeryconfig_sql  # Linux and macOS
set EWOKS_CONFIG_URI=results.celeryconfig_sql  # Windows
```

From Python

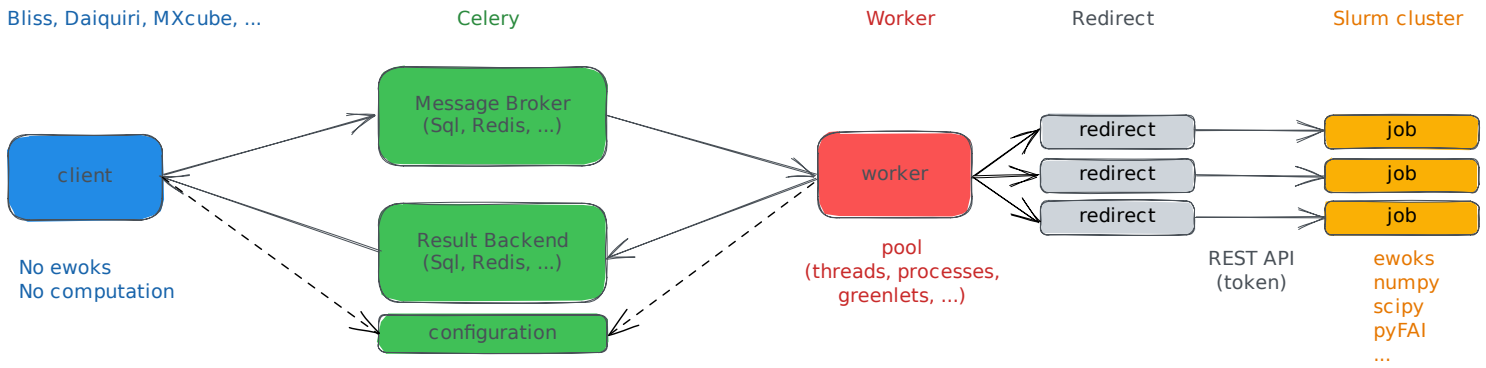
```
from ewoksjob.client import submit

future = submit(args=args, kwargs=kwargs)
result = future.get(timeout=None)
```

From the command line

```
ewoks submit results/workflow2.json --wait inf
```

Slurm execution

Supported through `ewoksjob` with `pyslurmutils` as a way to start Slurm jobs and receive results.



Requires an extra dependency **in the worker environment**

```
pip install "ewoksjob[sql,redis,worker,slurm]"
```

Create a Slurm access token

```
ssh myname@rnice

scontrol token lifespan=86400

SLURM_JWT=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE2Nzk1NjU5OD
UsImlhdCI6MTY3OTU2NTk4NCwic3VuIjoiZGVub2xmIn0.s9uGkTPg9xNO21LG5a10fdq
vfQfnfVZ4ad58bVYncrg
```

Start a Slurm worker

You can start a Slurm worker on any computer on the **intranet**

```
ewoksjob --config=results.celeryconfig_sql worker --pool=slurm \
 --slurm-url=http://... \
 --slurm-user=myname \
 --slurm-token=eyJhbGciOiJIUzI1NiIs...
```

ewoksjob uses the *pyslurmutils* library for the communication with Slurm (REST API).

Environment variables can be used instead of command line arguments

```
export EWOKS_CONFIG_URI=results.celeryconfig_sql
export SLURM_URL=http://...
export SLURM_USER=myname
export SLURM_TOKEN=eyJhbGciOiJIUzI1NiIs...

ewoksjob worker --pool=slurm
```

On Slurm or Visa (https://visa.esrf.fr/) you can activate an ewoks client or worker environment with

```
module load ewoks
```

This environment also provides a shell function to request a Slurm token

```
slurm-token-env
```

Slurm job parameters can be provided as worker arguments with `-sp`

```
ewoksjob worker --pool=slurm \
 --slurm-pre-script="module load ewoks" \
 -sp time_limit=240 \
 -sp current_working_directory=/home/esrf/${USER}/ewoksfordevs
```

pyslurmutils documentation:

https://pyslurmutils.readthedocs.io/en/latest/#job-parameters

Slurm REST documentation:

https://slurm.schedmd.com/rest_api.html#v0.0.37_job_properties

Submit a workflow to Slurm

From Python

