ABINIT School 2024 "Towards ab initio simulations on exascale supercomputers"

Bruyères-le-Châtel (France), January 29 – February 2 2024

Workflows in ABINIT with

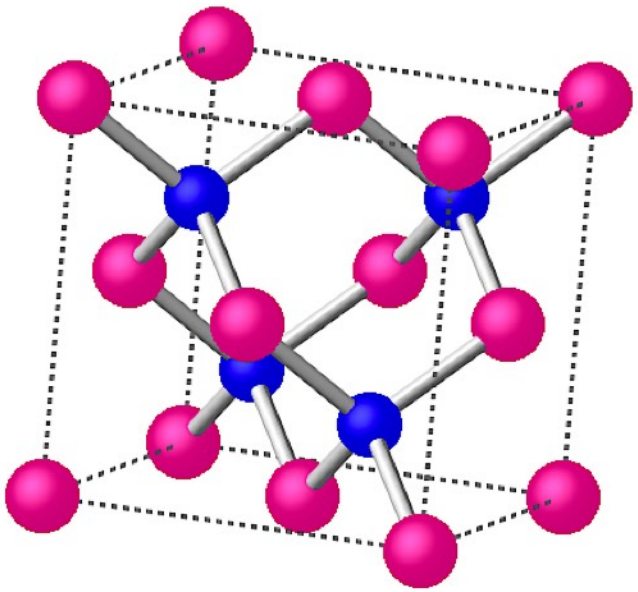
Jobflow and atomate

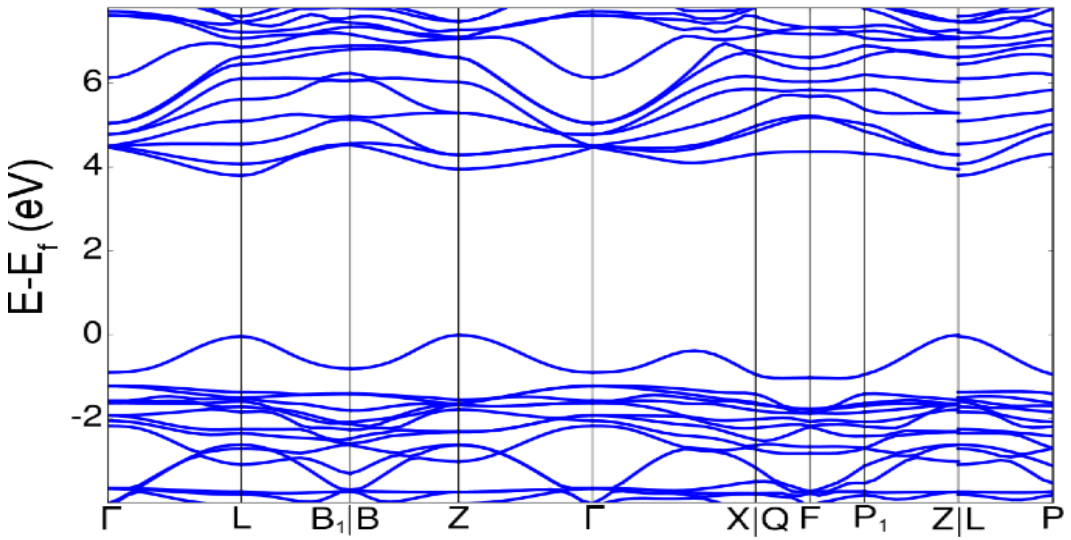
G.-M. Rignanese (UCLouvain)

Objective

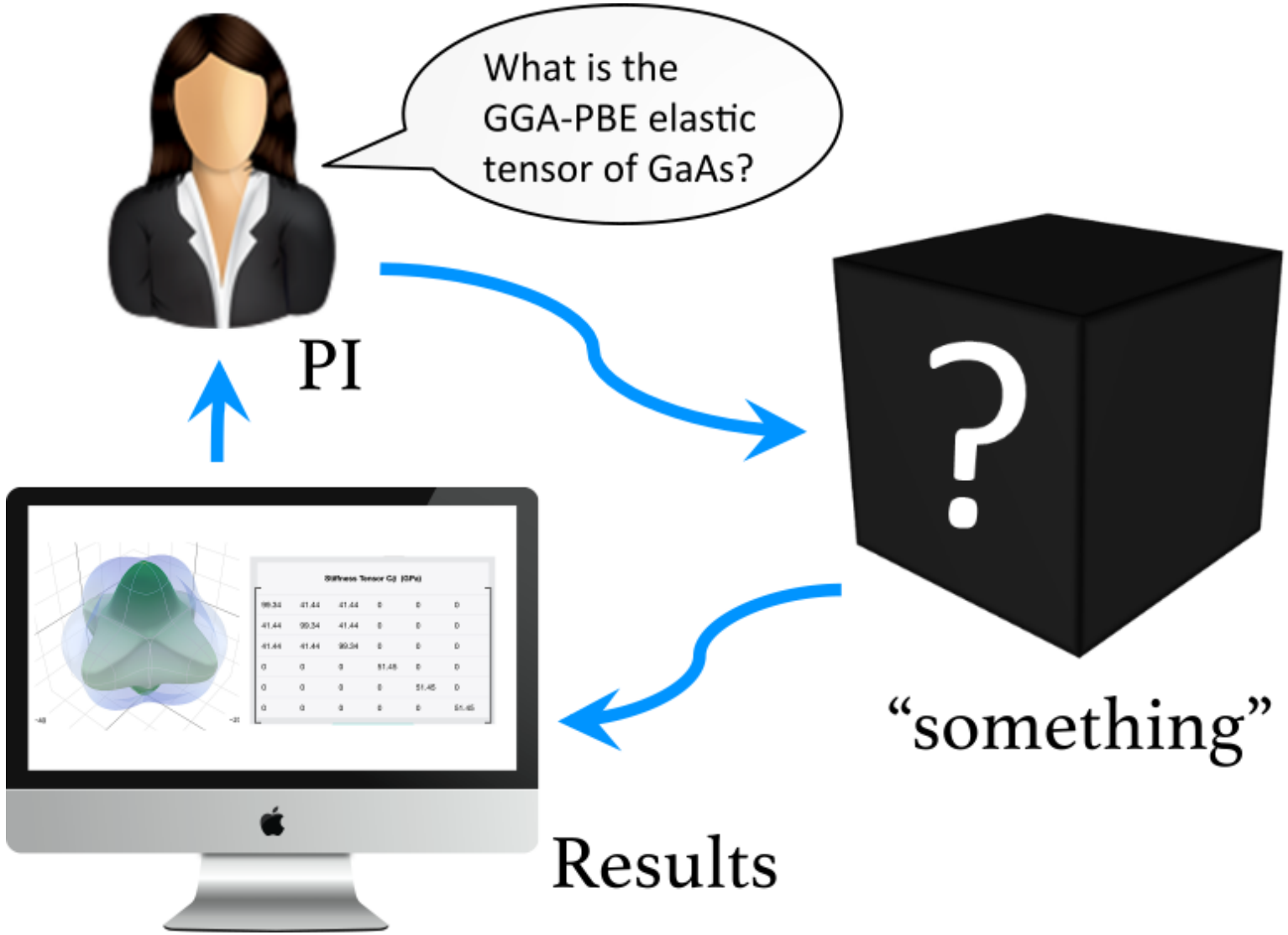
- easily compute a property for a (complex) material from its structure

