University of BRISTOL

SWIFT HEP

SoftWare InFrastructure and Technology for High Energy Physics

SWIFT-HEP WP5 Analysis Systems

Sam Eriksen

27th March 2024

Overview

- WP5 overview + roadmap
- Current progress in WP5
- Future for WP5

Previous updates

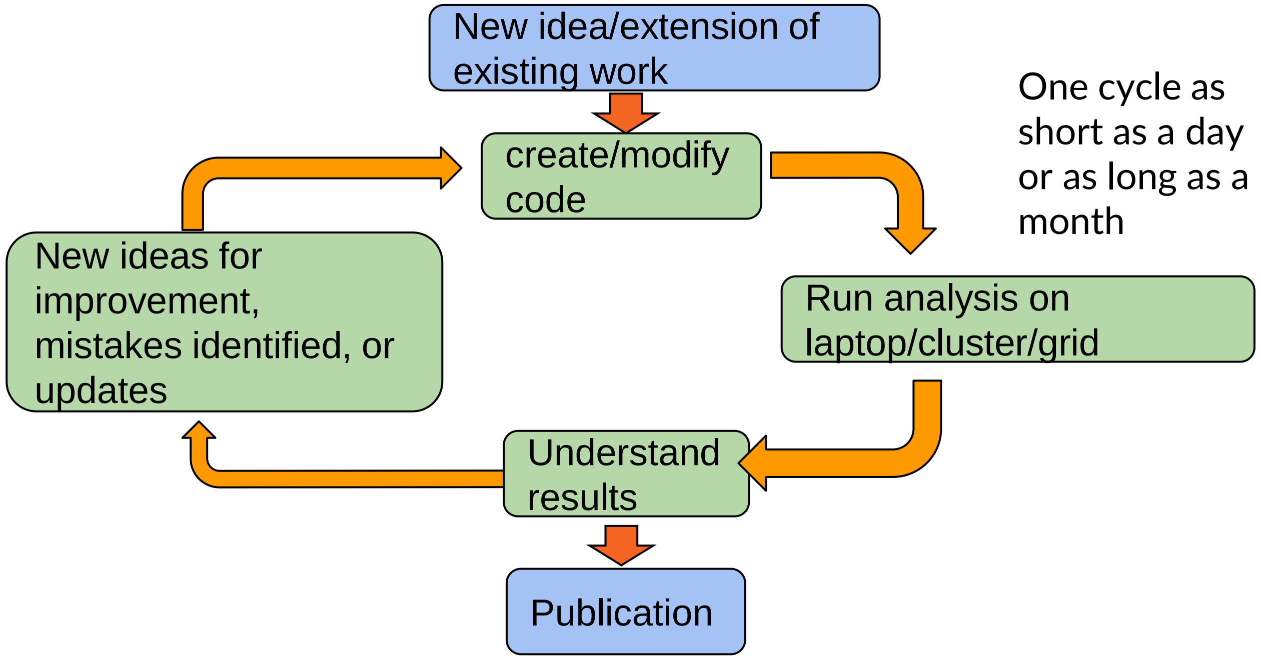
- November 2023
- September 2023
- May 2023
- March 2023
- February 2023

