luigi analysis workflow
— Large Scale End-to-End Analysis Automation over Distributed Resources —

Marcel Rieger

CHEP 2023 - Norfolk

9.5.2023
Motivational questions

- **Portability**: Does the analysis depend on ...
  - where it runs?
  - where it stores data?
  - Execution/storage should **not** dictate code design!

- **Reproducibility**: When a postdoc / PhD student leaves, ...
  - can someone else run the analysis?
  - is there a loss of information? Is a new *framework* required?
  - Dependencies often **only** exist in the physicists head!

- **Preservation**: After an analysis is published ...
  - are people investing time to preserve their work?
  - can it be repeated after $O(\text{years})$?
  - Daily working environment should provide preservation features **out-of-the-box**!

- Personal experience: $\frac{2}{3}$ of "analysis" time for technicalities, $\frac{1}{3}$ left for physics
  $\rightarrow$ **Physics output doubled if it were the other way round?**
Most analyses are both **large and complex**
- Structure & requirements between workloads mostly undocumented
- Manual execution & steering of jobs, bookkeeping of data across storage elements, different data revisions, ...
  → Time-consuming & error-prone

**Workflow management must ...**
- provide full automation → Execution through a **single command**
- cover all possible use cases → Examples on next slides
Example: ttbb measurement visualization

* Excerpt of the full analysis
Example: ttbb measurement visualization

Entry points

Results

* Excerpt of the full analysis
Example: Fully orchestrated LHC Run 2 + 3 analysis with columnflow
Example: Analysis Grand Challenge (with ML)

- **Entry point**: Columnar data
- **Dependency**: Event selection, systematic uncertainties
- **Intermediate result**: Statistical model building
- **Result**: Fits results and diagnostics

From Elliott's talk
Example: Embedding other analysis libraries (coffea)

Entry point:

Collections (create task graphs) → Task Graph → Schedulers (execute task graphs)
- Dask Array
- Dask DataFrame
- Dask Bag
- Dask Delayed
- Futures

Single-machine (threads, processes, synchronous)
Distributed

From Lindsey's talk

NB: "Task graph" in the following is a higher-level concept
- Python package for building complex pipelines
- Development started at Spotify, now open-source and community-driven

**Building blocks**
1. Workloads defined as **Task** classes that can **require** other **Tasks**
2. Tasks produce output **Targets**
3. **Parameters** customize tasks & control runtime behavior

- Web UI with two-way messaging (task → UI, UI → task), automatic error handling, task history browser, collaborative features, command line interface, ...
Luigi’s execution model is make-like

1. Create dependency tree for triggered task
2. Determine tasks to actually run:
   - Walk through tree (top-down)
   - For each path, stop if all output targets of a task exist*

- Only processes what is really necessary
- Scalable through simple structure
- Error handling & automatic re-scheduling

* in this case, the task is considered complete
Example dependency trees

Work of a B.Sc. student after 2 weeks!
# reco.py

import luigi

from my_analysis.tasks import Selection

class Reconstruction(luigi.Task):
    dataset = luigi.Parameter(default="ttH")

    def requires(self):
        return Selection(dataset=self.dataset)

    def output(self):
        return luigi.LocalTarget(f"reco_{self.dataset}.root")

    def run(self):
        inp = self.input()  # output() of requirements
        outp = self.output()

        # perform reco on file described by "inp" and produce "outp"
        ...

> python reco.py Reconstruction --dataset ttbar
Software design follows 3 primary goals:

1. Experiment-agnostic core (in fact, not even related to physics)
2. Scalability on HEP infrastructure (but not limited to it)
3. Decoupling of run locations, storage locations & software environments
   - Not constrained to specific resources
   - All components interchangeable

Toolbox to follow an analysis design pattern
- No constraint on language or data structures
  → Not a framework

Most used workflow system for analyses in CMS
- O(20) analyses, O(60-80) people
- Central groups, e.g. HIG, TAU, BTV, ...
1. Job submission