  → **automatized calculations**

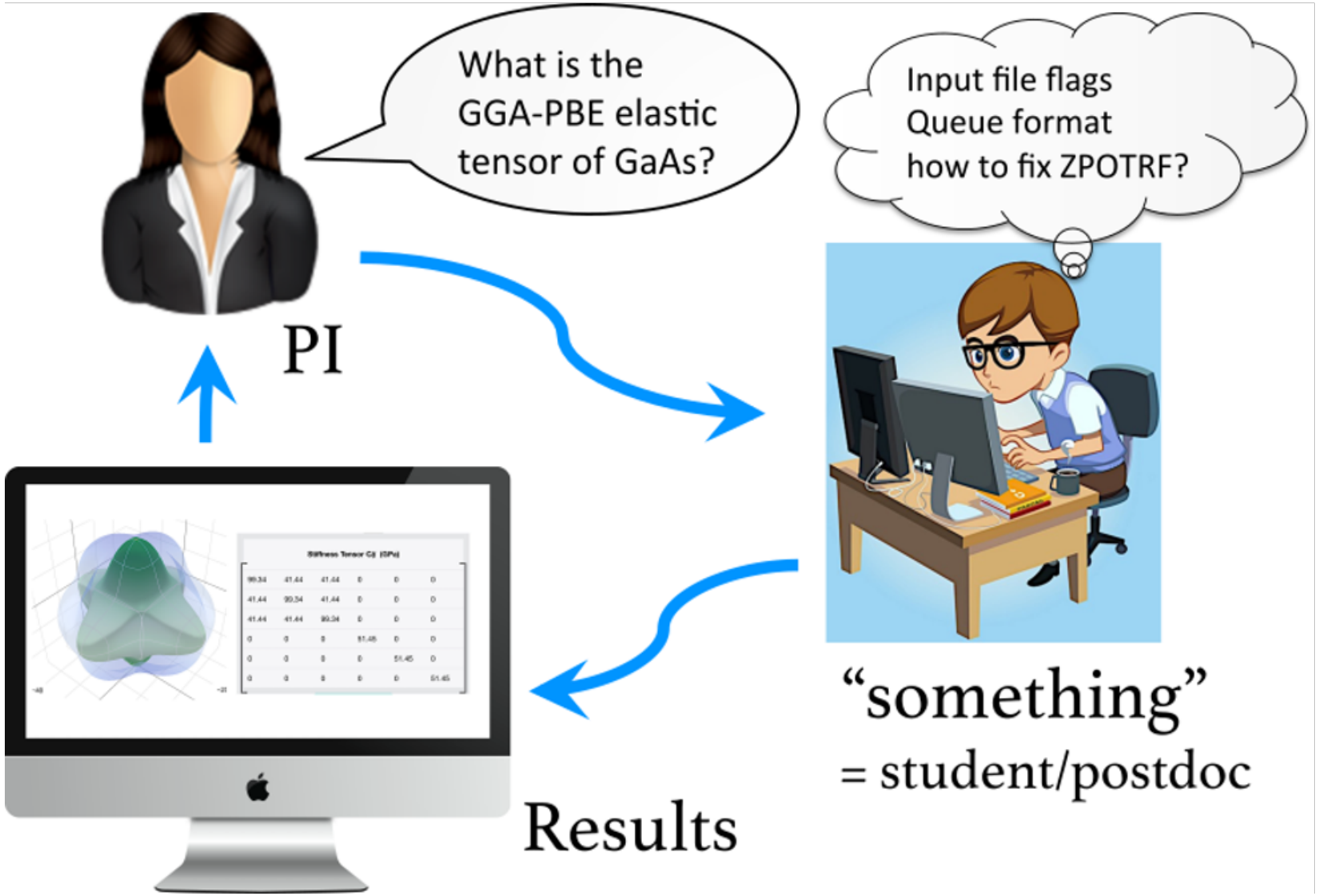




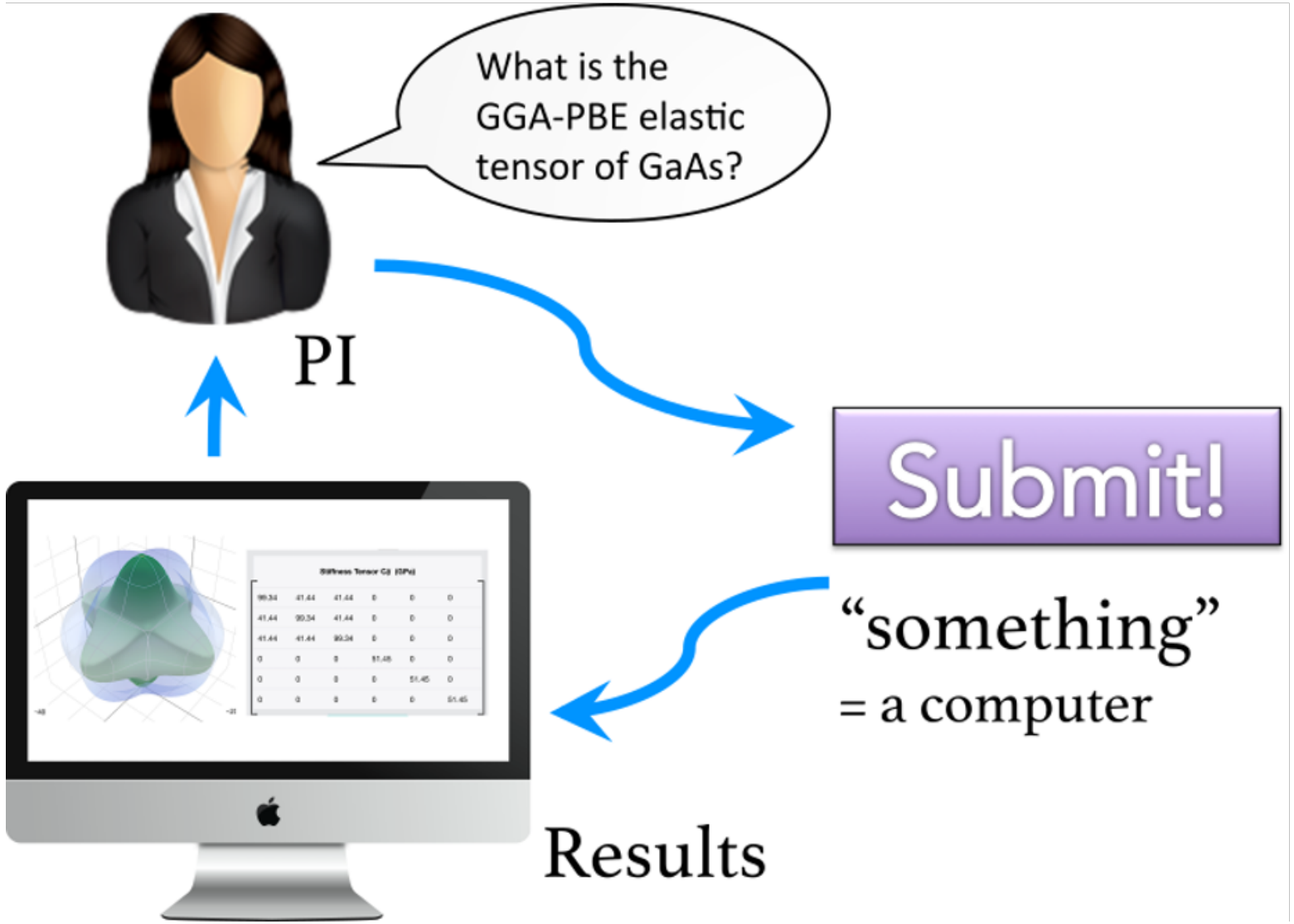
Automated calculations



Automated calculations



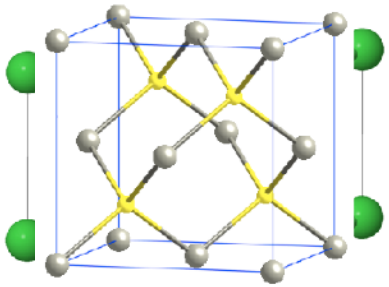
Automated calculations



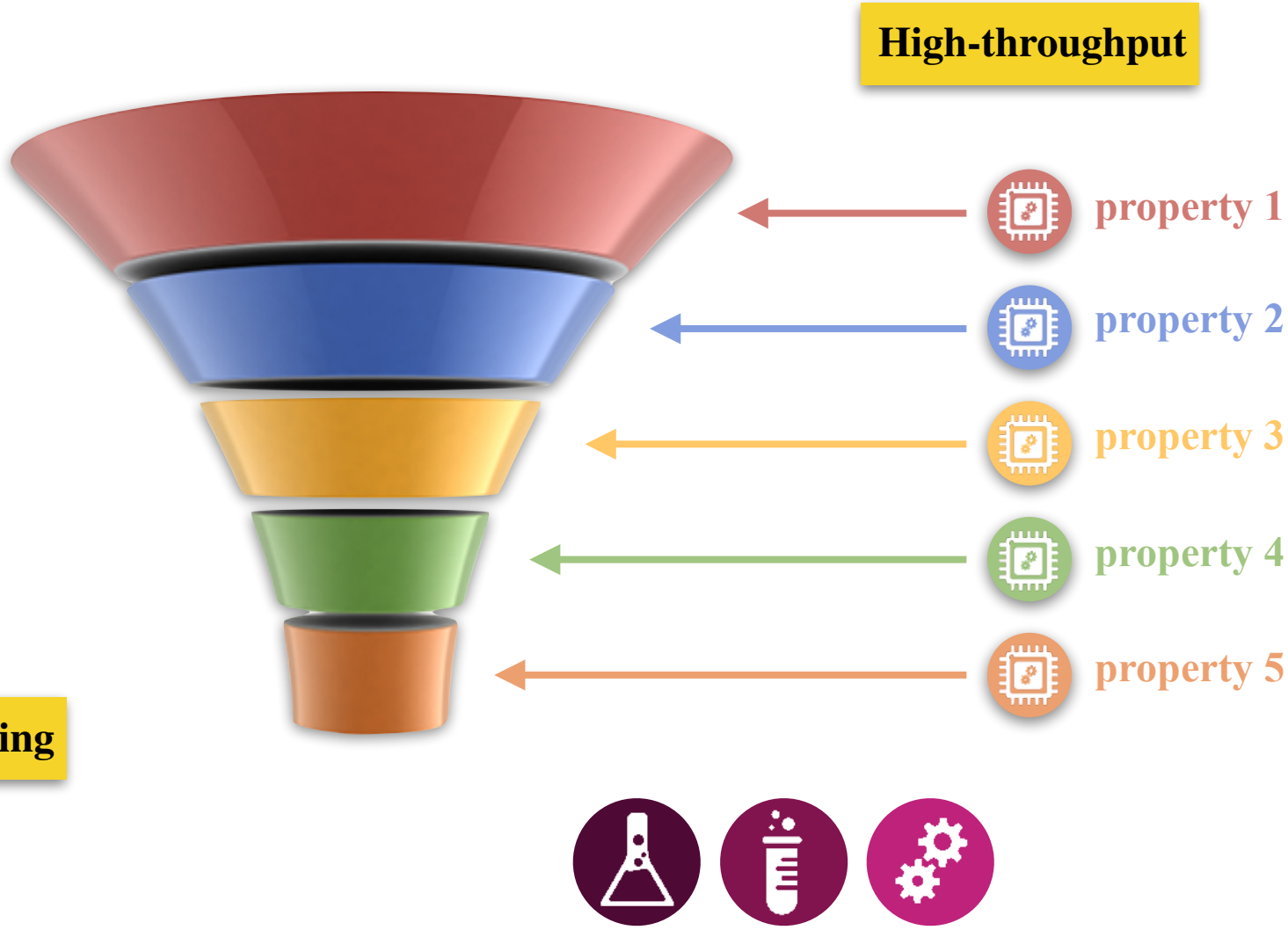
Automated calculations

- **High volume** of calculated properties and materials through an **automatized process** → **high-throughput calculations**
- Several steps involved:
  - Input generation
  - Code execution
  - Error correction
  - Data storage
  - Data analysis