WP5: Analysis Systems

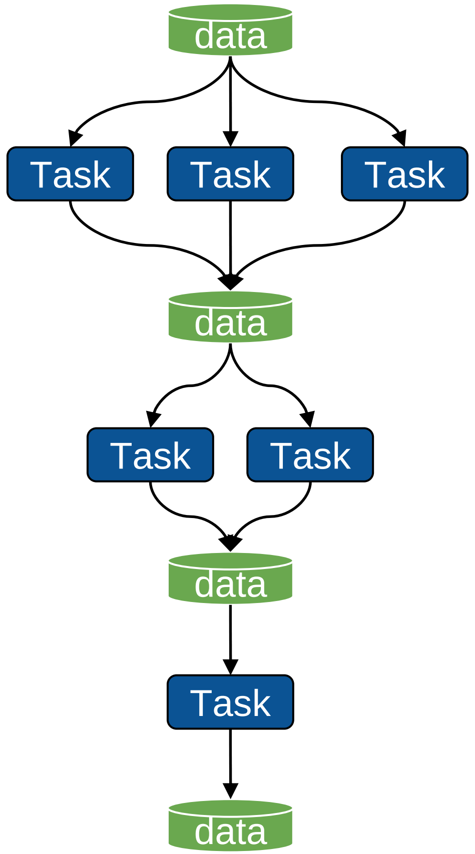
WP5: Analysis Systems

Run analysis workloads optimally on distributed resources

Analysis Anatomy

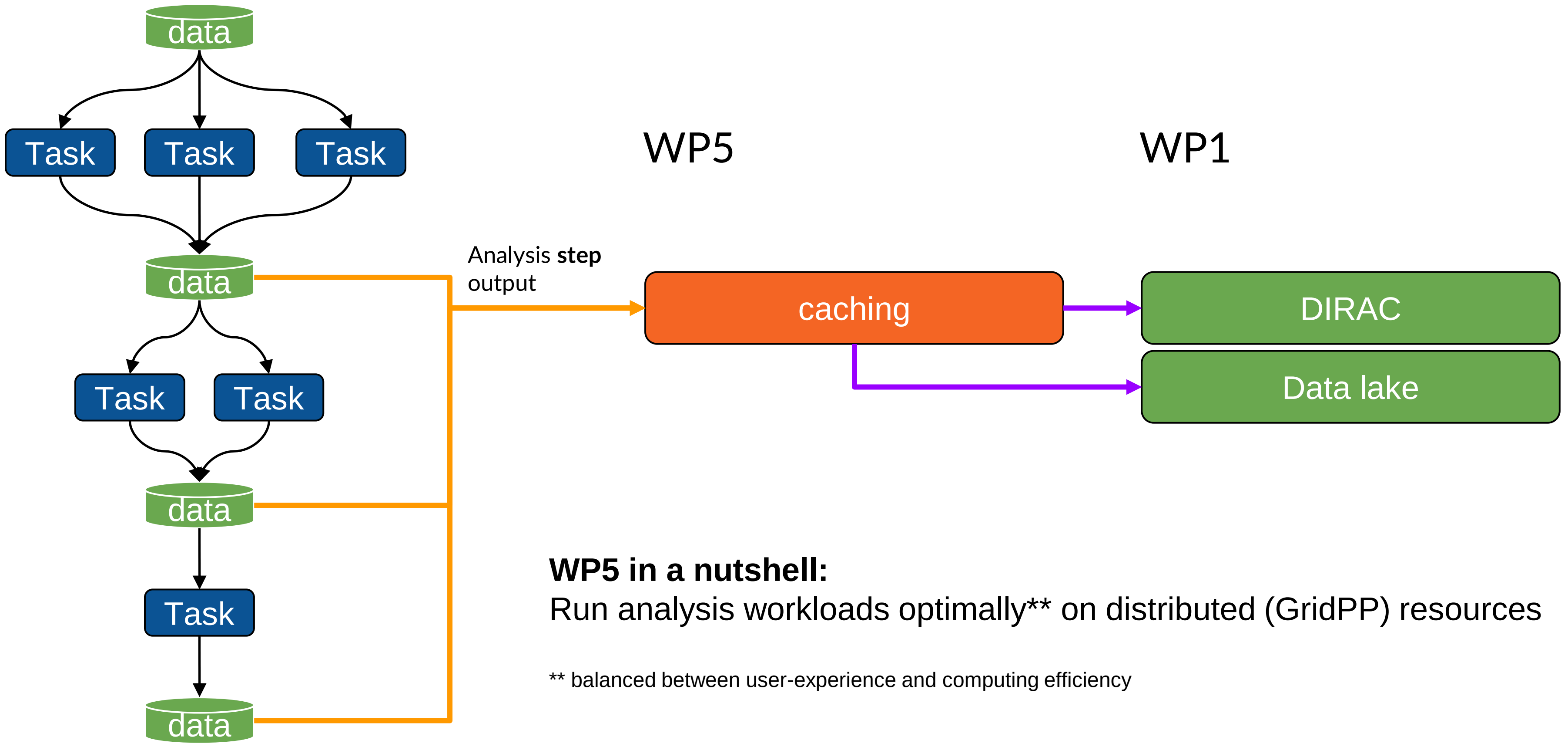

New idea/extension of existing work
create/modify code
Run analysis on laptop/cluster/grid
Understand results
New ideas for improvement, mistakes identified, or updates
Publication

One cycle as short as a day or as long as a month



See talk by Luke Kreczko for some **BIG** Picture

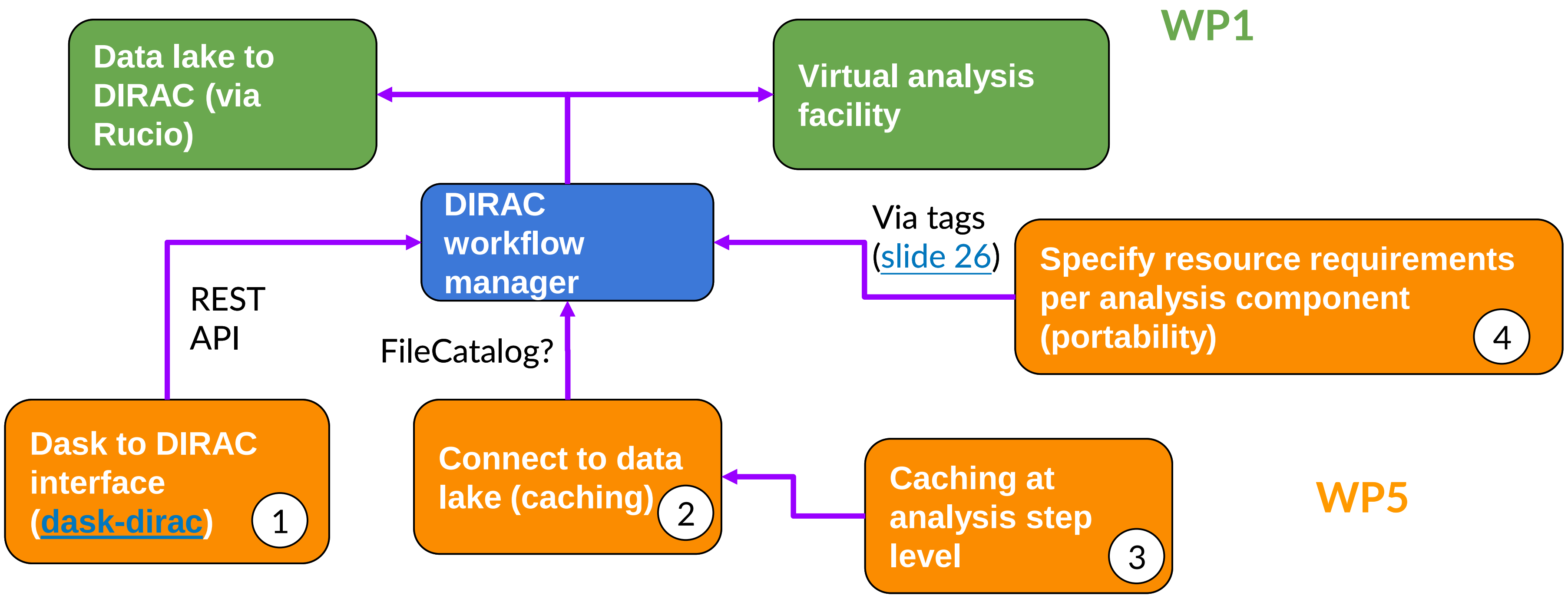
Analysis Anatomy



WP5 in a nutshell:
Run analysis workloads optimally** on distributed (GridPP) resources

** balanced between user-experience and computing efficiency

WP5: Roadmap


WP1
Data lake to DIRAC (via Rucio)
Virtual analysis facility
DIRAC workflow manager
Via tags (slide 26)
Specify resource requirements per analysis component (portability) 4
REST API
FileCatalog?
Dask to DIRAC interface (dask-dirac) 1
Connect to data lake (caching) 2
Caching at analysis step level 3
WP5

More detailed slides

WP5: Roadmap

1 **Dask to DIRAC interface (dask-dirac)**

- Add extension to dask
- Dask is able to parallelize any python code

2 **Connect to data lake (caching)**

- Add the ability to save output after dask instance has closed

3 **Caching at analysis step level**

- Avoid having to re-run analysis steps

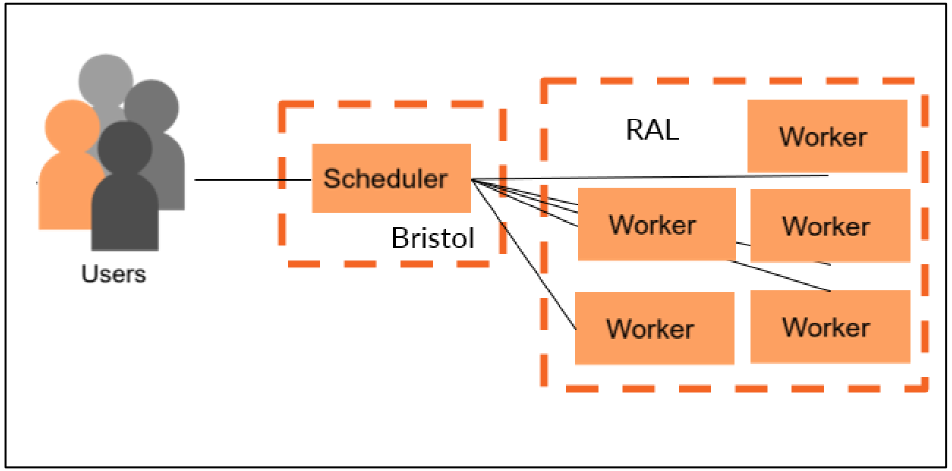
4 **Specify resource requirements per analysis component (portability)**