- Idea: submission built into tasks, no need to write extra code
- Currently supported job systems: HTCondor, LSF, gLite, ARC, Slurm, CMS-CRAB
- Mandatory features such as automatic resubmission, flexible task ↔ job matching, job files fully configurable at submission time, internal job staging in case of saturated queues, ...
- From the `htcondor_at_cern` example:

```bash
lxplus129:law_test > law run CreateChars --workflow htcondor
INFO: [pid 30564] Worker Worker(host=lxplus129.cern.ch, username=mrieger) running
CreateChars(branch=-1, start_branch=0, end_branch=26, version=v1)
go to submit 26 htcondor job(s)
submitted 1/26 job(s)
submitted 26/26 job(s)
14:35:40: all: 26, pending: 26 (+26), running: 0 (+0), finished: 0 (+0), retry: 0 (+0), failed: 0 (+0)
... 
14:37:10: all: 26, pending: 0 (+0), running: 26 (+26), finished: 0 (+0), retry: 0 (+0), failed: 0 (+0)
14:37:40: all: 26, pending: 0 (+0), running: 10 (-16), finished: 16 (+16), retry: 0 (+0), failed: 0 (+0)
14:38:10: all: 26, pending: 0 (+0), running: 0 (+0), finished: 26 (+10), retry: 0 (+0), failed: 0 (+0)
INFO: [pid 30564] Worker Worker(host=lxplus129.cern.ch, username=mrieger) done!
```

```bash
lxplus129:law_test >
```
Job status polling from CMS HH combination

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>State</th>
<th>ID</th>
<th>Start Time</th>
<th>Duration</th>
<th>Status</th>
<th>Retries</th>
<th>Failed</th>
</tr>
</thead>
<tbody>
<tr>
<td>16:04:23</td>
<td>all</td>
<td>3321</td>
<td>2821</td>
<td>0</td>
<td>running</td>
<td>426</td>
<td>0</td>
</tr>
<tr>
<td>16:04:37</td>
<td>all</td>
<td>3321</td>
<td>2829</td>
<td>0</td>
<td>running</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>16:06:15</td>
<td>all</td>
<td>3321</td>
<td>2827</td>
<td>0</td>
<td>running</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>16:06:17</td>
<td>all</td>
<td>3321</td>
<td>2813</td>
<td>0</td>
<td>running</td>
<td>242</td>
<td>0</td>
</tr>
<tr>
<td>16:08:11</td>
<td>all</td>
<td>3321</td>
<td>2828</td>
<td>0</td>
<td>running</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>16:08:26</td>
<td>all</td>
<td>3321</td>
<td>2810</td>
<td>0</td>
<td>running</td>
<td>222</td>
<td>0</td>
</tr>
<tr>
<td>16:09:44</td>
<td>all</td>
<td>3321</td>
<td>2819</td>
<td>0</td>
<td>running</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>16:10:03</td>
<td>all</td>
<td>3321</td>
<td>2808</td>
<td>0</td>
<td>running</td>
<td>420</td>
<td>0</td>
</tr>
<tr>
<td>16:12:26</td>
<td>all</td>
<td>3321</td>
<td>2817</td>
<td>0</td>
<td>running</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>16:12:46</td>
<td>all</td>
<td>3321</td>
<td>2882</td>
<td>0</td>
<td>running</td>
<td>422</td>
<td>0</td>
</tr>
<tr>
<td>16:15:11</td>
<td>all</td>
<td>3321</td>
<td>2811</td>
<td>0</td>
<td>running</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>16:15:39</td>
<td>all</td>
<td>3321</td>
<td>2794</td>
<td>0</td>
<td>running</td>
<td>420</td>
<td>0</td>
</tr>
<tr>
<td>16:17:18</td>
<td>all</td>
<td>3321</td>
<td>2806</td>
<td>0</td>
<td>running</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>16:17:49</td>
<td>all</td>
<td>3321</td>
<td>2792</td>
<td>0</td>
<td>running</td>
<td>415</td>
<td>0</td>
</tr>
<tr>
<td>16:19:34</td>
<td>all</td>
<td>3321</td>
<td>2800</td>
<td>0</td>
<td>running</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>16:20:15</td>
<td>all</td>
<td>3321</td>
<td>2788</td>
<td>0</td>
<td>running</td>
<td>413</td>
<td>0</td>
</tr>
<tr>
<td>16:21:26</td>
<td>all</td>
<td>3321</td>
<td>2795</td>
<td>0</td>
<td>running</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>16:21:53</td>
<td>all</td>
<td>3321</td>
<td>2784</td>
<td>0</td>
<td>running</td>
<td>411</td>
<td>0</td>
</tr>
<tr>
<td>16:23:47</td>
<td>all</td>
<td>3321</td>
<td>2791</td>
<td>0</td>
<td>running</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>16:24:10</td>
<td>all</td>
<td>3321</td>
<td>2779</td>
<td>0</td>
<td>running</td>
<td>411</td>
<td>0</td>
</tr>
<tr>
<td>16:26:05</td>
<td>all</td>
<td>3321</td>
<td>2705</td>
<td>0</td>
<td>running</td>
<td>92</td>
<td>0</td>
</tr>
<tr>
<td>16:26:33</td>
<td>all</td>
<td>3321</td>
<td>2683</td>
<td>0</td>
<td>running</td>
<td>582</td>
<td>0</td>
</tr>
<tr>
<td>16:29:08</td>
<td>all</td>
<td>3321</td>
<td>2693</td>
<td>0</td>
<td>running</td>
<td>87</td>
<td>0</td>
</tr>
<tr>
<td>16:29:21</td>
<td>all</td>
<td>3321</td>
<td>2647</td>
<td>0</td>
<td>running</td>
<td>530</td>
<td>0</td>
</tr>
<tr>
<td>16:30:39</td>
<td>all</td>
<td>3321</td>
<td>2651</td>
<td>0</td>
<td>running</td>
<td>46</td>
<td>0</td>
</tr>
<tr>
<td>16:30:54</td>
<td>all</td>
<td>3321</td>
<td>2621</td>
<td>0</td>
<td>running</td>
<td>550</td>
<td>0</td>
</tr>
<tr>
<td>16:32:02</td>
<td>all</td>
<td>3321</td>
<td>2634</td>
<td>0</td>
<td>running</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>16:32:26</td>
<td>all</td>
<td>3321</td>
<td>2608</td>
<td>0</td>
<td>running</td>
<td>555</td>
<td>0</td>
</tr>
<tr>
<td>16:33:29</td>
<td>all</td>
<td>3321</td>
<td>2630</td>
<td>0</td>
<td>running</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>16:34:18</td>
<td>all</td>
<td>3321</td>
<td>2597</td>
<td>0</td>
<td>running</td>
<td>561</td>
<td>0</td>
</tr>
<tr>
<td>16:35:16</td>
<td>all</td>
<td>3321</td>
<td>2586</td>
<td>0</td>
<td>running</td>
<td>568</td>
<td>0</td>
</tr>
<tr>
<td>16:37:39</td>
<td>all</td>
<td>3321</td>
<td>2612</td>
<td>0</td>
<td>running</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>16:39:19</td>
<td>all</td>
<td>3321</td>
<td>2577</td>
<td>0</td>
<td>running</td>
<td>559</td>
<td>0</td>
</tr>
<tr>
<td>16:39:52</td>
<td>all</td>
<td>3321</td>
<td>2605</td>
<td>0</td>
<td>running</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>16:41:04</td>
<td>all</td>
<td>3321</td>
<td>2566</td>
<td>0</td>
<td>running</td>
<td>556</td>
<td>0</td>
</tr>
<tr>
<td>16:41:25</td>
<td>all</td>
<td>3321</td>
<td>2593</td>
<td>0</td>
<td>running</td>
<td>23</td>
<td>0</td>
</tr>
</tbody>
</table>
2. **Remote targets**