High-throughput *ab initio* screening

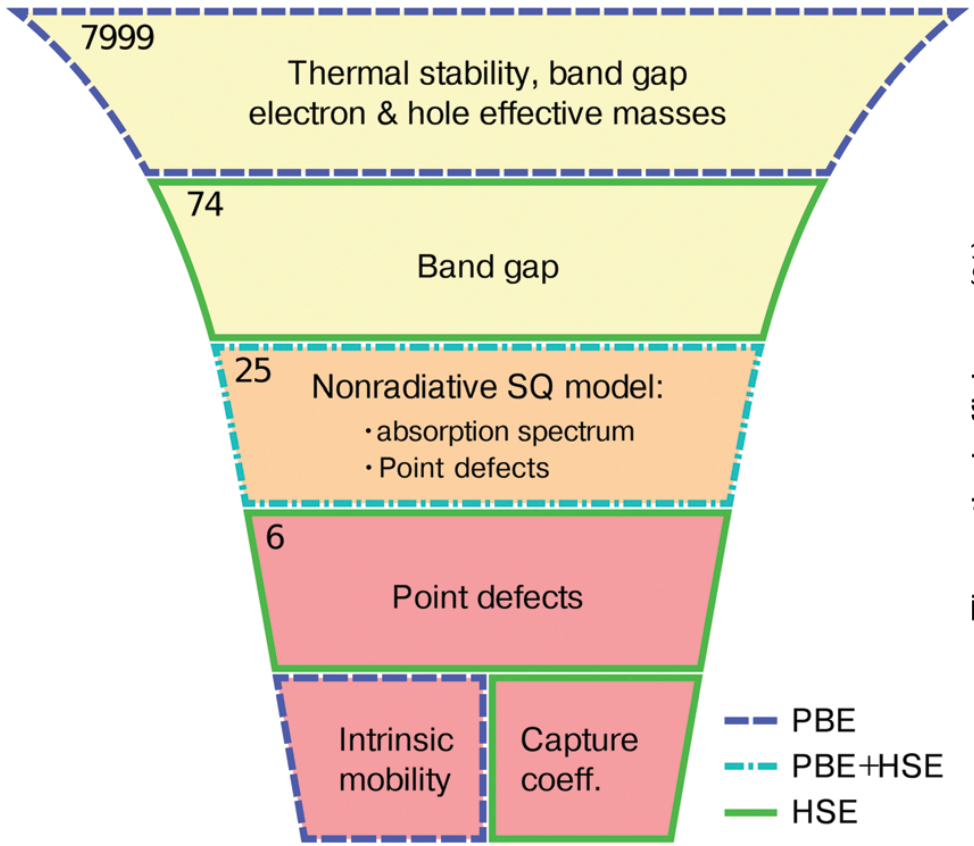


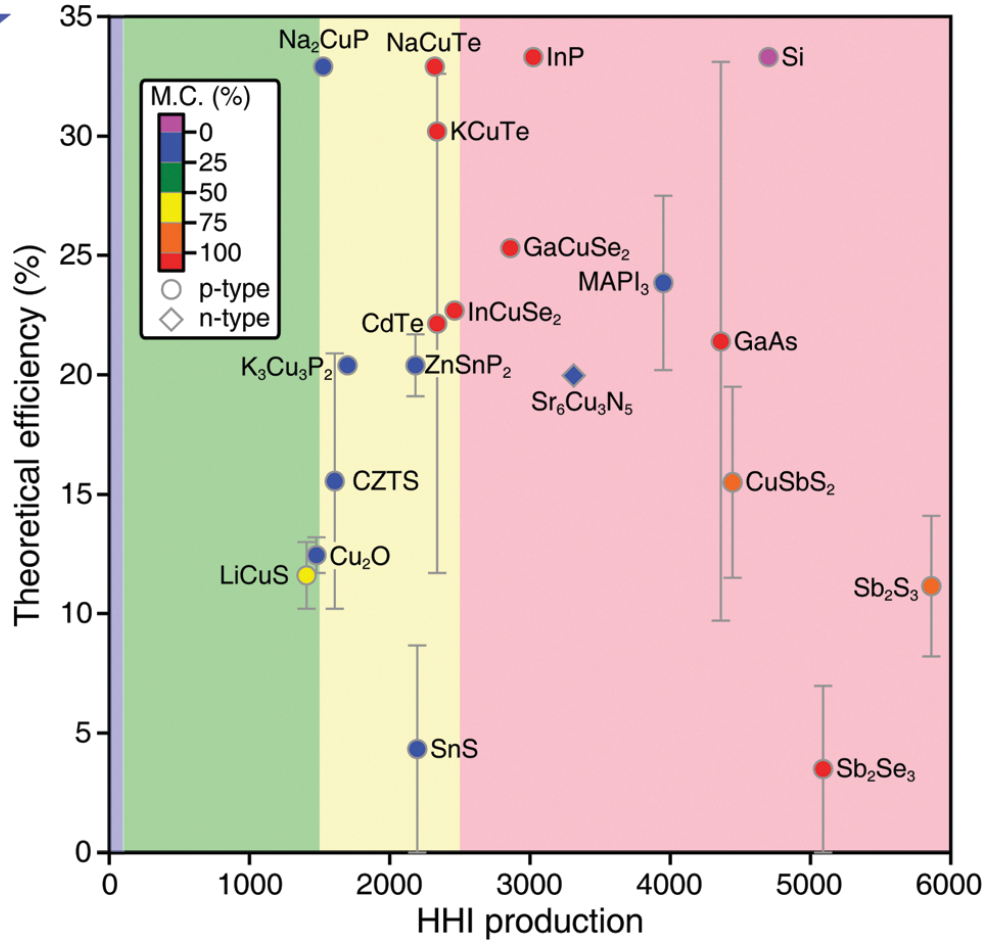
Consider as many compounds as possible, typically $O(10^3) \rightarrow O(10^5)$



$O(10^1) \rightarrow O(10^2)$ compounds

High-throughput *ab initio* screening





KCuTe, NaCuTe, LiCuS,
Na_2CuP, $K_3Cu_3P_2$, $Sr_6Cu_3N_5$

[D. Dahliah *et al.,* Energy Environ. Sci.**14**, 5057 (2021)]

Common workflows

npj | Computational Materials

www.nature.com/npjcompumats

ARTICLE OPEN

Check for updates

Common workflows for computing material properties using different quantum engines

Sebastiaan P. Huber[1✉], Emanuele Bosoni[2], Marnik Bercx[1], Jens Bröder[3,4], Augustin Degomme[5], Vladimir Dikan[2], Kristjan Eimre[6], Espen Flage-Larsen[7,8], Alberto Garcia[2], Luigi Genovese[5], Dominik Gresch[9], Conrad Johnston[10], Guido Petretto[11], Samuel Poncé[1], Gian-Marco Rignanese[11], Christopher J. Sewell[1], Berend Smit[12], Vasily Tseplyaev[3,4], Martin Uhrin[1], Daniel Wortmann[3], Aliaksandr V. Yakutovich[1,12], Austin Zadoks[1], Pezhman Zarabadi-Poor[13,14], Bonan Zhu[14,15], Nicola Marzari[1] and Giovanni Pizzi[1✉]

The prediction of material properties based on density-functional theory has become routinely common, thanks, in part, to the steady increase in the number and robustness of available simulation packages. This plurality of codes and methods is both a boon and a burden. While providing great opportunities for cross-verification, these packages adopt different methods, algorithms, and paradigms, making it challenging to choose, master, and efficiently use them. We demonstrate how developing common interfaces for workflows that automatically compute material properties greatly simplifies interoperability and cross-verification. We introduce design rules for reusable, code-agnostic, workflow interfaces to compute well-defined material properties, which we implement for eleven quantum engines and use to compute various material properties. Each implementation encodes carefully selected simulation parameters and workflow logic, making the implementer's expertise of the quantum engine directly available to non-experts. All workflows are made available as open-source and full reproducibility of the workflows is guaranteed through the use of the AiiDA infrastructure.

npj Computational Materials (2021)7:136; https://doi.org/10.1038/s41524-021-00594-6