```python
from ewoksjob.client import submit

# Optionally override the worker's slurm parameters set with `-sp`
wd = "/home/esrf/myname/ewoksfordevs"
kwargs["_slurm_spawn_arguments"] = {
    "parameters": {
        "time_limit": 240,
        "current_working_directory": wd,
    },
    "pre_script": "module load ewoks",
}

# If the workflow is defined on the client side
# send it as a python dictionary
future = submit(args=("results/workflow1.json",), kwargs=kwargs)
result = future.get(timeout=None)
```

From the command line

```
ewoks submit results/workflow1.json --wait inf
```

The path `results/workflow1.json` must exist in the `current_working_directory` on Slurm.

All ewoks tasks of *workflow1* must be available in the loaded Python environment on Slurm loaded by the `--slurm-pre-script` command.

The *pyslurmutils* library comes with a CLI. It is possible for example to monitor SLURM jobs:

```
pyslurmutils status -m 1 --url=... --user=... --token=...
```

The environment variables *SLURM_URL*, *SLURM_USER* and *SLURM_TOKEN* can also be used here instead of the command line arguments.

Documentation: https://pyslurmutils.readthedocs.io

✏ Exercise: submit a workflow to Slurm

Documentation: https://ewoksjob.readthedocs.io/

Environment variable for client and worker

```
export EWOKS_CONFIG_URI=results.celeryconfig_sql
```

Environment variable for the worker

```
export SLURM_URL=http://...
export SLURM_USER=myname
export SLURM_TOKEN=eyJhbGciOiJIUzI1NiIs...
```

Start the worker (needs *EWOKS_CONFIG_URI* and slurm variables)

```
ewoksjob worker --pool=slurm \
  --slurm-pre-script="module load ewoks" \
  --slurm-log-directory=/home/esrf/${USER}/slurm/ \
  -sp current_working_directory=/home/esrf/${USER}/ewoksfordevs \
  -sp time_limit=10
```

Worker parameters:

- `--slurm-pre-script`: python environment with ewoks and task dependencies
- `--slurm-log-directory`: in case you want slurm logs during execution (cleaned up afterwards)
- `-sp current_working_directory`: to find workflow files and local tasks
- `-sp time_limit`: job wall time

Submit workflow from the client (needs *EWOKS_CONFIG_URI*)

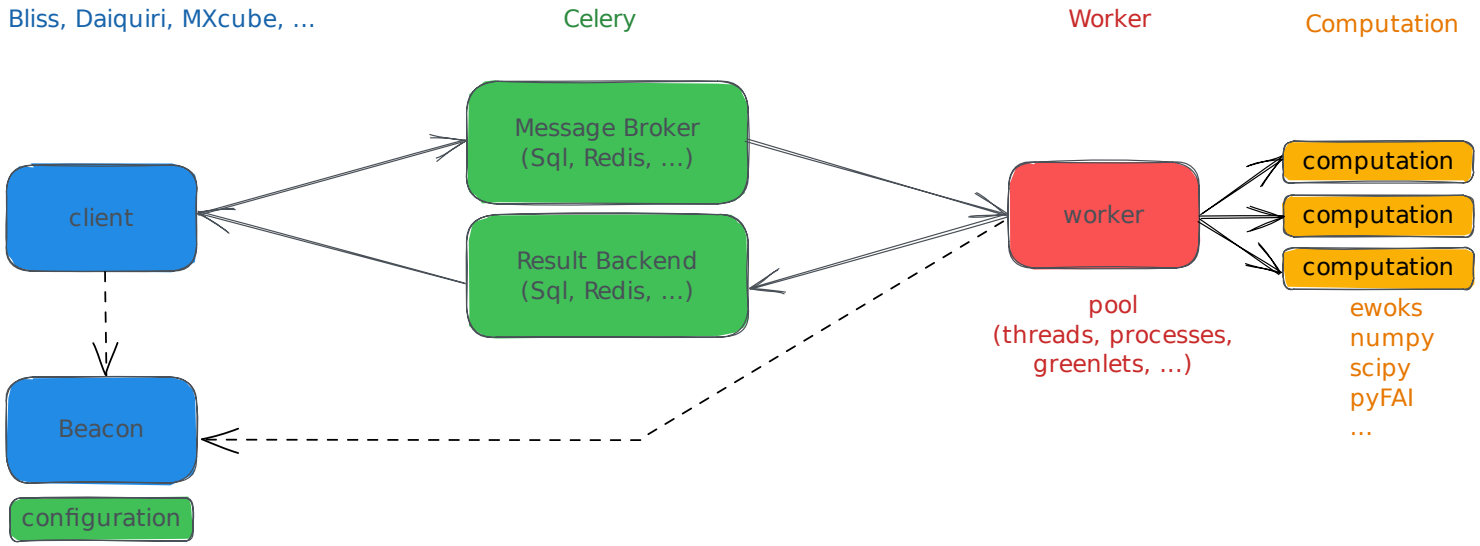
```
ewoks submit results/workflow1.json --wait inf
```

Optionally, monitor slurm jobs (needs slurm variables)