- Idea: work with remote files **as if they were local**
- Remote targets built on top of GFAL2 Python bindings
  - Supports all WLCG protocols (XRootD, WebDAV, GridFTP, dCache, SRM, ...) + DropBox
  - API **identical** to local targets
  
 ❗ Actual remote interface **interchangeable** (GFAL2 is just a good default, fsspec integration easily possible)
- Mandatory features: automatic retries, **local caching** *(backup)*, configurable protocols, round-robin, ...

---

**“FileSystem” configuration**

```yaml
# law.cfg
[wlcg_fs]
base: root://eosuser.cern.ch/eos/user/m/mrieger
...
```

- Base path prefixed to all paths using this “fs”
- Configurable per file operation (stat, listdir, ...)
- Protected against removal of parent directories
2. Remote targets

- Idea: work with remote files as if they were local
- Remote targets built on top of GFAL2 Python bindings
  - Supports all WLCG protocols (XRootD, WebDAV, GridFTP, dCache, SRM, ...) + DropBox
  - API identical to local targets
    - Actual remote interface interchangeable (GFAL2 is just a good default, fsspec integration easily possible)
- Mandatory features: automatic retries, local caching (backup), configurable protocols, round-robin, ...

Conveniently reading remote files

```python
# read a remote json file
target = law.WLCGFileTarget("/file.json", fs="wlcg_fs")

with target.open("r") as f:
    data = json.load(f)
```
2. Remote targets

- Idea: work with remote files as if they were local
- Remote targets built on top of GFAL2 Python bindings
  - Supports all WLCG protocols (XRootD, WebDAV, GridFTP, dCache, SRM, ...) + DropBox
  - API identical to local targets
  - Actual remote interface interchangeable (GFAL2 is just a good default, fsspec integration easily possible)
- Mandatory features: automatic retries, local caching (backup), configurable protocols, round-robin, ...