Common workflows

nature reviews physics

https://doi.org/10.1038/s42254-023-00655-3

Expert recommendation

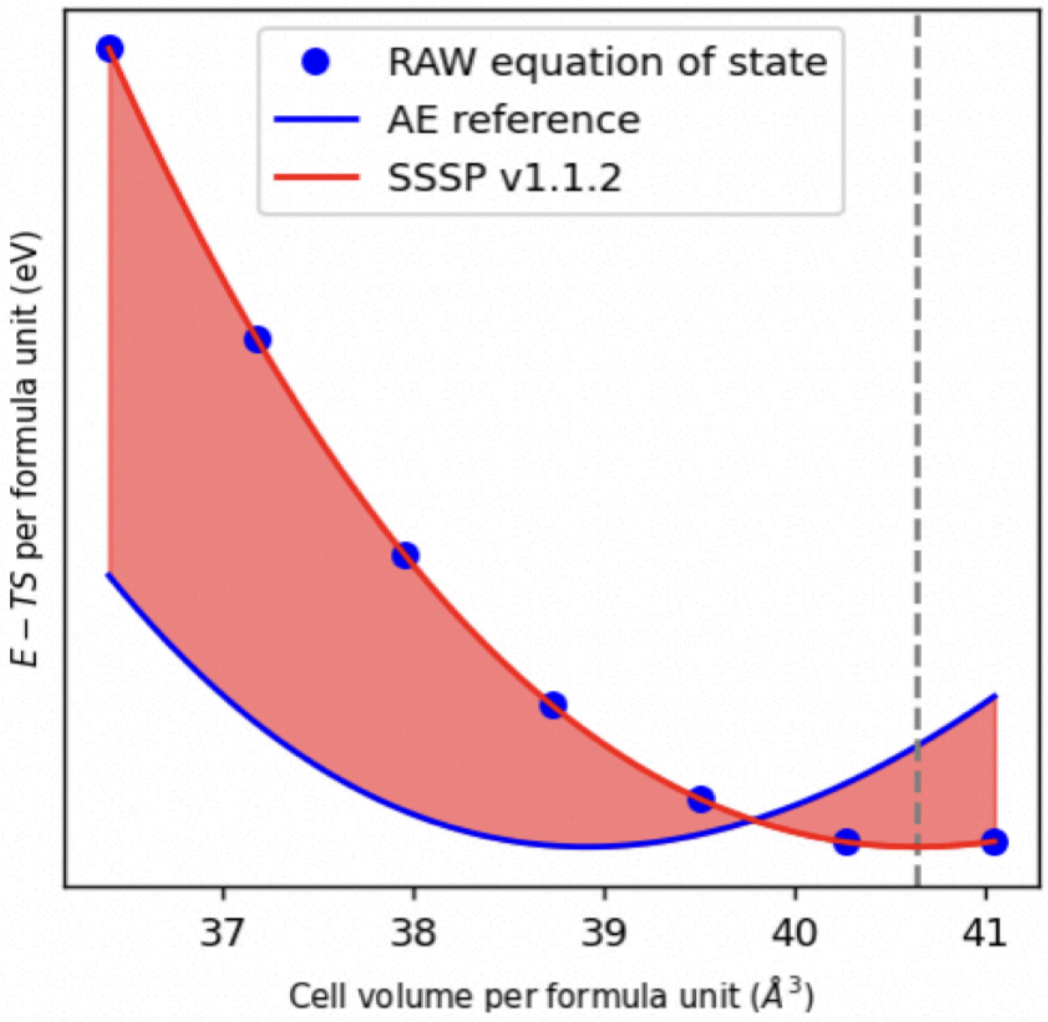
Check for updates

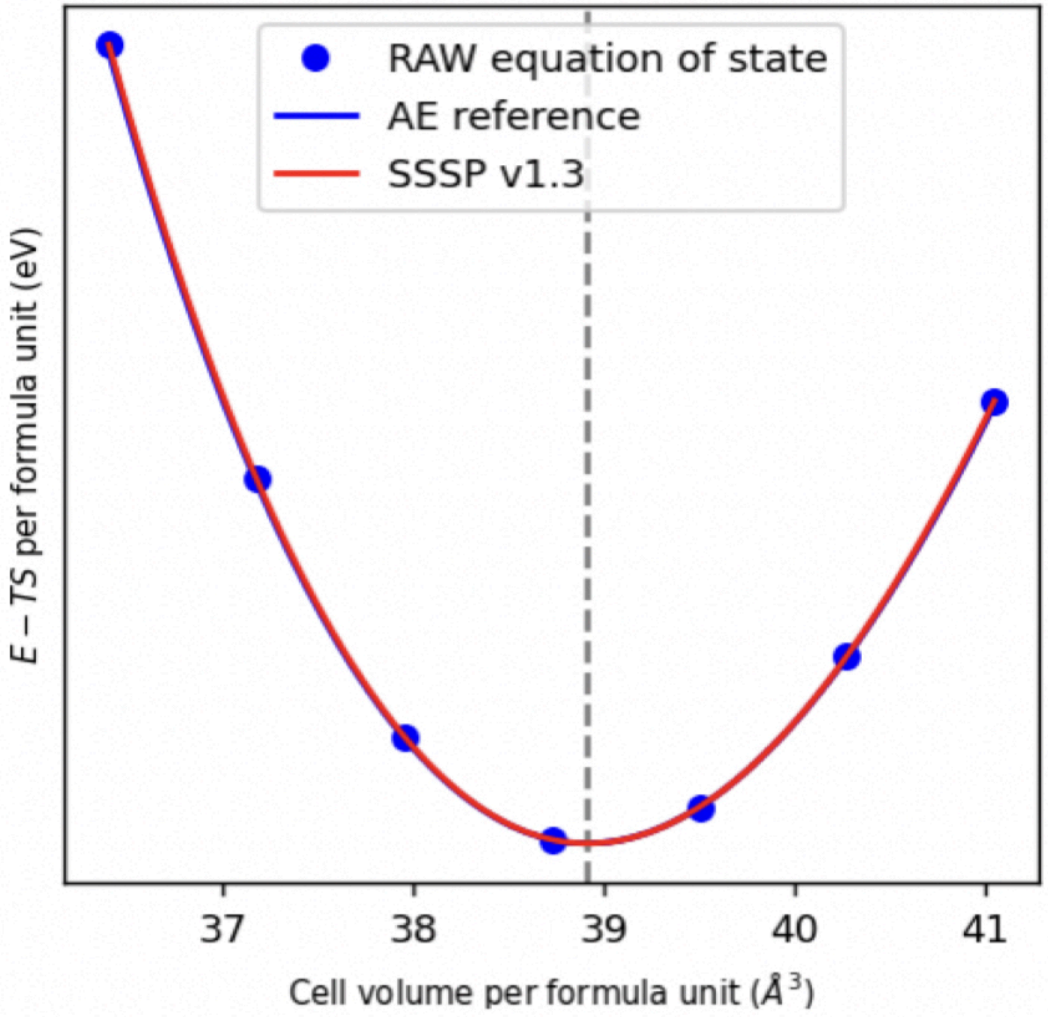
How to verify the precision of density-functional-theory implementations via reproducible and universal workflows

Emanuele Bosoni[1], Louis Beal[2], Marnik Bercx[3], Peter Blaha[4], Stefan Blügel[5], Jens Bröder[5,6], Martin Callsen[7,8,9], Stefaan Cottenier[7,8], Augustin Degomme[2], Vladimir Dikan[1], Kristjan Eimre[3], Espen Flage-Larsen[10,11], Marco Fornari[12], Alberto Garcia[1], Luigi Genovese[2], Matteo Giantomassi[13], Sebastiaan P. Huber[3,14], Henning Janssen[5], Georg Kastlunger[15], Matthias Krack[16], Georg Kresse[17,18], Thomas D. Kühne[19,20], Kurt Lejaeghere[8,21], Georg K. H. Madsen[4], Martijn Marsman[17,18], Nicola Marzari[3,16], Gregor Michalicek[5], Hossein Mirhosseini[22], Tiziano M. A. Müller[23], Guido Petretto[13], Chris J. Pickard[24,25], Samuel Poncé[13], Gian-Marco Rignanese[13], Oleg Rubel[26], Thomas Ruh[4,8,27], Michael Sluydts[7,8,28], Danny E. P. Vanpoucke[7,29], Sudarshan Vijay[15], Michael Wolloch[17,18], Daniel Wortmann[5], Aliaksandr V. Yakutovich[30], Jusong Yu[3,16], Austin Zadoks[3], Bonan Zhu[31,32] & Giovanni Pizzi[3,16]

[Nat. Rev. Phys. 6, 45 (2024)]

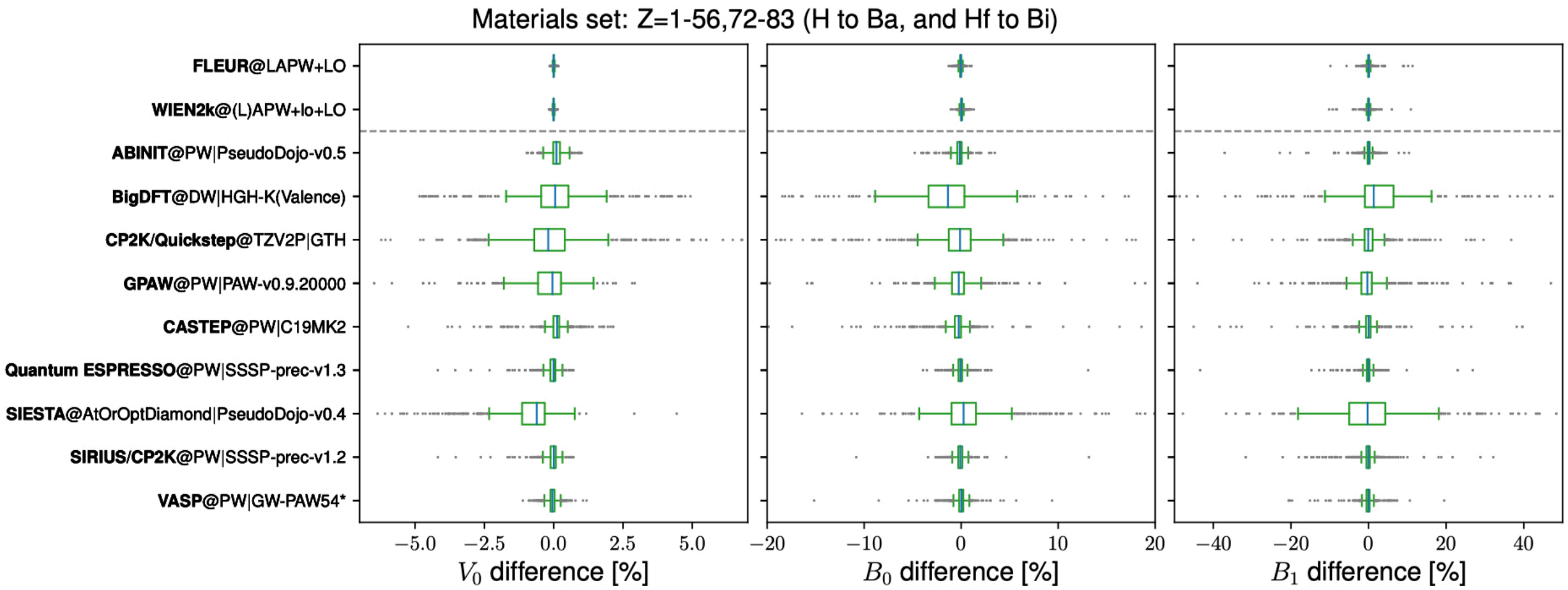
Common workflows





[Nat. Rev. Phys. 6, 45 (2024)]

Common workflows



[Nat. Rev. Phys. 6, 45 (2024)]

Automated calculations

- High-throughput ≠ magical calculation tool
  - No bug free code
  - Misspells
  - Default values cannot fit all the use cases
- Test the workflow on a small set before starting
  - Are the inputs correct?
  - Are the results stored correctly?
- Always check the results
  - Do they make sense?
- Clearly define your output
  - Can you find the result that you want?