- E.g. Let some stages run on GPUs

See talk by Luke Kreczko for some future planning

Where did we leave things at the last workshop

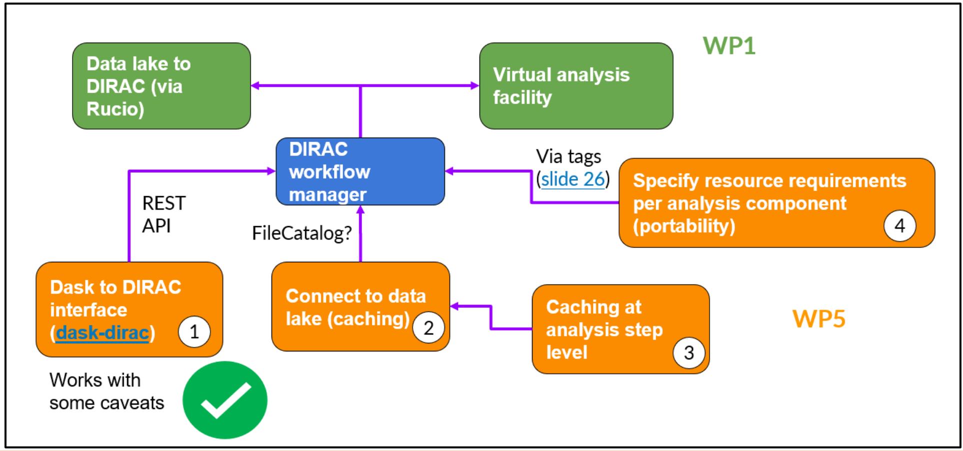
Where we left things last time



- Have a stable workflow
- Start scheduler at Bristol
- Submit jobs (via DIRAC) to run workers are some site (Bristol / RAL)

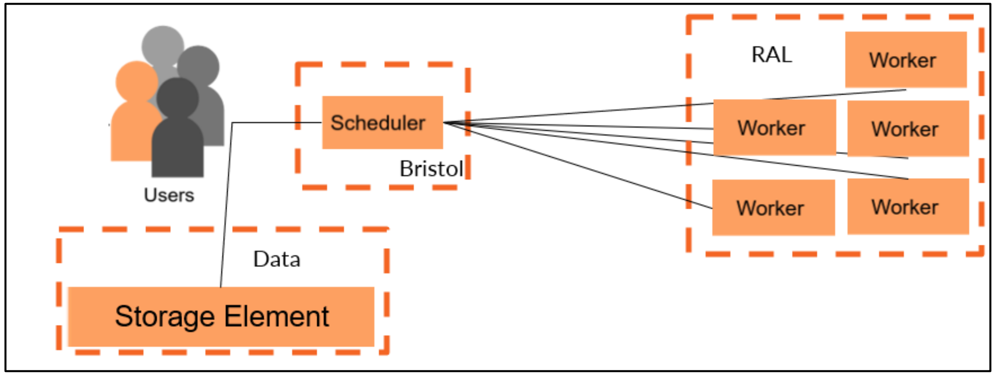
Dask to DIRAC interface (dask-dirac)

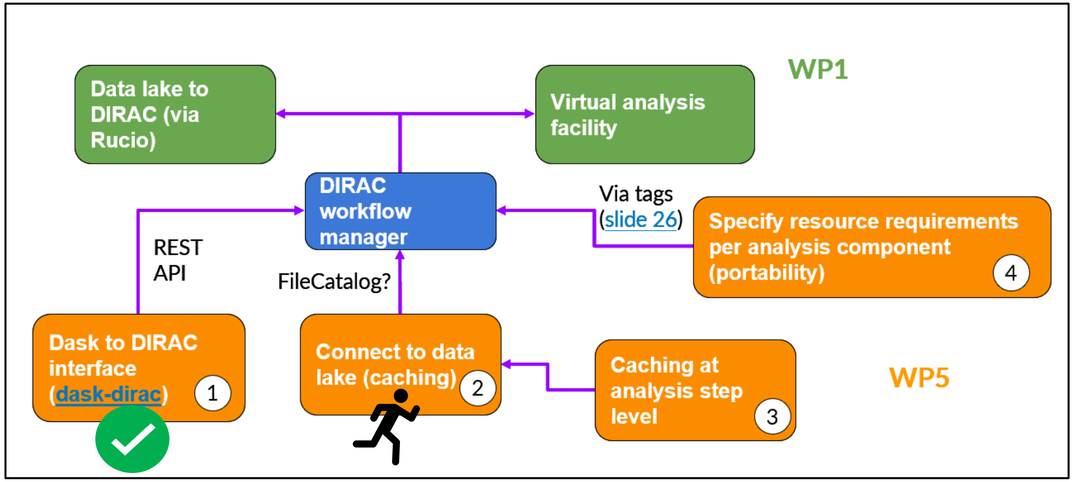
- Reliable dask-DIRAC interface
- Integrated with CMS AGC
- Performed some basic benchmarking


WP1
Data lake to DIRAC (via Rucio)
Virtual analysis facility
DIRAC workflow manager
Via tags (slide 26)
Specify resource requirements per analysis component (portability) 4
REST API
FileCatalog?
Dask to DIRAC interface (dask-dirac) 1
Connect to data lake (caching) 2
Caching at analysis step level 3
WP5
Works with some caveats

Where we left things last time

- Able to do file manipulation (fairly easy – except adding files)
- Use gfal for file adding
- Register file via HTTP




WP1
Data lake to DIRAC (via Rucio)
Virtual analysis facility
DIRAC workflow manager
Via tags (slide 26)
Specify resource requirements per analysis component (portability) 4
REST API
FileCatalog?
Dask to DIRAC interface (dask-dirac) 1
Connect to data lake (caching) 2
Caching at analysis step level 3
WP5

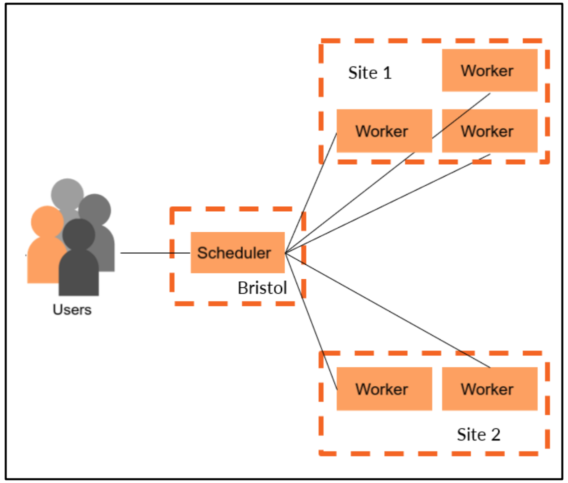
- This stage was painful
- Not automated