Conveniently reading remote files

```python
# read a remote json file
target = law.WLCGFileTarget("/file.json", fs="wlcg_fs")

# use convenience methods for common operations
data = target.load(formatter="json")
```
2. Remote targets

- Idea: work with remote files as if they were local
- Remote targets built on top of GFAL2 Python bindings
  - Supports all WLCG protocols (XRootD, WebDAV, GridFTP, dCache, SRM, ...) + DropBox
  - API identical to local targets
    - Actual remote interface interchangeable (GFAL2 is just a good default, fsspec integration easily possible)
- Mandatory features: automatic retries, local caching (backup), configurable protocols, round-robin, ...

Conveniently reading remote files

```python
# same for root files with context guard
target = law.WLCGFileTarget("/file.root", fs="wlcg_fs")

with target.load(formatter="root") as tfile:
    tfile.ls()
```
2. Remote targets

- Idea: work with remote files as if they were local
- Remote targets built on top of GFAL2 Python bindings
  - Supports all WLCG protocols (XRootD, WebDAV, GridFTP, dCache, SRM, ...) + Dropbox
  - API identical to local targets
    - Actual remote interface interchangeable (GFAL2 is just a good default, fsspec integration easily possible)
- Mandatory features: automatic retries, local caching (backup), configurable protocols, round-robin, ...

Conveniently reading remote files

```python
# multiple other "formatters" available
target = law.WLCGFileTarget("/model.pb", fs="wlcg_fs")

graph = target.load(formatter="tensorflow")
session = tf.Session(graph=graph)
```
3. **Environment sandboxing**

- Diverging software requirements between typical workloads is a great feature / challenge / problem

- Introduce sandboxing:
  - Run entire task in **different environment**

- Existing sandbox implementations:
  - Sub-shell with init file (e.g. for CMSSW)
  - Virtual envs
  - Docker images
  - Singularity images
Triggers: CLI, scripting and notebooks

- **CLI**
  - `law run Reconstruction --dataset ttbar --workflow htcondor`
  - Full auto-completion of tasks and parameters

- **Scripting**
  - Mix task completeness checks, job execution & input/output retrieval with custom scripts
  - Easy interface to existing tasks for prototyping

- **Notebooks**

```python
from analysis.tasks import Selection
import awkward as ak

def get_n_jets(events):
    n_jets = ak.num(events.Jet, axis=1)
    print(n_jets)
```

```
In [5]: %law run ShowFrequencies --print-status -l

print task status with max_depth -1 and target_depth 0

0 > ShowFrequencies(slow=False)
    1 > MergeCounts(slow=False)
        LocalFileTarget(fs=local_fs, path=${DATA_PATH}/chars_merged.json)
            existent
    2 > CountChars(file_index=1, slow=False)
        LocalFileTarget(fs=local_fs, path=${DATA_PATH}/chars_1.json)
            existent
    3 > FetchLoremIpsum(file_index=1, slow=False)
        LocalFileTarget(fs=local_fs, path=${DATA_PATH}/loremipsum_1.txt)
            existent
```
# reco.py

```python
import luigi

from my_analysis.tasks import Selection

class Reconstruction(luigi.Task):
    dataset = luigi.Parameter(default="ttH")

    def requires(self):
        return Selection(dataset=self.dataset)

    def output(self):
        return luigi.LocalTarget(f"reco_{self.dataset}.root")

    def run(self):
        inp = self.input()  # output() of requirements
        outp = self.output()

        # perform reco on file described by "inp" and produce "outp"
        ...
```

Example

```bash
$ python reco.py Reconstruction --dataset ttbar
```
# reco.py

```python
import luigi
import law
from my_analysis.tasks import Selection

class Reconstruction(law.Task):
    dataset = luigi.Parameter(default="ttH")

    def requires(self):
        return Selection(dataset=self.dataset)

    def output(self):
        return law.LocalFileTarget(f"reco_{self.dataset}.root")

    def run(self):
        inp = self.input()  # output() of requirements
        outp = self.output()

        # perform reco on file described by "inp" and produce "outp"
        ...
```

```bash
law run Reconstruction --dataset ttbar
```
# reco.py

```python
import luigi
import law
from my_analysis.tasks import Selection

class Reconstruction(law.Task, law.HTCondorWorkflow):
  dataset = luigi.Parameter(default="ttH")

  def requires(self):
    return Selection(dataset=self.dataset)

  def output(self):
    return law.LocalFileTarget(f"reco_{self.dataset}.root")

  def run(self):
    inp = self.input()  # output() of requirements
    outp = self.output()

    # perform reco on file described by "inp" and produce "outp"
    ...
```

```bash
> law run Reconstruction --dataset ttbar --workflow htcondor
```
```python
# reco.py
import luigi
import law
from my_analysis.tasks import Selection

class Reconstruction(law.Task, law.HTCondorWorkflow):
    dataset = luigi.Parameter(default="ttH")

    def requires(self):
        return Selection(dataset=self.dataset)

    def output(self):
        return law.WLCGFileTarget("reco_{self.dataset}.root")

    def run(self):
        inp = self.input()  # output() of requirements
        outp = self.output()

        # perform reco on file described by "inp" and produce "outp"
        ...
```