Jobflow

[https://github.com/materialsproject/jobflow]

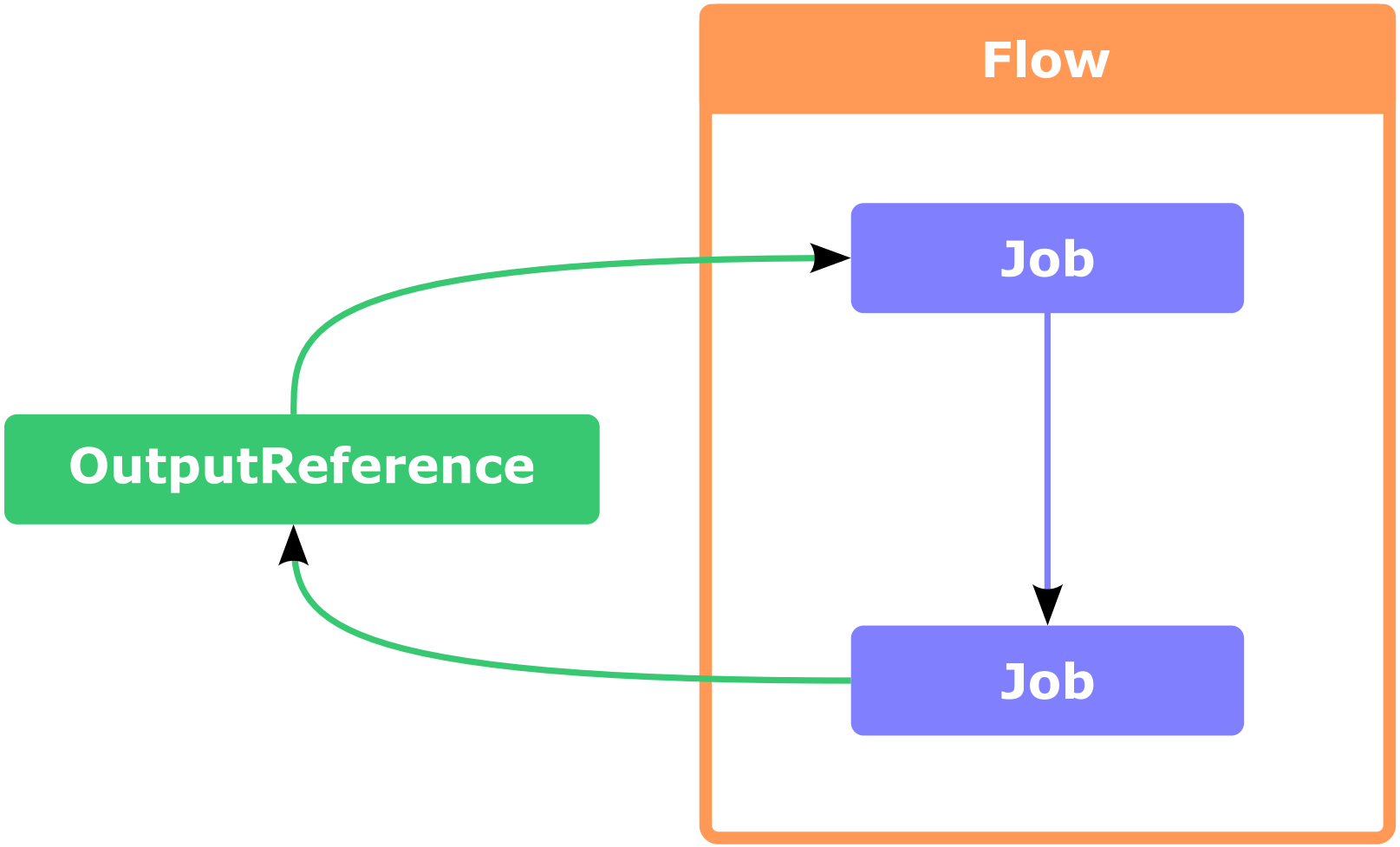
Introduction

Jobflow is a free, open-source library for writing and executing workflows. Complex workflows can be defined using simple python functions and executed locally or on arbitrary computing resources using the FireWorks workflow manager.

Some features that distinguish jobflow are dynamic workflows, easy compositing and connecting of workflows, and the ability to store workflow outputs across multiple databases.

- Clean and flexible Python API
- Integration with multiple databases (MongoDB, AWS S3, GridFS, …)
- Support for dynamic workflows
  - Additions
  - Restarts
- Can support different workflow managers (e.g. Fireworks)

Jobflow : Core concepts



Jobflow : Core concepts

Job

- Is an atomic computing job
- Any python function can be a Job
- Must return "serializable" outputs, i.e. outputs that can be easily stored in a database

Flow

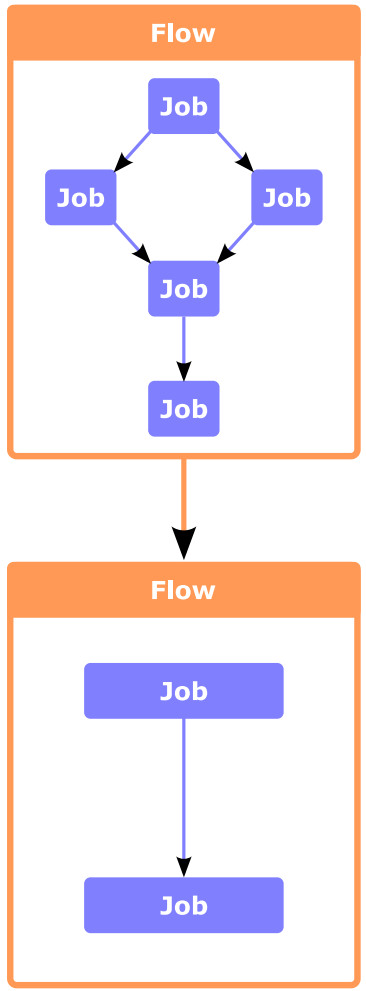
- Is a collection of Jobs or other Flows
- Connectivity between Jobs/Flows is automatically determined from the Jobs and Flows inputs
- Execution order automatically determined from the connectivity

OutputReference

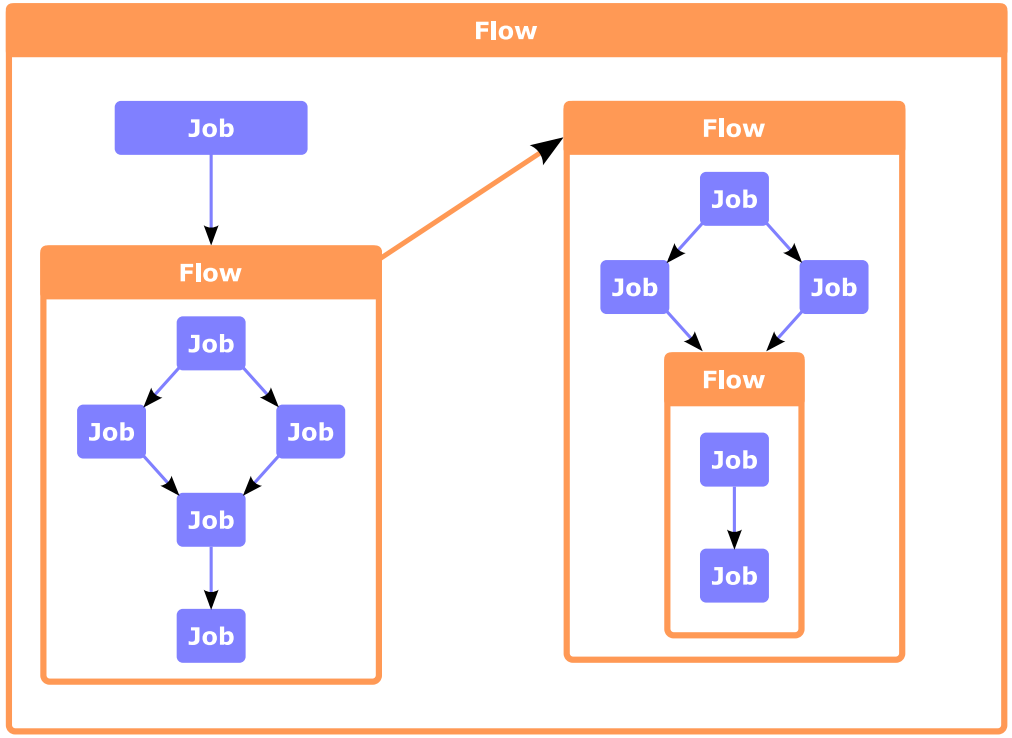
- **Deferred** output of a Job or a Flow
- Used to construct the workflow graph
- Also allows for more complex actions (e.g. dynamic workflows)

Jobflow : Connecting and composing

Connecting workflows

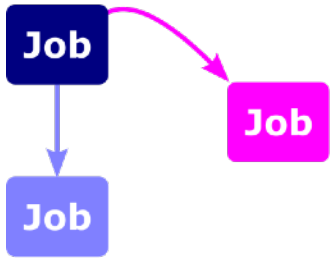


Composed workflows

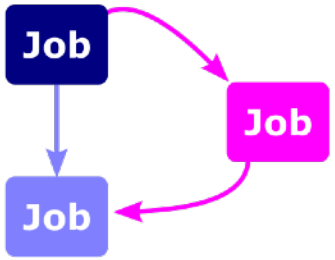


Jobflow : Dynamic workflows

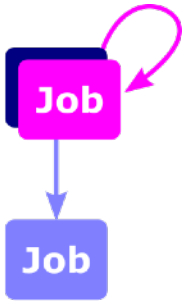
Addition



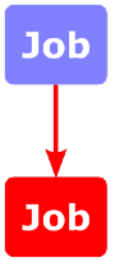
Detour



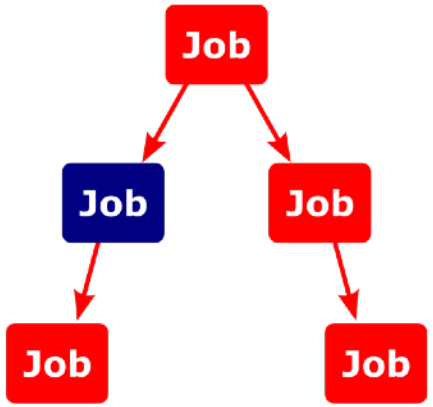
Replace



Stop children



Stop flow



Jobflow : Jobs

- **Jobs** are usually defined using the @job decorator
- Decorate the function or method that is supposed to become a Job
  - This modifies the actual function/method
  - The method now return a Job object
  - The execution is deferred
  - The function/method should contain what needs to be executed but will be executed later

```
from jobflow import job

@job
def add(a, b):
    return a + b
```