```
pyslurmutils status -m 1
```

Online data processing in Bliss

In Bliss, the Celery configuration is managed by the *Beacon* server



Celery configuration

We could directly specify the URI to the Celery configuration

```
export EWOKS_CONFIG_URI=beacon://<host>:<port>/ewoks/config.yml
```

But, it is enough the specify the URL of the Beacon host

```
export BEACON_HOST=<host>:<port>
```

⚠ *Warning: make sure `EWOKS_CONFIG_URI` is not defined because it has priority over `BEACON_HOST`.*

Workflow for online and offline processing

We will create a workflow that can run both online (during the scan) and offline (after the experiment).

Workflow for online and offline processing

We will create a workflow that can run both online (during the scan) and offline (after the experiment).

Data processing: azimuthal integration of X-ray Powder Diffraction data (XRPD).

Workflow for online and offline processing

We will create a workflow that can run both online (during the scan) and offline (after the experiment).

Data processing: azimuthal integration of X-ray Powder Diffraction data (XRPD).

The tutorial resources provide a simplified integration task

`results.xrpd.PyFaiIntegrate`

and a workflow

`results/workflow3.json`

which takes the URL to a Bliss scan and saves the integrated diffraction patterns in the official *PROCESSED_DATA* directory of the proposal.

Workflow for online and offline processing

We will create a workflow that can run both online (during the scan) and offline (after the experiment).

Data processing: azimuthal integration of X-ray Powder Diffraction data (XRPD).

The tutorial resources provide a simplified integration task

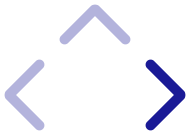
`results.xrpd.PyFaiIntegrate`

and a workflow

`results/workflow3.json`

which takes the URL to a Bliss scan and saves the integrated diffraction patterns in the official *PROCESSED_DATA* directory of the proposal.

A proper implementation of diffraction tasks can be found in *ewoksxrpd*.



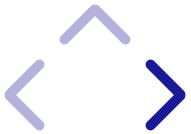
Defining an ewoks task that works during and after the scan requires taking care of some aspects

- diffraction patterns need to be fetched during the acquisition (not simply reading from a file)
- when saving results in HDF5, it can happen that multiple workflows try to write to the same file at the same time
- when saving results in HDF5 chunking, compression and write optimization is desirable
- where do I save results?

Defining an ewoks task that works during and after the scan requires taking care of some aspects

- diffraction patterns need to be fetched during the acquisition (not simply reading from a file)
- when saving results in HDF5, it can happen that multiple workflows try to write to the same file at the same time
- when saving results in HDF5 chunking, compression and write optimization is desirable
- where do I save results?

Here we highlight the different solutions used in `results.xrpd.PyFaiIntegrate`



Read Bliss data during or after the scan with `blissdata`.

```
from blissdata.h5api import dynamic_hdf5

with dynamic_hdf5.File("/path/to/file.h5") as f:
    ...
```

The goal of the `blissdata.h5api` is to read the data as if you read it from a static HDF5 file with `h5py`.

https://bliss.gitlab-pages.esrf.fr/bliss/master/data/h5py_like_api.html

Deal with concurrent writing from parallel workflows

`silx` provides a utility that yields an HDF5 group when it becomes available. This effectively serializes concurrent writing from different workflows.