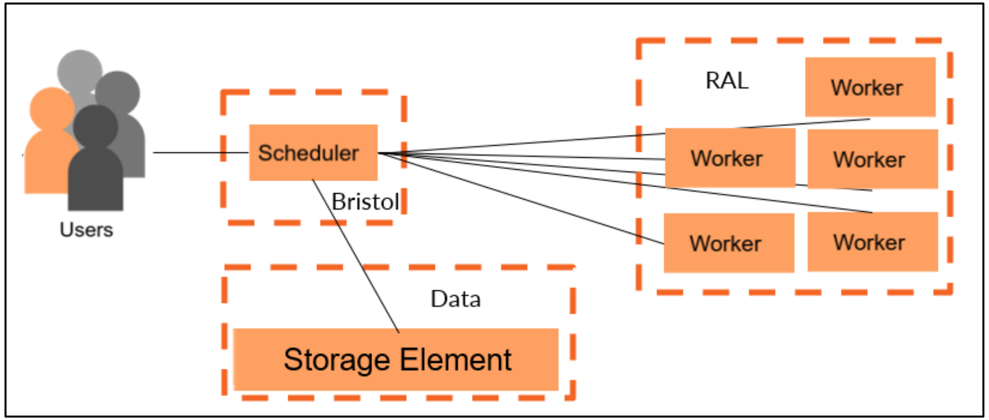
What's happened since the last workshop

What have we been focusing on

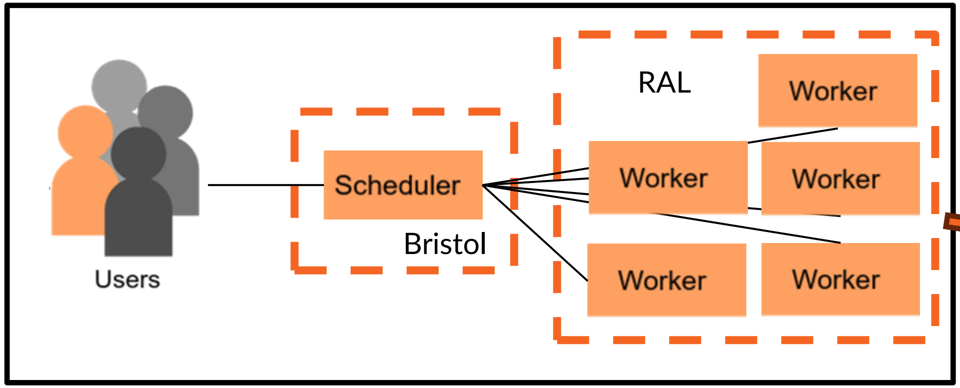
- Enable multi-site submission

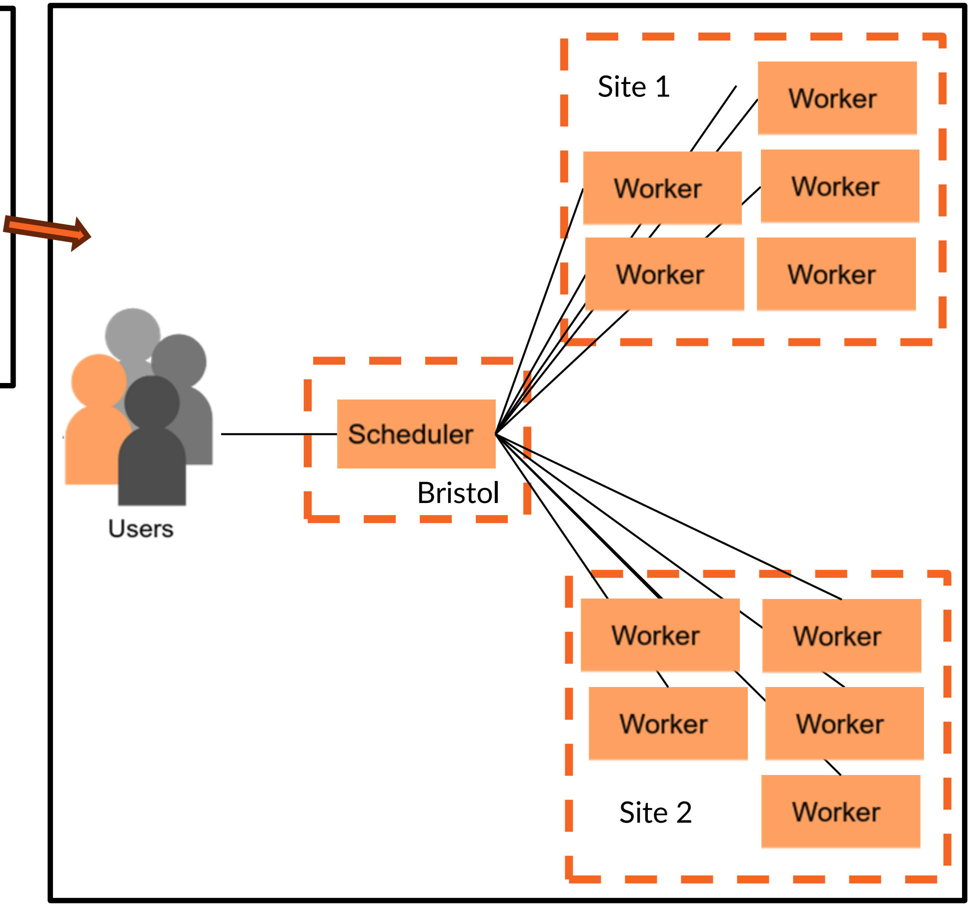
- Explore best ways to interact with data storage





Multi-site submission





- Moved JDL handling to use templates
- Able to easily control which sites are used
- May want to add more fine control in future (eg 3-workers at site X, 4-workers at site Y for data locality)

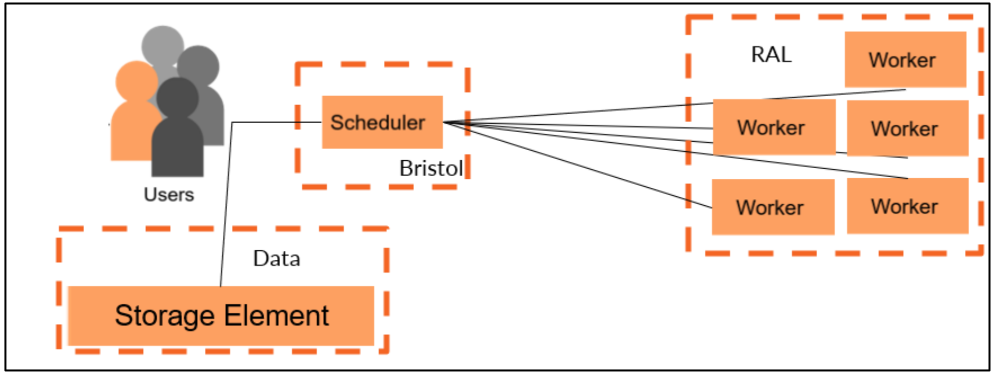
Interactions with data lake

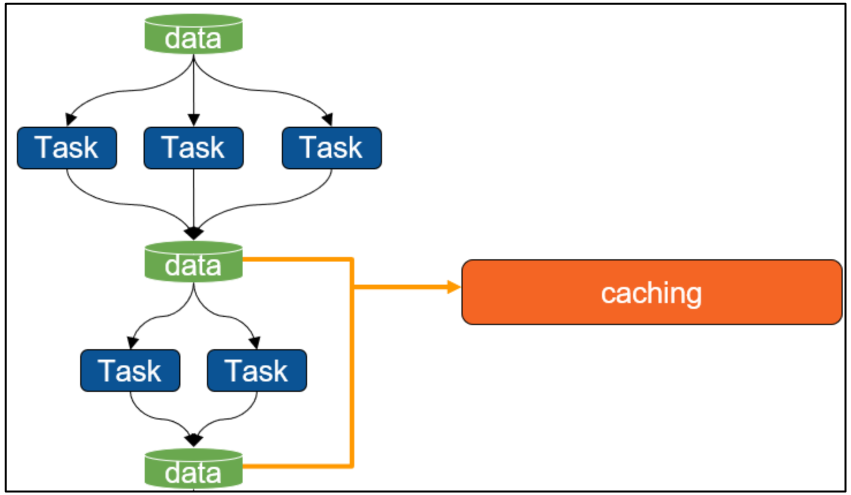
Connect to data lake (caching)