Example

$law run Reconstruction --dataset ttbar --workflow htcondor$
# reco.py

```python
import luigi
import law
from my_analysis.tasks import Selection

class Reconstruction(law.SandboxTask, law.HTCondorWorkflow):
    dataset = luigi.Parameter(default="ttH")
    sandbox = "docker::cern/cc7-base"

    def requires(self):
        return Selection(dataset=self.dataset)

    def output(self):
        return law.WLCGFileTarget("reco_{self.dataset}.root")

    def run(self):
        inp = self.input()  # output() of requirements
        outp = self.output()

        # perform reco on file described by "inp" and produce "outp"
        ...
```

Example

```bash
law run Reconstruction --dataset ttbar --workflow htcondor
```
Resource-agnostic workflow management **essential** for large & complex analyses

→ Need for a flexible **design pattern** to automate arbitrary workloads

→ **All** information transparently encoded through **tasks**, **targets** & **requirements**

→ **End-to-end automation** of analyses over distributed resources

→ Full decoupling of **run locations**, **storage locations** & **software environments**

→ Allows to build frameworks that check every point in the **CMS analysis wishlist** (mostly exp. agnostic)

→ [github.com/riga/law](https://github.com/riga/law), [law.readthedocs.io](https://law.readthedocs.io)

→ [github.com/spotify/luigi](https://github.com/spotify/luigi), [luigi.readthedocs.io](https://luigi.readthedocs.io)

**Collaboration & contributions welcome!**
Backup
Other "workflow" engines

- **Metrics for comparison**
  - Low-level array processing vs. high-level embedding
  - Pythonic usage
  - Usage Overhead (requires a DB, server, custom hardware, ...)
  - Built-in features
  - Configurability
  - ...
Existing WMS: MC production

Tailored systems
- Structure known in advance
- Workflows static & recurring
- One-dimensional design
- Special production infrastructure
- Homogeneous software requirements

Wishlist for end-user analyses
- Structure “iterative”, a-priori unknown
- Dynamic workflows, fast R&D cycles
- DAG with arbitrary dependencies
- Incorporate any existing infrastructure
- Use custom software, everywhere

→ Requirements for HEP analyses mostly orthogonal
A typical example: ML workflow with uncertainties

Nominal MC

- Reconstruction
- MVA Split
  - MVA Training
  - MVA Evaluation
- Inference

...
A typical example: ML workflow with uncertainties
A typical example: ML workflow with uncertainties

MC, Syst. 1

- Reconstruction
- MVA Split
- MVA Evaluation
- Inference

MC with systematic weights derived from nominal sample

train, test, evaluate
A typical example: ML workflow with uncertainties

MC, Syst. II

- Reconstruction
- MVA Split
  - train
  - test
  - weights
- MVA Training
- MVA Evaluation
- Inference

MC with systematic generated from new events
Hands-on!

- Print character frequencies in the "loremipsum" placeholder text (from examples/loremipsum)

  ➤ Fetch 6 paragraphs as txt files from some server
  ➤ Count character frequencies and save them in json
  ➤ Merge into a single json file
  ➤ Print frequencies

- for the notebook version

- Additional example: Workflow using CERN HTCondor
Things to try

- **Interactive parameters**
  - Append `--print-status RECURSION_LEVEL[,TARGET_LEVEL]`
  - Append `--print-deps RECURSION_LEVEL`
  - Append `--remove-output RECURSION_LEVEL[,MODE],[RESTART]`
  - Append `--fetch-output RECURSION_LEVEL[,MODE],[DIRECTORY]`

- **Parallelize**
  - Append `--workers 4`

- **Add a task**
  - **LinearizeChars**
    - Create an ordered string "aaaaabbccdddeeeeee..." from all existing characters and save it in a text file
Many tasks exhibit the same overall structure and/or purpose

- "Run over \( N \) existing files" / "Generate \( N \) events/toys" / "Merge \( N \) into \( M \) files"
- All these tasks can **profit from the same features**
  - "Only process file \( x \) and/to \( y \)"; "Remove outputs of \( x, y \& z \)"
  - "Process \( N \) files, but consider the task finished once \( M < N \) are done"; "..."