```
from silx.io import h5py_utils

with h5py_utils.open_item("/path/to/result.h5", "/", mode="a") as f:
    ...
```

Optimize saving in HDF5

`ewoksdata` provides a class which takes care of HDF5 write optimization (chunking size, compression, write aligned with the chunks).

```
from ewoksdata.data.hdf5.dataset_writer import DatasetWriter

with DatasetWriter(group, "intensity") as writer:
    for data in ...:
        writer.add_point(data)
```

Build the saving path in a robust way

There is an official location to save results from online data processing:

```
/data/visitor/ch6562/id31/20230309/PROCESSED_DATA
```

Never compose this path directly as it might change in the future. Derive it from a dataset filename with `blissoda`:

```
from blissoda.utils.directories import get_processed_dir

processed_dir = get_processed_dir(dataset_filename)
```

✏ Exercise: process a live XRPD scan

Start a Bliss demo environment and shell on https://visa.esrf.fr/ (make sure the machine has at least 16GB of memory)

✏ Exercise: process a live XRPD scan

Start a Bliss demo environment and shell on https://visa.esrf.fr/ (make sure the machine has at least 16GB of memory)

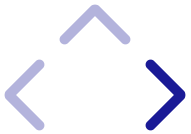
Create a worker environment on VISA

```
python3 -m venv worker_env
source worker_env/bin/activate
pip install "ewoksjob[blissworker]" pyfai blissdata \
            blissoda ewoksdata silx
```

Start a worker on VISA (make sure it can import `results.xrpd.PyFaiIntegrate`)

```
export BEACON_HOST="localhost:10001"
ewoksjob worker -Q myworker -n myworker@$(hostname)
```

The `-Q` (for `queues`) option will allow clients to submit workflow to this specific worker. The `-n` option is a unique identifier in case there are several workers running on the same host.



Create an ewoks client environment on VISA

```
python3 -m venv client_env
source client_env/bin/activate
pip install "ewoksjob[beacon,redis]" ewoks
```

Start the processing before, during or after starting the scan