Jobflow : Jobs

- This modifies the actual function/method
  - The method now return a Job object
  - The execution is deferred
  - The function/method should contain what needs to be executed but will be executed later
- Each job has a unique identifier (and also an index for replaced jobs)

```
from jobflow import job

@job
def add(a, b):
    return a + b

add_job = add(1, 2)
print(add_job)
print(add_job.uuid)
print(add_job.index)
```

DEMO

Jobflow : Jobs

- One can run single jobs (not really useful)

```
from jobflow import job, SETTINGS
store = SETTINGS.JOB_STORE
store.connect()

@job
def add(a, b):
    return a + b

add_job = add(1, 2)
add_job.run(store=store)
```

See below

DEMO

- This is actually done internally when executing jobs and flows **locally**, **with Fireworks**, or **with Jobflow-remote**

Jobflow : Flows

- A **Flow** is a list of **Jobs** with their interdependencies
- Jobs outputs are accessed through the *output* attribute
- Connectivity/Dependencies automatically detected

```
from jobflow import job, Flow

@job
def add(a, b):
    return a + b

@job
def multiply(a, b):
    return a * b

add_job = add(1, 2)
multiply_job = multiply(add_job.output, 3)

flow = Flow([add_job, multiply_job])
flow.draw_graph().show()
```

Jobflow : Flows

- Graph of connectivity easily visualized

```
from jobflow import job, Flow

@job
def add(a, b):
    return a + b

@job
def multiply(a, b):
    return a * b

add_job = add(1, 2)
multiply_job = multiply(add_job.output, 3)

flow = Flow([add_job, multiply_job])
flow.draw_graph().show()
```

DEMO

Intermediate Wrap-up

- What did we learn ?
  - Logic of deferred output
  - How to create Jobs and Flows?

Jobflow : Running Jobs and Flows

- Jobs can be run locally using the *run_locally* method of Jobflow
  - Runs in the current directory and returns the output of the job
  - Reports some logging messages

```
from jobflow import job
from jobflow import run_locally

@job
def add(a, b):
    return a + b

add_job = add(1, 2)
output = run_locally(add_job)
print(output)
```

```
2022-07-26 16:36:17,018 INFO Started executing jobs locally
2022-07-26 16:36:17,135 INFO Starting job - add (340be7e3-21a1-4db4-a0cc-94b3ed039408)
2022-07-26 16:36:17,135 INFO Finished job - add (340be7e3-21a1-4db4-a0cc-94b3ed039408)
2022-07-26 16:36:17,135 INFO Finished executing jobs locally
```

Jobflow : Running Jobs and Flows

- Jobs can be run locally using the *run_locally* method of Jobflow
  - Runs in the current directory and returns the output of the job
  - Reports some logging messages

```
from jobflow import job,run_locally

@job
def add(a, b):
    return a + b

add_job = add(1, 2)
output = run_locally(add_job)
print(output)
```

```
{'c91d0183-3121-4b4f-ab11-ea67a26d534f': {1: Response(output=3,
                                                      detour=None,
                                                      addition=None,
                                                      replace=None,
                                                      stored_data=None,
                                                      stop_children=False,
                                                      stop_jobflow=False)}}
```

Jobflow : Running Jobs and Flows

- The same is true with Flows
  - It returns the output of all the jobs

```
from jobflow import job, Flow, run_locally

@job
def add(a, b):
    return a + b

@job
def multiply(a, b):
    return a * b

add_job = add(1, 2)
multiply_job = multiply(add_job.output, 3)

flow = Flow([add_job, multiply_job])
output = run_locally(flow)
print(output)
```

Jobflow : Running Jobs and Flows

- The same is true with Flows
  - It returns the output of all the jobs

```python
from jobflow import job, Flow, run_locally

@job
def add(a, b):
    return a + b

@job
def multiply(a, b):
    return a * b

add_job = add(1, 2)
multiply_job = multiply(add_job.output, 3)

flow = Flow([add_job, multiply_job])
output = run_locally(flow)
print(output)
```

```
{'4b47282e-4a71-43b6-9e63-fe9d876c015a': {1: Response(output=9,
                                                       detour=None,
                                                       addition=None,
                                                       replace=None,
                                                       stored_data=None,
                                                       stop_children=False,
                                                       stop_jobflow=False)},
 '4f75f404-4ee0-4dc0-95cd-f9a1d18c123f': {1: Response(output=3,
                                                       detour=None,
                                                       addition=None,
                                                       replace=None,
                                                       stored_data=None,
                                                       stop_children=False,
                                                       stop_jobflow=False)}}
```

DEMO

Jobflow : Running Jobs and Flows

- The same is true with Flows
  - It returns the output of all the jobs
  - Logging

```
2022-07-26 16:44:32,873 INFO Started executing jobs locally
2022-07-26 16:44:32,874 INFO Starting job - add (4f75f404-4ee0-4dc0-95cd-f9a1d18c123f)
2022-07-26 16:44:32,874 INFO Finished job - add (4f75f404-4ee0-4dc0-95cd-f9a1d18c123f)
2022-07-26 16:44:32,874 INFO Starting job - multiply (4b47282e-4a71-43b6-9e63-fe9d876c015a)
2022-07-26 16:44:32,875 INFO Finished job - multiply (4b47282e-4a71-43b6-9e63-fe9d876c015a)
2022-07-26 16:44:32,875 INFO Finished executing jobs locally
```

Jobflow : Running Jobs and Flows

- The same is true with Flows
  - It returns the output of all the jobs
  - Logging
  - By default: all jobs are run in this same folder!
  - Set the *create_folders* argument to *True* to create one folder for each job (highly recommended)

```
output = run_locally(flow, create_folders=True)
```

  - What happens if we don't do that ?

Jobflow : Running Jobs and Flows

- Running in the same folder

```
from jobflow import job, Flow, run_locally
from pathlib import Path

@job
def write(string):
    Path('file.txt').write_text(string)

job1 = write('Hello Michael')
job2 = write('Hello John')
flow = Flow([job1, job2])
run_locally(flow)
```

Jobflow : Running Jobs and Flows

- One directory for each job

```
from jobflow import job, Flow, run_locally
from pathlib import Path

@job
def write(string):
    Path('file.txt').write_text(string)

job1 = write('Hello Michael')
job2 = write('Hello John')
flow = Flow([job1, job2])
run_locally(flow, create_folders=True)
```

DEMO

- Directory structure:

```
job_2022-07-26-14-58-32-654266-46095/
job_2022-07-26-14-58-32-654908-21738/
```

  - Note that when running with Fireworks, this is done automatically

Jobflow : Running Jobs and Flows

- What happens during execution ?
  - The first Job of a Flow runs based on its inputs
  - It stores its output in a database
  - The next Job starts by resolving its arguments
    - From the output references, it looks up in the database and replaces them with the real outputs from the previous Jobs (that have now run)
  - It then runs based on its own inputs and the outputs from previous Jobs
  - It stores its output in a database
  - … and so on and so forth …
- Wait … What ? We did not set up any MongoDB database ?

Small aside : mongoDB and Maggma

- "Pure" **MongoDB** in python is done via the PyMongo python package:

> PyMongo 4.3.2 Documentation ¶
>
> Overview
>
> **PyMongo** is a Python distribution containing tools for working with MongoDB, and is the recommended way to work with MongoDB from Python. This documentation attempts to explain everything you need to know to use **PyMongo**.

https://www.mongodb.com/docs/manual/reference/operator/query/

- **Maggma**:

> What is Maggma
>
> Maggma is a framework to build data pipelines from files on disk all the way to a REST API in scientific environments. Maggma has been developed by the Materials Project (MP) team at Lawrence Berkeley National Laboratory.
>
> Maggma is written in Python and supports Python 3.7+.

Small aside : mongoDB and Maggma

- Maggma **Store**:

> Store ¶
>
> A major challenge in building scalable data piplines is dealing with all the different types of data sources out there. Maggma's `Store` class provides a consistent, unified interface for querying data from arbitrary data sources. It was originally built around MongoDB, so it's interface closely resembles `PyMongo` syntax. However, Maggma makes it possible to use that same syntax to query other types of databases, such as Amazon S3, GridFS, or even files on disk.

- It "behaves" like a MongoDB database
  - MongoDBStore
  - MongoURIStore
  - S3Store
  - JSONStore
  - MemoryStore
  - ...