Simplified workflow

1. Run some analysis
2. Get it sent back to you
3. Save to locally
4. Move it to storage

Does the data need to be sent all the way back to the user?





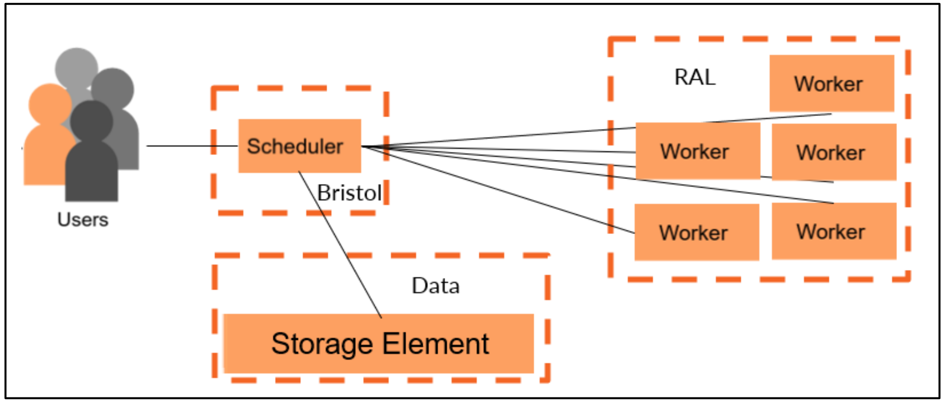
Interactions with data lake

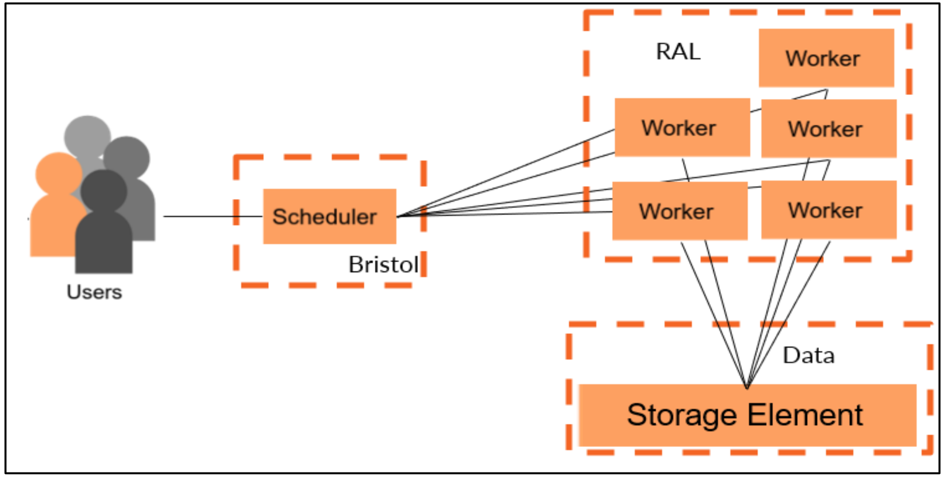
Connect to data lake (caching)

More ‘efficient’ workflow

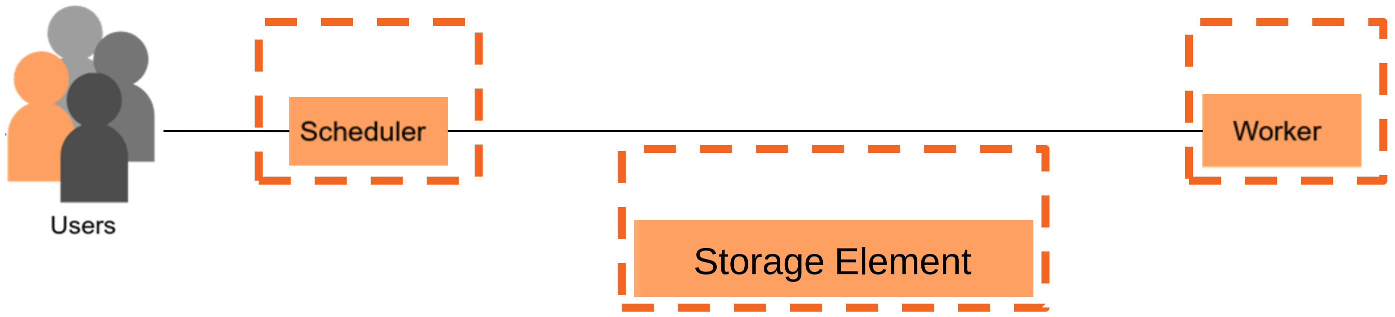
1. Run some analysis
2. Have the data saved as default
3. Get it sent back to you if you like

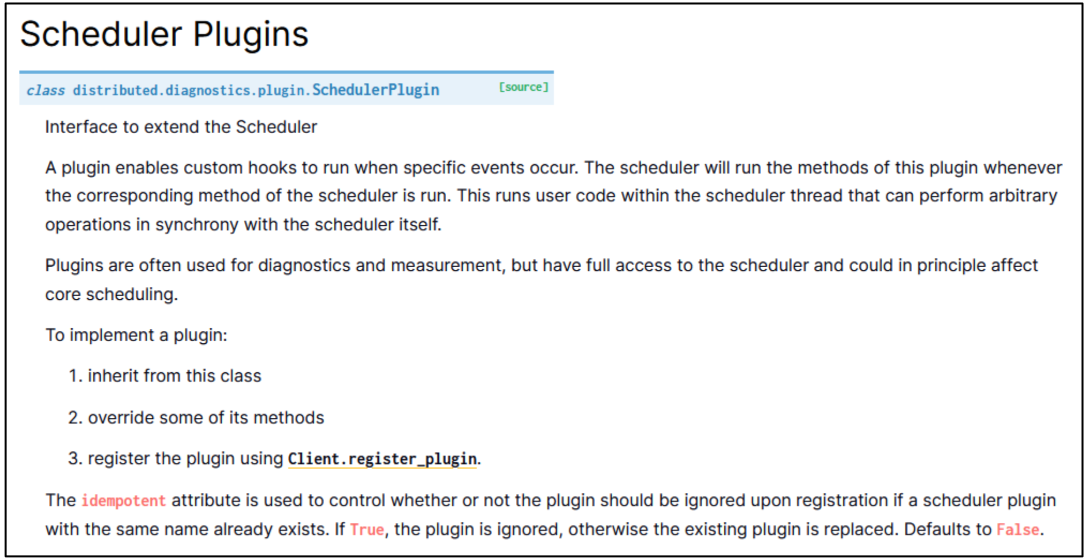
What are possible ways to integrate this in the dask framework?





dask plugins





Scheduler Plugins

`class distributed.diagnostics.plugin.SchedulerPlugin` [source]

Interface to extend the Scheduler

A plugin enables custom hooks to run when specific events occur. The scheduler will run the methods of this plugin whenever the corresponding method of the scheduler is run. This runs user code within the scheduler thread that can perform arbitrary operations in synchrony with the scheduler itself.