```
export BEACON_HOST="localhost:10001"
ewoks submit results/workflow3.json \
    -c queue=myworker --wait inf --outputs=all \
    -p filename=/tmp/.../sample_0001.h5 \
    -p 'scan="1.1"'
```

Start the scan

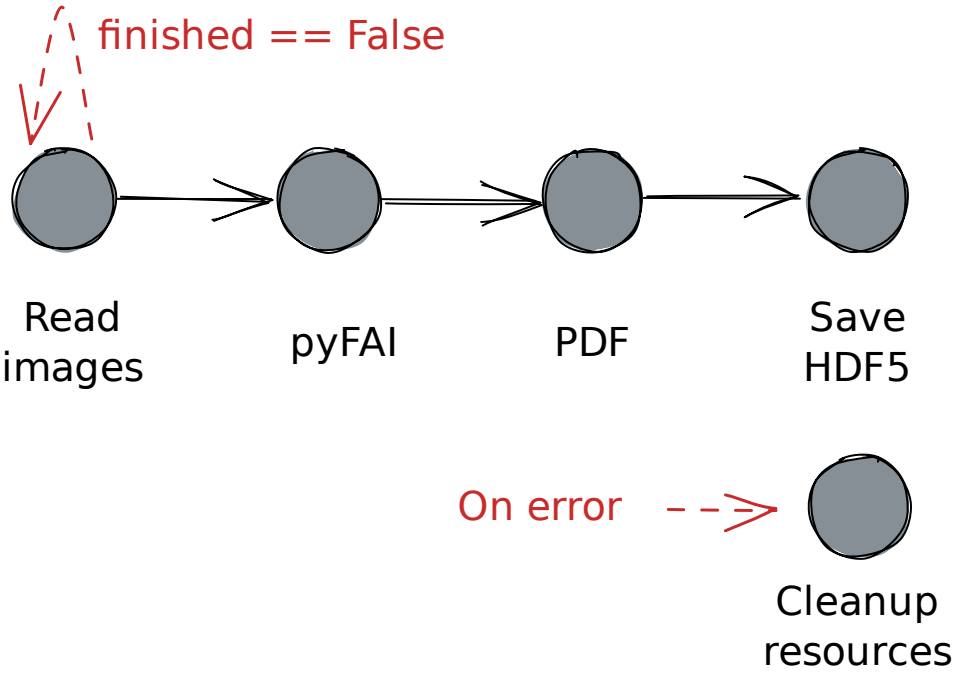
```
DEMO [1] loopscan(1000, 0.1, difflab6)
```

Part 4: Final remarks

Ewoks features not covered

Conditional links

For example: self-triggering workflow with a global error handler

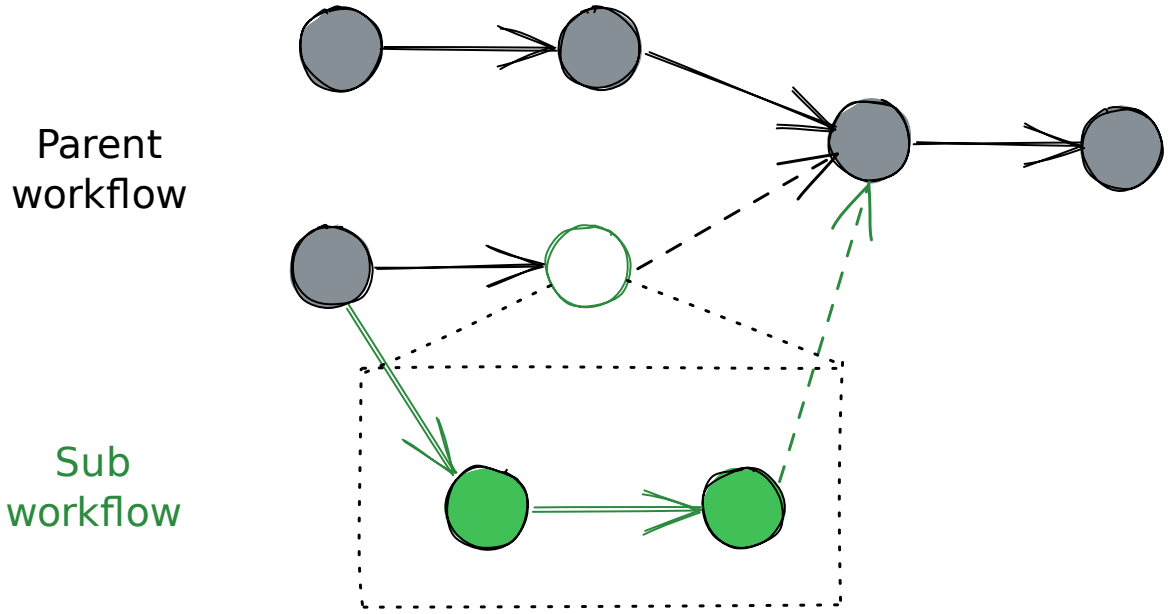


Not supported by the desktop GUI, web GUI only.

https://ewokscore.readthedocs.io/en/latest/definitions.html

Sub-Workflows as nodes in other workflows

Workflow as a node in another workflow



Links to/from workflow nodes have an additional *sub_target*/*sub_source* argument.

Not supported by the desktop GUI, web GUI only.

https://ewokscore.readthedocs.io/en/latest/definitions.html

Desktop GUI