→ Calls for a generic container object that provides guidance and features for these cases

**Workflow "containers"**
- Task that introduces a parameters called \(--\text{branch } b\) (\texttt{luigi.IntParameter})
  - \( b \geq 0 \): Instantiates particular tasks called "branches"; \texttt{run()} will (e.g.) process file \( b \)
  - \( b = -1 \): Instantiates the workflow container itself; \texttt{run()} will run* all branch tasks
    * How branch tasks are run is implemented in different workflow types: local or several remote ones

**Practical advantages**
- Convenience: same features available in all workflows (see next slides)
- **Scalability and versatility for remote workflows**
  - Jobs: Better control of jobs, submission, task-to-job matching ... (see next slides)
  - Luigi: Central scheduler breaks when pinged by \( O(10k) \) tasks every few seconds
  - Remote storage: Allows batched file operations instead of file-by-file requests
- Tasks that each write a single character into a text file
- Character assigned to them though the branch map as their "branch data"

```python
import luigi
import law

from my_analysis.tasks import AnalysisTask

class WriteAlphabet(AnalysisTask, law.LocalWorkflow):
    def create_branch_map(self):
        chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        return dict(enumerate(chars))

    def output(self):
        return law.LocalFileTarget(f"char_{self.branch}.txt")

    def run(self):
        # branch_data refers to this branch's value in the branch map
        self.output().dump(f"char: {self.branch_data}", formatter="txt")
```
6 remote workflow implementations come with law
- htcondor, glite, lsf, arc, slurm, cms-crab (in PR#150)
- Based on generic "job manager" implementations in contrib packages

Job managers fully decoupled from most law functionality
- Simple extensibility
- No "auto-magic" in submission files, rather minimal and configurable through tasks
- Usable also without law

Most important features
- Job submission functionality "declared" via task class inheritance
- Provision of software and job-specific requirements through workflow_requires()
- Control over remote jobs through parameters:
  - --branch --branches: granular control of which tasks to process
  - --acceptance --tolerance: defines when a workflow is complete / failed
  - --poll-interval --walltime: controls the job status polling interval and runtime
  - --tasks-per-job --parallel-jobs: control of resource usage at batch systems
Effective remote targets — "Localization"

```python
# coding: utf-8
# flake8: noqa

import luigi
import law

from my_analysis.tasks import Selection
from my_analysis.algorithms import awesome_reconstruction

class Reconstruction(law.Task):
    def requires(self):
        return Selection.req(self)

    def output(self):
        return law.wlcm.WLCGFileTarget("/some/remote/path.parquet")

    def run(self):
        # !!!
        # awesome reconstruction is expecting local paths

        with self.input().localize("r") as inp:
            with self.output().localize("w") as outp:
                awesome_reconstruction(inp.path, outp.path)
```
```
# coding: utf-8
# flake8: noqa

import luigi
import law

from my_analysis.tasks import Selection
from my_analysis.algorithms import awesome_reconstruction

class Reconstruction(law.Task):
    def requires(self):
        return Selection.req(self)

    def output(self):
        return law.wlcg.WLCGFileTarget("/some/remote/path.parquet")

    @law.decorator.localize
def run(self):
        # !!!
        # awesome reconstruction is expecting local paths
        # but that's ok since the decorator does the localization
        awesome_reconstruction(self.input().path, self.output().path)
```
● Local cache for remote targets

Remote storage

Selection

Reconstruction

Save ✓

Load ?

No!
Effective remote targets — Caching

- Local cache for remote targets

Selection ➔ Reconstruction

Local cache

Remote storage

- save ✓
- sync ✓
- open ✓
Effective remote targets — Caching

- **Local cache for remote targets**
  - When enabled, all operations on remote targets are cached

- **Simple configuration**
  - When enabled, all operations on remote targets are cached

```
[wlcg_fs]
base: root://eosuser.cern.ch/eos/user/m/mrieger/myproject
use_cache: True
cache_root: /tmp/mrieger/wlcg_fs_cache
cache_max_size: 10GB
```
Consider this example again

- `law run Reconstruction --dataset ttbar --workflow htcondor`
- $O(500 - 4k)$ files, stored either locally or remotely
- Any workflow engine will first check if things need to be rerun
  - $O(500 - 4k)$ file requests (via network)!
  - Prepare for admins to find you 😢

What `law` does

- Reconstruction is a workflow
- Workflows output a so-called `TargetCollection`'s, containing all outputs of its branch tasks
- `TargetCollection`'s can check if their files are located in the same directory
- If they do, perform a single (remote) `listdir` and compare basenames \( \rightarrow \) single request

There is no free lunch

- Our HEP resources (clusters, grid, storage elements, software environments) are very inhomogeneous
- A realistic workflow engine
  - can make some good, simple assumptions based on known best-practices
  - it should always allow users to transparently change decisions & configure every single aspect!
Abstraction: analysis workflows