Plugins are often used for diagnostics and measurement, but have full access to the scheduler and could in principle affect core scheduling.

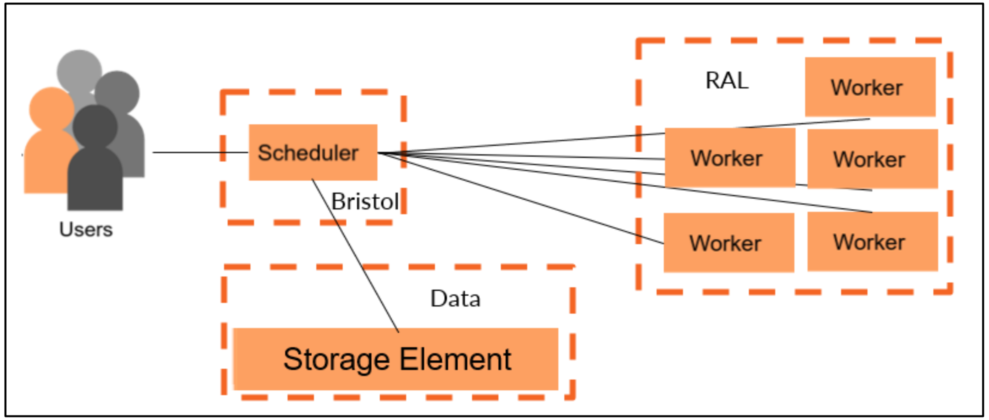
To implement a plugin:

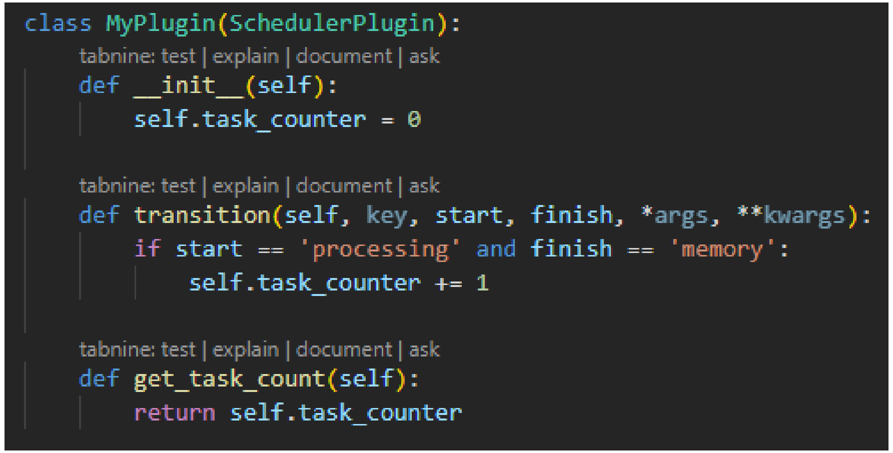
1. inherit from this class
2. override some of its methods
3. register the plugin using `Client.register_plugin`.

The `idempotent` attribute is used to control whether or not the plugin should be ignored upon registration if a scheduler plugin with the same name already exists. If `True`, the plugin is ignored, otherwise the existing plugin is replaced. Defaults to `False`.

dask plugins docs

Automation via Scheduler

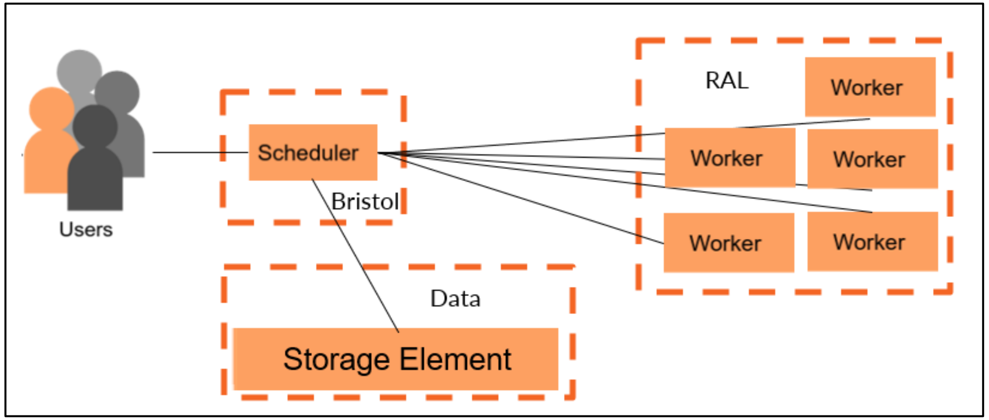



```
class MyPlugin(SchedulerPlugin):
    tabnine: test | explain | document | ask
    def __init__(self):
        self.task_counter = 0

    tabnine: test | explain | document | ask
    def transition(self, key, start, finish, *args, **kwargs):
        if start == 'processing' and finish == 'memory':
            self.task_counter += 1

    tabnine: test | explain | document | ask
    def get_task_count(self):
        return self.task_counter
```

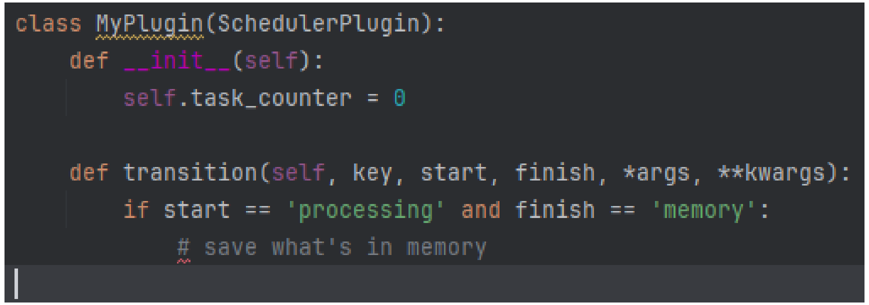
Things that can happen;