To provide support for the Ewoks desktop GUI (*ewoks-canvas*) you need to create an *Orange add-on* (which is a Python project) in which you define a *Qt* widget for every Ewoks task class.

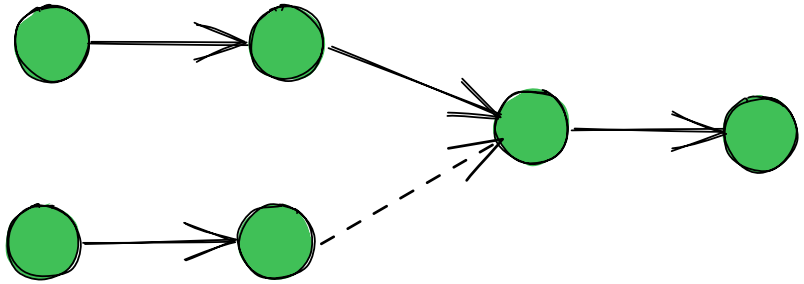
Documentation: https://ewoksorange.readthedocs.io

Task output persistence

For example: a workflow which includes calibration

filename = "/path/to/calib.h5"

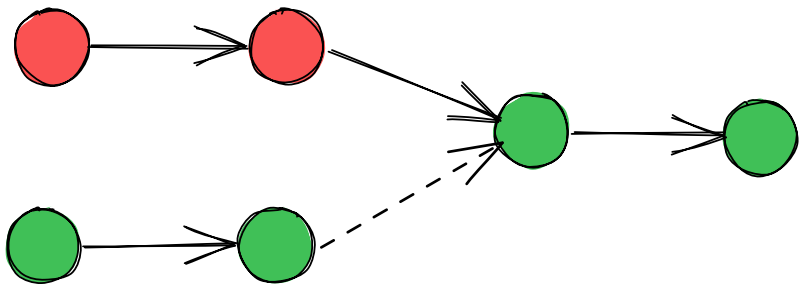
filename = "/path/to/dataset1.h5"



When persistence is enabled, the red tasks are not executed again because their inputs did not change.

filename = "/path/to/calib.h5"

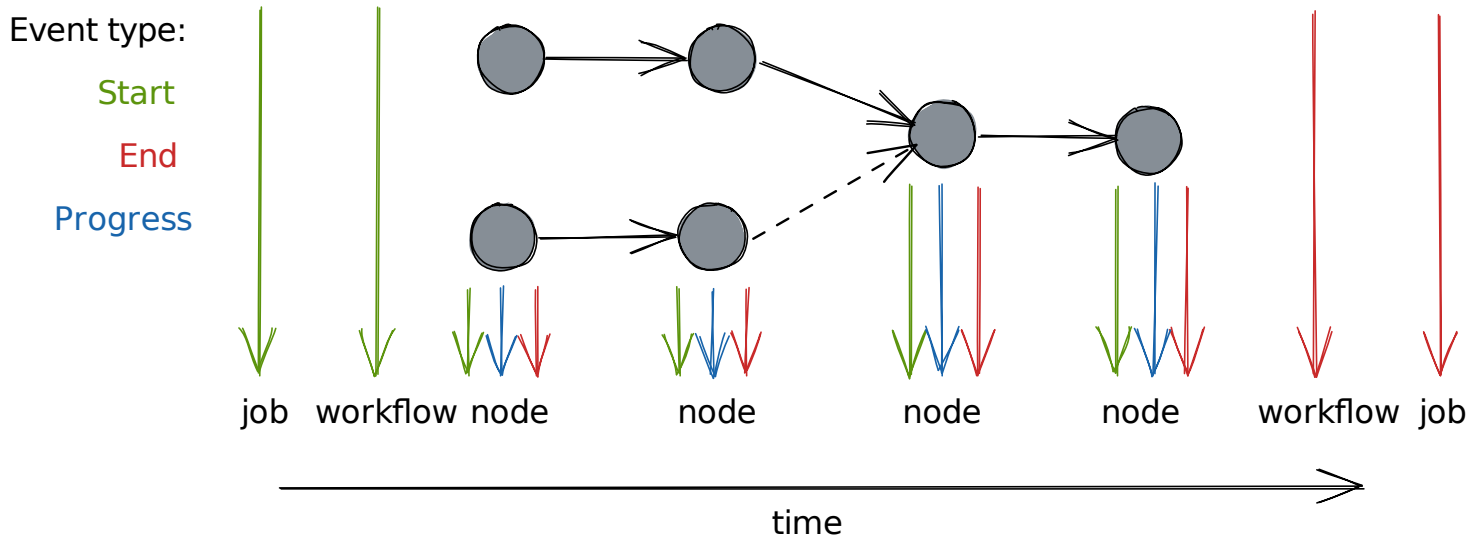
filename = "/path/to/dataset2.h5"



Persistence mechanism not solidified yet (current HDF5 or JSON files only).

In production at one beamline: XRPD at ID31 for STREAMLINE.

Ewoks events



Ewoks events are implemented with Python's logging facility.