Real MongoDB databases

"mocked" MongoDB databases

Small aside : mongoDB and Maggma

- By default, Jobflow uses the "mocked" in-memory MemoryStore
  - Note: This only works with *run_locally*, not with Fireworks or Jobflow-remote
- How to store objects/data ?
  - Need of standard way to do that
  - ⇒ serialization
    - process of converting a data object (e.g., Python objects) into a format that allows us to store or transmit the data and then recreate the object when needed using the reverse process of deserialization (examples: pickle, h5py → platform sensitive)
    - JavaScript Object Notation (JSON): dictionary representation of the object
      - MongoDB native format
      - Needs to be implemented!
      - MSONable (monty package: https://materialsvirtuallab.github.io/monty/)

Small aside: Dataclass

- dataclass allows to define classes
  - with less code
  - more functionality out of the box

Without dataclass

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

With dataclass

```
from dataclasses import dataclass

@dataclass
class Person:
    name: str
    age: int
```

- dataclass is compatible with MSONable

Small aside : Dataclass and Serialization

- MSONable (monty package: https://materialsvirtuallab.github.io/monty/)
  - Serialization: *as_dict* method
  - Deserialization: *from_dict* method

```python
from monty.json import MSONable

class Person(MSONable):
    def __init__(self, name, age):
        self.name = name
        self.age = age

p1 = Person("Michael", 46)
p1_dict = p1.as_dict()
print(p1_dict)
p1_from_dict = Person.from_dict(p1_dict)
```

```python
from dataclasses import dataclass
from monty.json import MSONable

@dataclass
class Person(MSONable):
    name: str
    age: int

p1 = Person("Michael", 46)
p1_dict = p1.as_dict()
print(p1_dict)
p1_from_dict = Person.from_dict(p1_dict)
```

Small aside : Serialization

- MSONable (monty package: https://materialsvirtuallab.github.io/monty/)
  - Recursive serialization: *MontyEncoder*
  - Recursive deserialization: *MontyDecoder*

```
from monty.json import MontyEncoder, MontyDecoder
from pymatgen.util.testing import PymatgenTest

si_struct = PymatgenTest().get_structure('Si')
sio2_struct = PymatgenTest().get_structure('SiO2')
srtio3_struct = PymatgenTest().get_structure('SrTiO3')

data = {"structures": [si_struct, sio2_struct],
        "out": [1, {"buried_struct": srtio3_struct}]}
print(data)
data_dict = MontyEncoder().encode(data)
decoder = MontyDecoder()
data_from_dict = MontyDecoder().decode(data_dict)
print(data_from_dict)
```

Small aside : Serialization

- MSONable (monty package: https://materialsvirtuallab.github.io/monty/)
  - Dump to file: *dumpfn*
  - Load from file: *loadfn*

```
from monty.serialization import dumpfn, loadfn
from pymatgen.util.testing import PymatgenTest

si_struct = PymatgenTest().get_structure('Si')
sio2_struct = PymatgenTest().get_structure('SiO2')
srtio3_struct = PymatgenTest().get_structure('SrTiO3')

data = {"structures": [si_struct, sio2_struct],
        "out": [1, {"buried_struct": srtio3_struct}]}
dumpfn(data, fn="data.json", indent=2)
data_from_file = loadfn(fn="data.json")
print(data_from_file)
```

Jobflow: Stores

- The Job store is configured in the *jobflow.yaml* file:

Maggma

```yaml
JOB_STORE:
  docs_store:
    type: MongoStore
    database: <<DB_NAME>>
    host: <<HOSTNAME>>
    port: <<PORT>>
    username: <<USERNAME>>
    password: <<PASSWORD>>
    collection_name: outputs
  additional_stores:
    data:
      type: GridFSStore
      database: <<DB_NAME>>
      host: <<HOSTNAME>>
      port: <<PORT>>
      username: <<USERNAME>>
      password: <<PASSWORD>>
      collection_name: outputs_blobs
```

Main database:

Things you are likely to query on (final energy, density, ...)

Additional database:

Things you want to be able to access (coordinates, electronic structure, ...)

Jobflow: Stores

- The Job store is configured in the *jobflow.yaml* file.
- The store can be accessed in python through the jobflow's settings:

```
from jobflow import SETTINGS

store = SETTINGS.JOB_STORE
```

- When a job has finished and has been saved in the Job store, it is possible to update the metadata (or anything else in the Job output document)

```
from jobflow import SETTINGS

store = SETTINGS.JOB_STORE
store.connect()

job_document = store.query_one({'metadata.mpid': 'mp-149'})
job_document['metadata']['reduced_formula'] = 'Si'

store.update(job_document)
```

Jobflow : Dynamic workflows

- Fibonnacci(1, 1): 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …
  - Using an *addition*

```
from jobflow import Response, job, run_locally

@job
def fibonacci(smaller, larger, stop_point=55):
    total = smaller + larger

    if total > stop_point:
        return total

    new_job = fibonacci(larger, total, stop_point=stop_point)
    return Response(output=total, addition=new_job)

fibonacci_job = fibonacci(1, 1)

outputs = run_locally(fibonacci_job, create_folders=True)
print(outputs)
```

Jobflow : Dynamic workflows

```
{'00424864-5a74-4611-b48f-a2223b2ee9e2': {1: Response(output=5,
                                          detour=None,
                                          addition=<jobflow.core.job.Job object at 0x7fba1c40dab0>,
                                          replace=None,
                                          stored_data=None,
                                          stop_children=False,
                                          stop_jobflow=False)},
 '21cc54a3-758d-48ba-a043-aea225fc0ade': {1: Response(output=3,
                                          detour=None,
                                          addition=<jobflow.core.job.Job object at 0x7fba1c3f15a0>,
                                          replace=None,
                                          stored_data=None,
                                          stop_children=False,
                                          stop_jobflow=False)},
 '2627bd79-8556-42d0-97d1-edf03024557f': {1: Response(output=2,
                                          detour=None,
                                          addition=<jobflow.core.job.Job object at 0x7fba1c42dea0>,
                                          replace=None,
                                          stored_data=None,
                                          stop_children=False,
                                          stop_jobflow=False)},
 ...
 'b3ee9b1a-3a59-4626-8a5f-2756d6ff1f80': {1: Response(output=13,
                                          detour=None,
                                          addition=<jobflow.core.job.Job object at 0x7fba1c40c670>,
                                          replace=None,
                                          stored_data=None,
                                          stop_children=False,
                                          stop_jobflow=False)}}
```

Jobflow : Dynamic workflows

- Fibonnacci(1, 1): 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …
  - Using a *replace*

```
from jobflow import Response, job, run_locally

@job
def fibonacci(smaller, larger, stop_point=55):
    total = smaller + larger

    if total > stop_point:
        return total

    new_job = fibonacci(larger, total, stop_point=stop_point)
    return Response(output=total, replace=new_job)

fibonacci_job = fibonacci(1, 1)

outputs = run_locally(fibonacci_job, create_folders=True)
print(outputs)
```

Jobflow : Dynamic workflows

```
{'e039bbab-ab83-45ff-afb4-e4673b8eb83d': {1: Response(output=2,
                                                      detour=None,
                                                      addition=None,
                                                      replace=<jobflow.core.job.Job object at 0x7fba1c40f190>,
                                                      stored_data=None,
                                                      stop_children=False,
                                                      stop_jobflow=False),
                                          2: Response(output=3,
                                                      detour=None,
                                                      addition=None,
                                                      replace=<jobflow.core.job.Job object at 0x7fba1c3f1990>,
                                                      stored_data=None,
                                                      stop_children=False,
                                                      stop_jobflow=False),
                                          3: Response(output=5,
                                                      detour=None,
                                                      addition=None,
                                                      replace=<jobflow.core.job.Job object at 0x7fba1c3f1060>,
                                                      stored_data=None,
                                                      stop_children=False,
                                                      stop_jobflow=False),
...
                                          8: Response(output=55,
                                                      detour=None,
                                                      addition=None,
                                                      replace=None,
                                                      stored_data=None,
                                                      stop_children=False,
                                                      stop_jobflow=False)}}
```

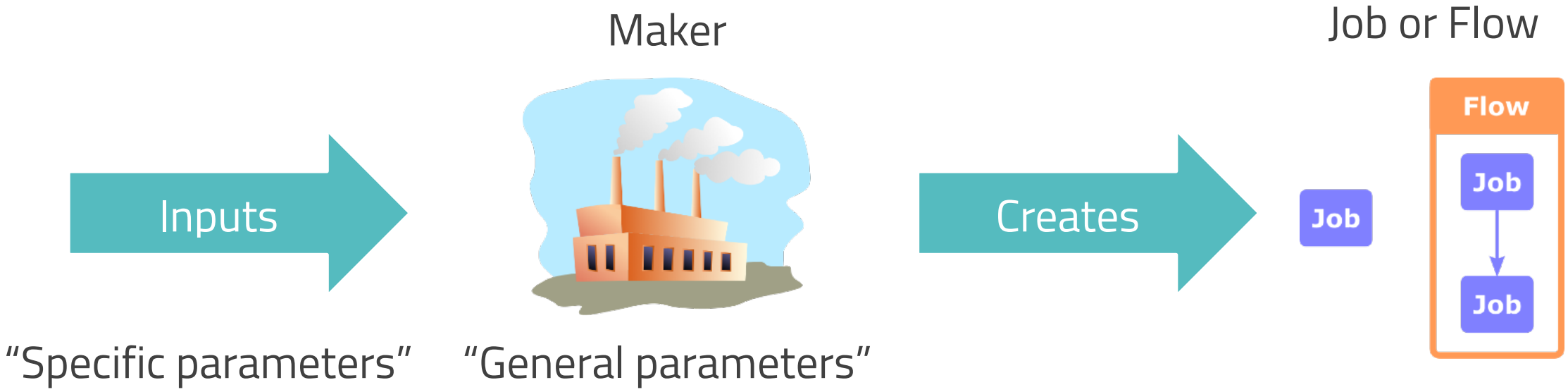
DEMO

Intermediate Wrap-up

- What did we learn ?
  - Logic of deferred output
  - How to create Jobs and Flows?
  - How to run Jobs and Flows?
  - Where these Jobs and Flows run?
  - Database of jobs/flows and results
    - Maggma Stores

What's next?