1. Run task when scheduler starts
2. Run task when task state changes
3. Run task when worker(s) connects/disconnects

Automation via Scheduler

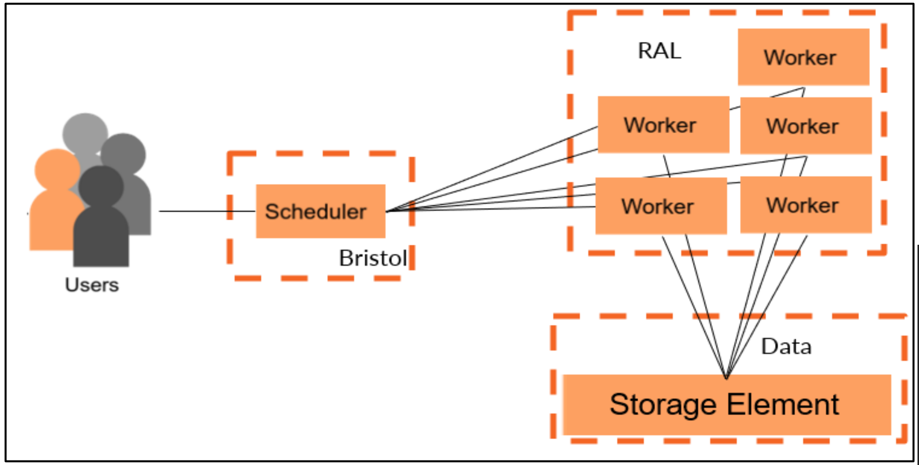


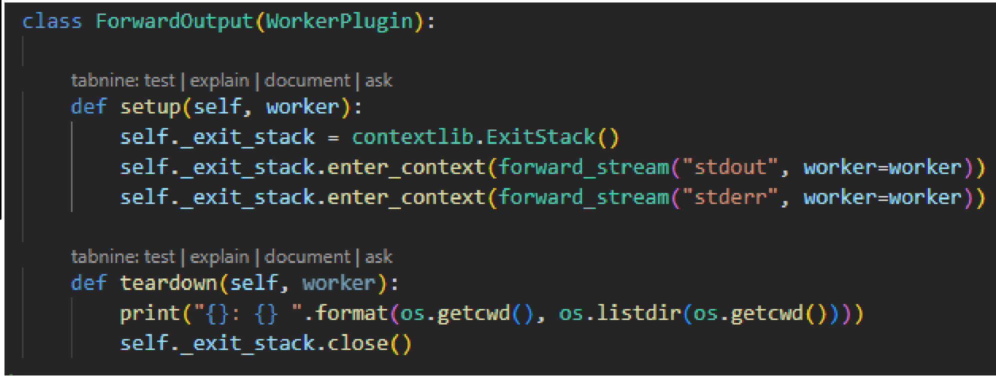
When tasks have processed and are in memory,
Do something



```
class MyPlugin(SchedulerPlugin):
    def __init__(self):
        self.task_counter = 0

    def transition(self, key, start, finish, *args, **kwargs):
        if start == 'processing' and finish == 'memory':
            # save what's in memory
```

Automation via Worker

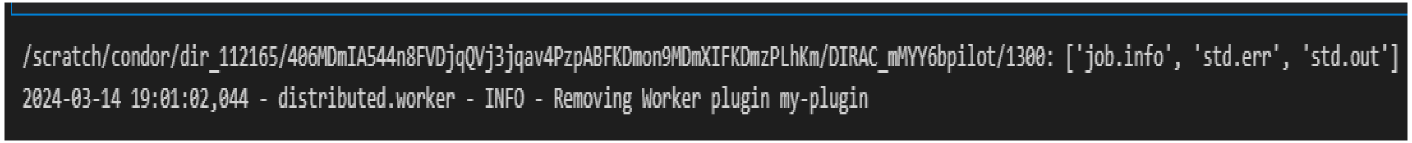




```
class ForwardOutput(WorkerPlugin):

    tabnine: test | explain | document | ask
    def setup(self, worker):
        self._exit_stack = contextlib.ExitStack()
        self._exit_stack.enter_context(forward_stream("stdout", worker=worker))
        self._exit_stack.enter_context(forward_stream("stderr", worker=worker))

    tabnine: test | explain | document | ask
    def teardown(self, worker):
        print("{}: {} ".format(os.getcwd(), os.listdir(os.getcwd())))
        self._exit_stack.close()
```

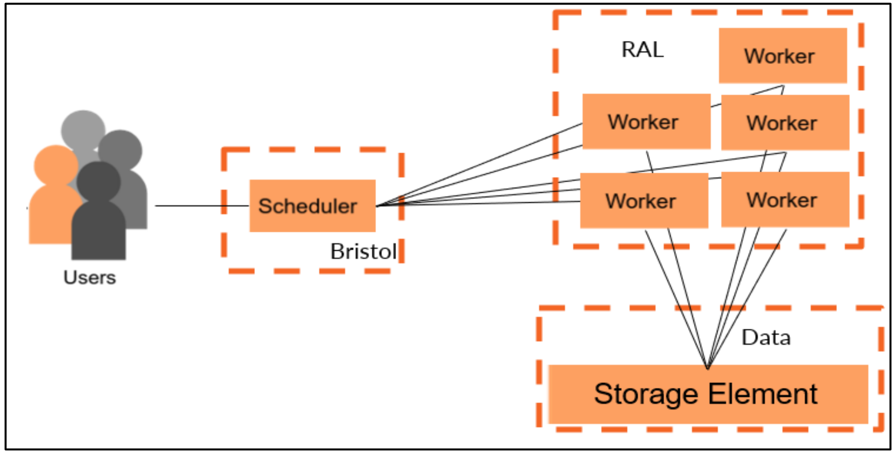
Things that can happen;

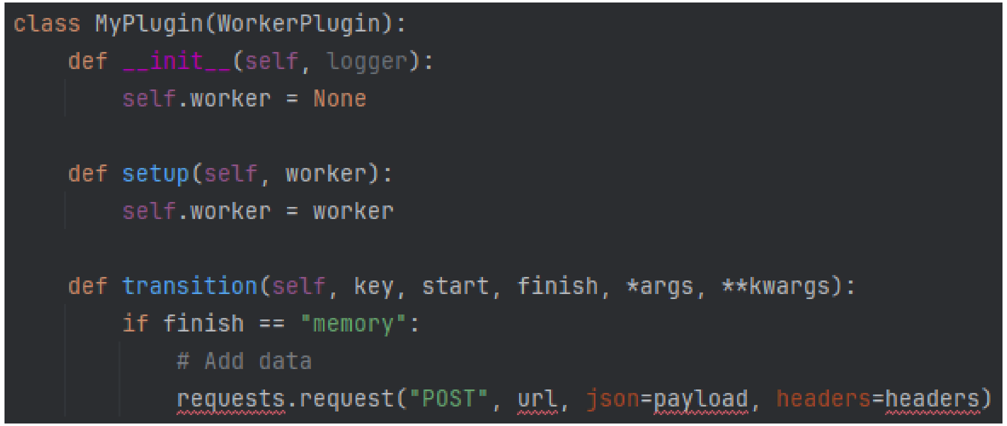
1. Run task when worker starts/shutsdown
2. Run task when task state changes



```
/scratch/condor/dir_112165/406MDmIA544n8FVDjqQVj3jqav4PzpABFKDmon9MDmXIFKDmzPLhKm/DIRAC_mMYY6bpilot/1300: ['job.info', 'std.err', 'std.out']
2024-03-14 19:01:02,044 - distributed.worker - INFO - Removing Worker plugin my-plugin
```