```
ewoks execute results/workflow2.json -l info
```

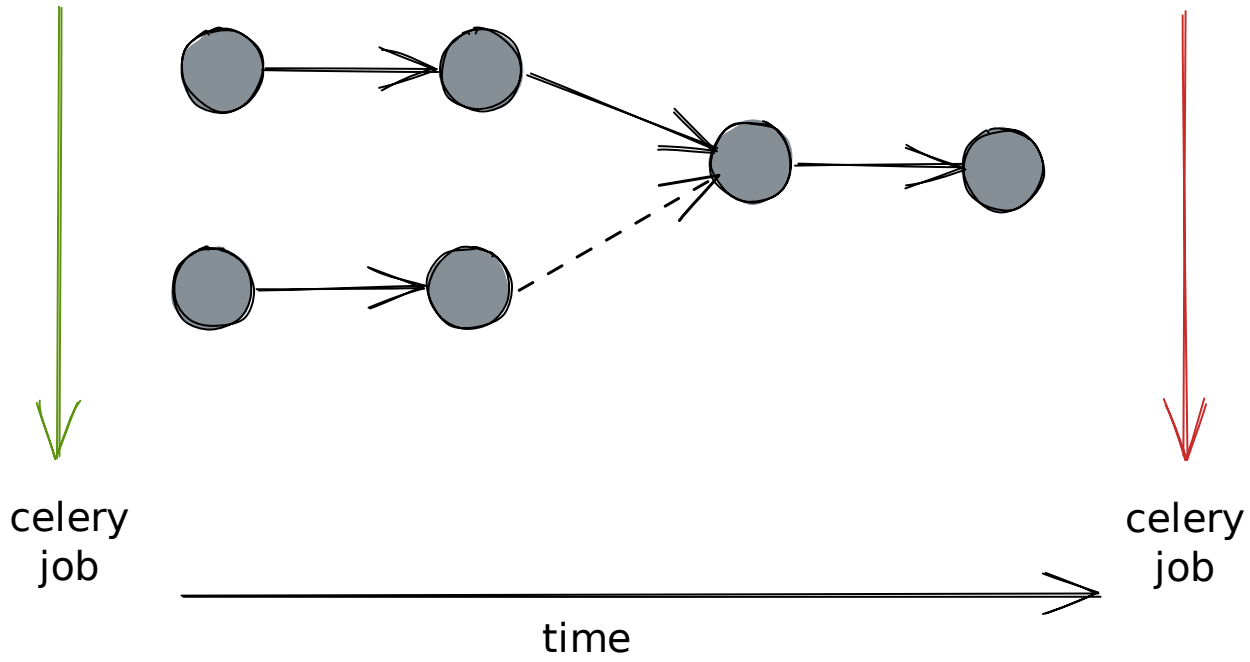
Event handlers can be created and registered.

Event content: https://ewoksutils.readthedocs.io

For example: node *end* events contain persistence URI's (if any).

Example: https://ewoks.readthedocs.io/en/latest/tutorials/ewoks_events.html

Celery worker events



Celery events on a higher level than Ewoks events.

Documentation: https://docs.celeryq.dev/en/latest/userguide/monitoring.html

Data portal integration

```python
from ewoks import execute_graph

result = execute_graph(..., upload_parameters=...)
```

Uses `IcatClient` from *pyicat-plus*.

Documentation: https://pyicat-plus.readthedocs.io

In production: XRPD at ID31 for STREAMLINE.

Data provenance

Save the workflow with the results.

```
from ewoks import execute_graph

execute_graph(..., convert_destination="/path/to/file.json")
```

In production: XRPD at ID31 for STREAMLINE.

A more systematic way is currently missing. Perhaps it could be done in conjuction with task persistence or ewoks events?

Deploy software on Slurm

Automated with gitlab: https://gitlab.esrf.fr/apptainer

Make sure the environment has Ewoks installed with all Ewoks task libraries you need.

Documentation: https://apptainer.gitlab-pages.esrf.fr/admin/templates/

Ewoks Tutorial For Developers

Questions ?