- How do we make things more automatic ?
  - Automatic generation of jobs/flows based on generic parameters
  - Concept of **Makers**



Jobflow : Makers

- Example: Relaxation Workflow
  - **General parameters**:

    the parameters of the procedure that will be applied to all my calculations
    - Accuracy
    - *k*-point density
    - Exchange-correlation
  - **Specific parameters**:

    the parameters for one calculation
  - Structure
  - Atoms/cell constraints

Jobflow : Makers

- Must be a *dataclass*
- Inherit from jobflow's Maker
- Must implement a *make* method that returns a Job or a Flow
- Are automatically serializable
  - because they inherit from Maker
  - and Maker inherits from MSONable
- Can be stored in a file (JSON or yaml) for reproducibility

Jobflow : Makers

- Toy example: IncrementMaker
  - General parameters: increment (in this case, the default value: 1)

```
from dataclasses import dataclass
from jobflow import Maker, job
from monty.serialization import dumpfn

@dataclass
class IncrementMaker(Maker):
    name: str = "increment"
    increment: int = 1

    @job
    def make(self, a):
        return a + self.increment

maker = IncrementMaker()
print(maker)
dumpfn(maker, "maker.json", indent=2)
```

Jobflow : Makers

- Toy example: IncrementMaker
  - General parameters: increment (in this case, 10)

```
from dataclasses import dataclass
from jobflow import Maker, job
from monty.serialization import dumpfn

@dataclass
class IncrementMaker(Maker):
    name: str = "increment"
    increment: int = 1

    @job
    def make(self, a):
        return a + self.increment

maker = IncrementMaker(increment=10)
print(maker)
dumpfn(maker, "maker10.json", indent=2)
```

Jobflow : Makers

- Toy example: IncrementMaker
  - Job creation using makers and run

```
from dataclasses import dataclass
from jobflow import Flow, Maker, job, run_locally
from monty.serialization import dumpfn

@dataclass
class IncrementMaker(Maker):
    name: str = "increment"
    increment: int = 1

    @job
    def make(self, a):
        return a + self.increment

maker = IncrementMaker(increment=10)
increment_job = maker.make(a=4)
print(increment_job)
outputs = run_locally(increment_job, create_folders=True)
print(outputs)
```

DEMO

Intermediate Wrap-up

- What did we learn ?
  - Logic of deferred output
  - How to create Jobs and Flows?
  - How to run Jobs and Flows?
  - Where these Jobs and Flows run?
  - Database of jobs/flows and results
    - Maggma Stores
  - Concept of “Makers”
    - (dataclasses, serialization, MSONable, ...)

Atomate2

Atomate2 is a free, open-source software for performing complex materials science workflows using simple Python functions. Features of atomate2 include

- It is built on open-source libraries: pymatgen, custodian, jobflow, and FireWorks.
- A library of "standard" workflows to compute a wide variety of desired materials properties.
- The ability scale from a single material, to 100 materials, or 100,000 materials.
- Easy routes to modifying and chaining workflows together.
- It can build large databases of output properties that you can query, analyze, and share in a systematic way.
- It automatically keeps meticulous records of jobs, their directories, runtime parameters, and more.

- Clean and flexible Python API
- Workflows for VASP mainly and ABINIT (almost merged)
- Easily share (parametrized) workflows with a single file representing the workflow
- Can be executed with or without a database (MongoDB)

Atomate2

Atomate2 is a free, open-source software for performing complex materials science workflows using simple Python functions. Features of atomate2 include

- It is built on open-source libraries: pymatgen, custodian, jobflow, and FireWorks.
- A library of "standard" workflows to compute a wide variety of desired materials properties.
- The ability scale from a single material, to 100 materials, or 100,000 materials.
- Easy routes to modifying and chaining workflows together.
- It can build large databases of output properties that you can query, analyze, and share in a systematic way.
- It automatically keeps meticulous records of jobs, their directories, runtime parameters, and more.

- Documentation: https://materialsproject.github.io/atomate2
- Github page: https://github.com/materialsproject/atomate2
- Support forum: https://matsci.org/c/atomate

Atomate2: Available workflows

- Structure optimization
- (Molecular dynamics) [draft]
- Electronic band structures
- Elastic, dielectric, and piezoelectric tensors
- Electronic transport
- Phonopy
- …

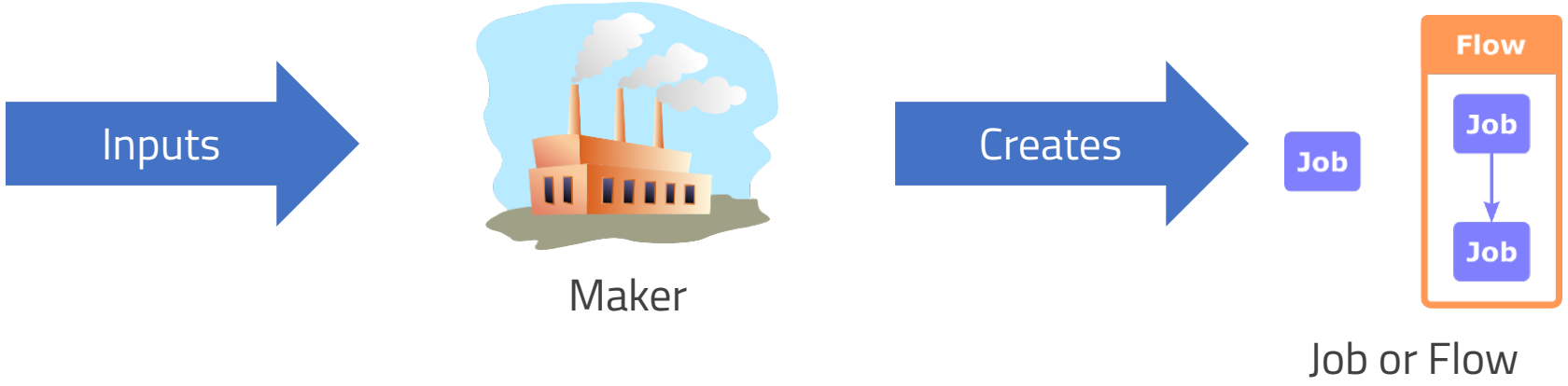
```
from pymatgen.util.testing import PymatgenTest
from atomate2.vasp.flows.core import DoubleRelaxMaker
from jobflow import run_locally

si_struct = PymatgenTest().get_structure('Si')

maker = DoubleRelaxMaker()
relax_flow = maker.make(structure=si_struct)

run_locally(relax_flow, create_folders=True)
```

Atomate2: Input generators

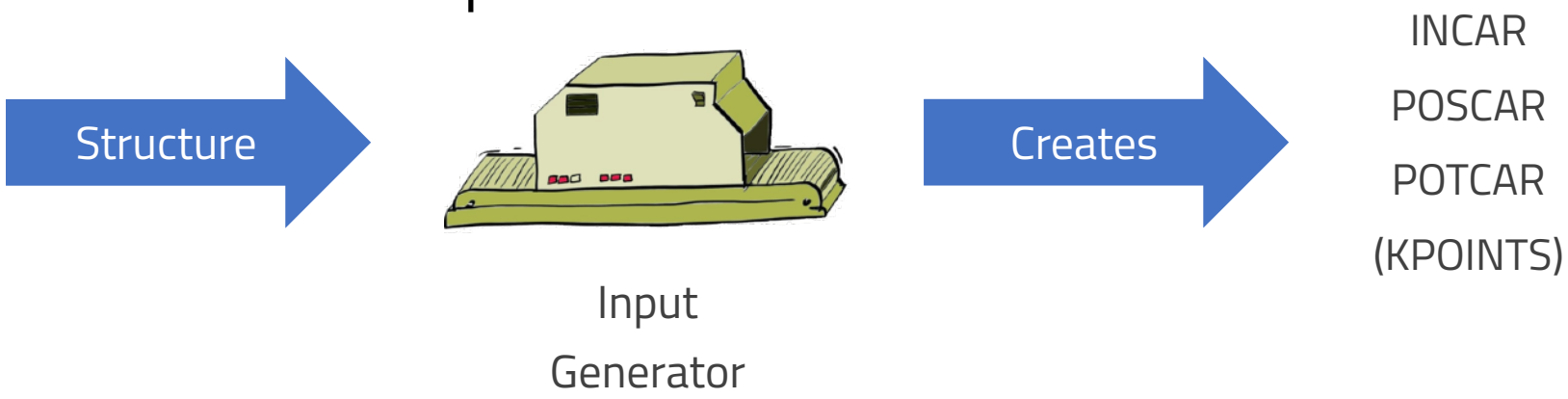
```
from atomate2.vasp.jobs.core import RelaxMaker
from atomate2.vasp.sets.core import RelaxSetGenerator
from monty.serialization import dumpfn

maker = RelaxMaker(
        input_set_generator=RelaxSetGenerator(
            user_incar_settings={'ENCUT': 250},
            user_kpoints_settings={'reciprocal_density': 30}
        )
    )

dumpfn(maker, "relax_maker_loose.json", indent=2)
```

Atomate2: Input generators

- Makers create Jobs (or Flows)



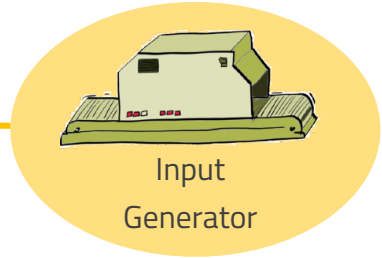
- Input generators create inputs



Atomate2: Input generators

Automated calculations

- **High volume** of calculated properties and materials through an **automatized process** → **high-throughput calculations**
- Several steps involved:
  - Input generation
  - Code execution
  - Error correction
  - Data storage
  - Data analysis



Atomate2: abinit (not yet merged)

- atomate2.abinit.sets.base
  - Special AbinitInputSet (subclass of InputSet)
  - AbinitInputGenerator (abstract class)
    - Contains all the machinery to link from previous files/calculations
    - Uses abipy heavily
    - Need to implement a get_abinit_input method (returns an abipy AbinitInput)
- atomate2.abinit.sets.core
  - Actual implementations (subclasses)
    - StaticSetGenerator
    - NonSCFSetGenerator
    - RelaxSetGenerator

Jobflow: article

1 Jobflow: Computational Workflows Made Simple

2 **Andrew S. Rosen** [1,2], **Max Gallant** [1,2], **Janine George** [3,4], **Janosh**
3 **Riebesell** [2,5], **Hrushikesh Sahasrabuddhe** [1,6], **Jimmy-Xuan Shen** [7],
4 **Mingjian Wen** [8], **Matthew L. Evans** [9,10], **Guido Petretto**[9], **David**
5 **Waroquiers** [9,10], **Gian-Marco Rignanese** [9,10,11], **Kristin A. Persson** [1,2,12],
6 **Anubhav Jain** [6], and **Alex M. Ganose** [13]

[J. Open Source Softw. **9**, 5995 (2024)]