What we are working towards

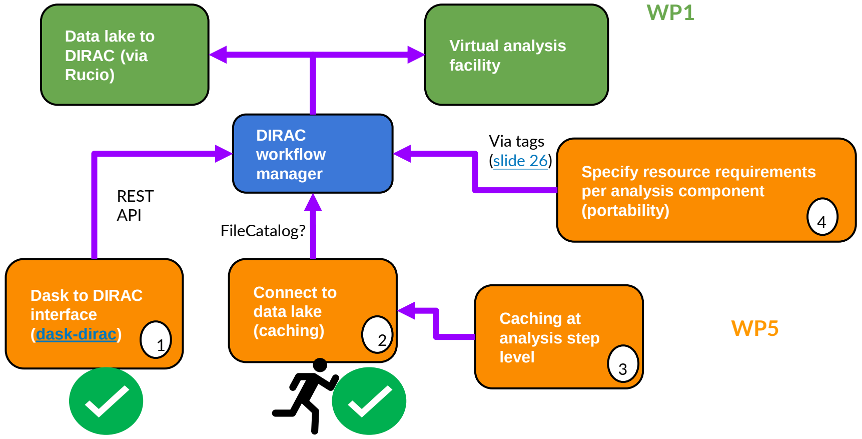




```
class MyPlugin(WorkerPlugin):
    def __init__(self, logger):
        self.worker = None

    def setup(self, worker):
        self.worker = worker

    def transition(self, key, start, finish, *args, **kwargs):
        if finish == "memory":
            # Add data
            requests.request("POST", url, json=payload, headers=headers)
```

1. Run some stage of analysis
2. Save output with RUCIO (above is REST API setup)
3. Implement a check before task starts to see if saved version exists – can also happen on worker startup.

Summary and plan going forward



Data lake to DIRAC (via Rucio)
Virtual analysis facility
WP1
DIRAC workflow manager
Via tags (slide 26)
Specify resource requirements per analysis component (portability)
4
REST API
FileCatalog?
Dask to DIRAC interface (dask-dirac)
1
Connect to data lake (caching)
2
Caching at analysis step level
3
WP5

- Multi-site worker deployment now works
- Dask plugins for data lake interactions being actively worked on. Once that it done, caching becomes possible
- We are still in the x509 -> token transition, so may be messy in the short term (see talk by Daniela)

University of BRISTOL

Questions?

SWIFT HEP

SoftWare InFrastructure and Technology for High Energy Physics

BACKUP

Making things reproducible

DIRAC certificate

- Get a conda env following: https://github.com/DIRACGrid/DIRACOS2
- Then run `dirac-proxy-init –g gridpp_user`

Get dask-dirac

- From pypi `pip install dask-dirac`
- (For development, use github)

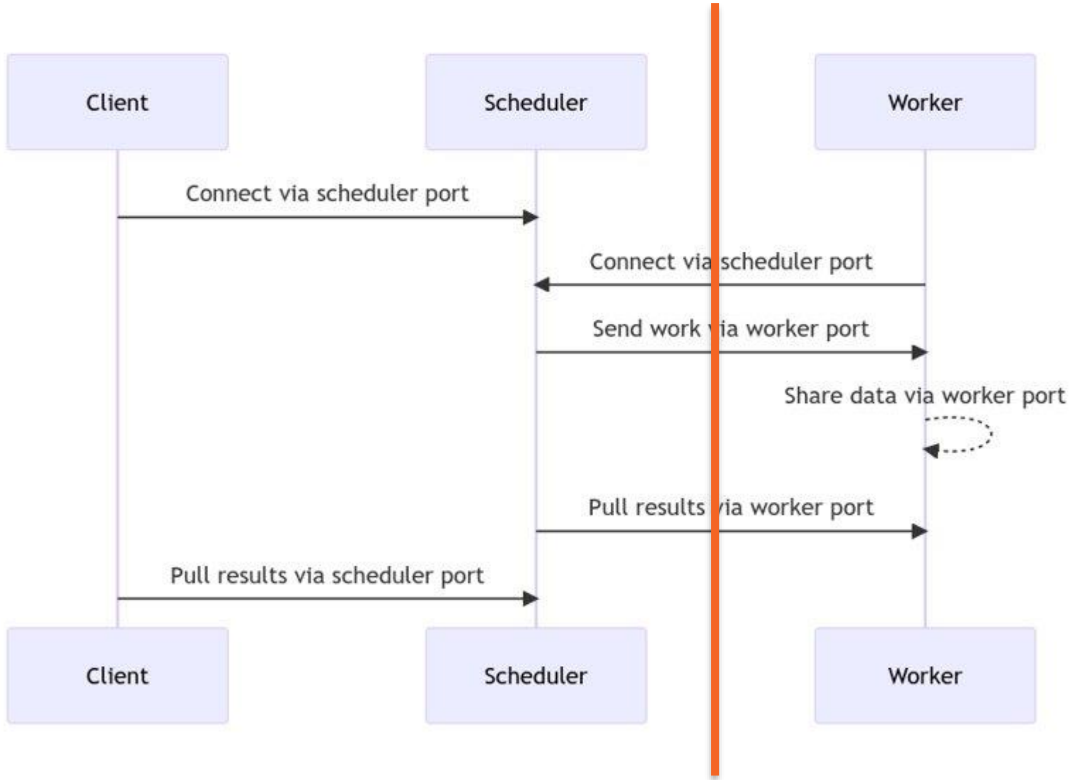
Get AGC branch with edits

- Using AGC from SWIFT-HEP fork
- `git clone -b se_daskdirac git@github.com:SWIFT-HEP/analysis-grand-challenge.git`
- Edit config to use users certificate

Run CMS analysis

Documentations in dask-dirac is in progress

The network explained


Client
Scheduler
Worker
Connect via scheduler port
Connect via scheduler port
Send work via worker port
Share data via worker port
Pull results via worker port
Pull results via scheduler port
Client
Scheduler
Worker

So where is the problem?

Imagine network boundary between scheduler and workers

Scheduler port is accessible from workers

Worker port is **ONLY** accessible to scheduler if connection is recycled (part of **ESTABLISHED** --> firewall OK)

Default Dask operation: this can happen at **RANDOM** (most likely for small # of workers)

The network solution

- We know connections can be recycled and bypass firewall if they are part of an ESTABLISHED connection

- We also know of a working solution in our field: The HTCondor Connection Broker
    - Workers, schedulers, etc connect to a SHARED_PORT
    - As long as SHARED_PORT is open in firewall on a node accessible to both scheduler and workers --> connection can be established

- Most simple solution: Can the Dask Connection proxy be rewritten to hold worker connections?
    - What are the downsides for 100-1000 worker nodes?