- Workflow, decomposable into particular workloads
- Workloads related to each other by common interface
  - In/outputs define directed acyclic graph (DAG)
- Alter default behavior via parameters
- Computing resources
  - Run location (CPU, GPU, WLCG, ...)
  - Storage location (local, dCache, EOS, ...)
- Software environment
- Collaborative development and processing
- Reproducible intermediate and final results

→ Reads like a checklist for analysis workflow management
(Remote) targets
import law

from my_analysis import SomeTaskWithROOTOutput, some_executable

law.contrib.load("wlcg")

class MyTask(law.Task):
    def requires(self):
        return SomeTaskWithROOTOutput.req(self)

    def output(self):
        return law.wlcg.WLCGFileTarget("large_root_file.root")

    def run(self):
        # using target formatters for loading and dumping
        with self.input().load(formatter="uproot") as in_file:
            with self.output().dump(formatter="root") as out_file:
                ...

        # using localized representation of (e.g.) output
        # to use its local path for some executable
        # (the referenced file is automatically moved to the
        # remote location once the context exits)
        with self.output().localize("w") as tmp_output:
            some_executable(tmp_output.path)

@law.decorator.localize
    def run(self):
        # when wrapped by law.decorator.localize
        # self.input() and self.output() returns localized
        # representations already and deals with subsequent copies
        some_executable(self.output().path)
Remote target implementation details

**Target**
- **FileTarget**
  - fs: FileSystem
- **RemoteFileTarget**
  - fs: RemoteFileSystem
- **WLCGFileTarget**
  - no extra functionality

**FileSystem**
- std. methods: stat, touch, exists, remove, listdir, ...

**RemoteFileSystem**
- file_interface_cls
- file_interface instance

**RemoteFileInterface**
- implements atomic file interactions

**GFALFileInterface**
- access through gfal2

"is"
"has"
1. Need to access file “a.root” (has unique, path-dep. hash X)
2. Stat file “a.root”
3. File “a.root” with hash X in cache with latest mtime? → no
4. Remote request
5. Store “a.root” using hash X
6. Change mtime of file to value from stat (see 2)
7. Return local path in cache
8. Work with local file

Local caching (1)

Remote storage (e.g. eos / dcache / ...)

Remote request

Local request

Remote

Local machine

PWD

/tmp

law/python process

Local cache

Configuration
Local caching (2)

1. Need to access file “a.root” (has unique, path-dep. hash X)
2. Stat file “a.root”
3. File “a.root” with hash X in cache with latest mtime? → yes
4. Return local path in cache
5. Work with local file

Remote storage (e.g. eos / dcache / ...)

Remote request

Local request

Remote

Local machine

PWD

/local

/stat

Work with local file

Remote machine

Local cache

/configuration

law/python process

Remote request
Workflows
Many tasks exhibit the same overall structure and/or purpose
- "Run over N existing files" / "Generate N events/toys" / "Merge N into M files"
- All these tasks can profit from the same features
  - "Only process file x and/to y", "Remove outputs of "x, y & z",
    "Process N files, but consider the task finished once M < N are done", "..."

→ Calls for a generic container object that provides guidance and features for these cases

Workflow "containers"
- Task that introduces a parameters called --branch b (luigi.IntParameter)
  - b >= 0: Instantiates particular tasks called "branches"; run() will (e.g.) process file b
  - b = -1: Instantiates the workflow container itself; run() will run* all branch tasks
    * How branch tasks are run is implemented in different workflow types: local or several remote ones

Practical advantages
- Convenience: same features available in all workflows (see next slides)
- Scalability and versatility for remote workflows
  - Jobs: Better control of jobs, submission, task-to-job matching ... (see next slides)
  - Luigi: Central scheduler breaks when pinged by O(10k) tasks every few seconds
  - Remote storage: allows batched file operations instead of file-by-file requests
class Workflow(law.BaseTask):

    branch = luigi.IntParameter(default=-1)

@property
def is_workflow(self):
    return self.branch == -1

def branch_tasks(self):
    return [self.req(self, branch=b) for b in self.create_branch_map()]

def workflow_requires(self):
    """requirements to be resolved before the workflow starts ""

def workflow_output(self):
    """output of the workflow (usually a collection of branch outputs) ""

def workflow_run(self):
    """run implementation ""

def create_branch_map(self):
    """Maps branch numbers to arbitrary payloads, e.g.
    \``return {0: "file_A.txt", 1: "file_C.txt", 2: ...}``
    To be implemented by inheriting tasks.
    ""
    raise NotImplemented

def requires(self):
    """usual requirement definition ""

def output(self):
    """usual output definition ""

def run(self):
    """usual run implementation """
Tasks that each write a single character into a text file
Character assigned to them though the branch map as their "branch data"

```python
import luigi
import law

from my_analysis.tasks import AnalysisTask

class WriteAlphabet(AnalysisTask, law.LocalWorkflow):
    def create_branch_map(self):
        chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        return dict(enumerate(chars))

    def output(self):
        return law.LocalFileTarget(f"char_{self.branch}.txt")

    def run(self):
        # branch_data refers to this branch's value in the branch map
        self.output().dump(f"char: {self.branch_data}", formatter="txt")
```
6 remote workflow implementations come with law

- htcondor, glite, lsf, arc, slurm, cms-crab (in PR#150)
- Based on generic "job manager" implementations in contrib packages

Job managers fully decoupled from most law functionality

- Simple extensibility
- No "auto-magic" in submission files, rather minimal and configurable through tasks
- Usable also without law

Most important features

- Job submission functionality "declared" via task class inheritance
- Provision of software and job-specific requirements through workflow_requires()
- Control over remote jobs through parameters:
  - --branch --branches: granular control of which tasks to process
  - --acceptance --tolerance: defines when a workflow is complete / failed
  - --poll-interval --walltime: controls the job status polling interval and runtime
  - --tasks-per-job --parallel-jobs: control of resource usage at batch systems
Miscellaneous
Package structure

Command-line interface
- Job interface
  - Job file factory interface
  - Job manager interface
  - Generic remote job script

Target definitions
- Generic + file interface
- Local target impl.
- Remote target interfaces

Config parsing & tools
Task decorators
Custom loggers
Notification tools (for e.g. slack/telegram)
Custom parameters
Utilities & helpers

3rd party tools

Sandboxing mechanism
- Sandbox task
- Sandbox interface
- Bash sandbox impl.

Base task definitions

Base workflow definition
- Local workflow impl.
- Remote workflow interface

Lightweight patches of luigi, e.g.:
- Disable dep. checks in sandboxes
- Colorize logs
  → Could be added directly to luigi
Command-line interface

Job interface
- Job file factory interface
- Job manager interface
- Generic remote job script

Target definitions
- Generic + file interface
- Local target impl.
- Remote target interfaces

Config parsing & tools
Task decorators
Custom loggers
Notification tools (for e.g. slack/telegram)
Custom parameters
Utilities & helpers

Lightweight patches of luigi, e.g.:
- Disable dep. checks in sandboxes
- Colorize logs
→ Could be added directly to luigi
Workload

Management layer
- Run location
- Storage location
- Environment

Analysis layer
- Algorithm code

Input

Workflow (DAG)
- **law - luigi analysis workflow**
  - Repository: [github.com/riga/law](https://github.com/riga/law)
  - Documentation: [law.readthedocs.io](https://law.readthedocs.io) (in preparation)
  - Minimal example: [github.com/riga/law/tree/master/examples/loremipsum](https://github.com/riga/law/tree/master/examples/loremipsum)
  - HTCondor example: [github.com/riga/law/tree/master/examples/htcondor_at_cern](https://github.com/riga/law/tree/master/examples/htcondor_at_cern)
  - Contact: [Marcel Rieger](mailto:marcel.rieger@cern.ch)

- **luigi - Powerful Python pipelining package (by Spotify)**
  - Repository: [github.com/spotify/luigi](https://github.com/spotify/luigi)
  - Documentation: [luigi.readthedocs.io](https://luigi.readthedocs.io)
  - “Hello world!”: [github.com/spotify/luigi/blob/master/examples/hello_world.py](https://github.com/spotify/luigi/blob/master/examples/hello_world.py)

- **Technologies**
  - Docker: [docker.com](https://docker.com)
  - Singularity: [singularity.lbl.gov](https://singularity.lbl.gov)
columnflow
- **columnflow**: Backend for large-scale columnar analyses
  - Reads and writes columns only if necessary
  - Creates new columns and merges with existing ones at the **latest possible instance**

- Stores intermediate outputs for
  - computations downstream
  - sharing results of same computations across groups
  - applications requiring per-event info (ML)
  - studies done by students
  - debugging purposes
  - difference to map-reduce pattern in coffea processors

- Heavy use of bare NumPy & TensorFlow & awkward, plus coffea NanoScheme behavior

- Full resolution of systematic uncertainties (next slide)
- Checks 15/17 points of the CMS analysis wishlist in the ATTF report
Example: Resolution of systematic uncertainties
Example: Resolution of systematic uncertainties

Initial tasks

Final results

"nominal"
Example: Resolution of systematic uncertainties

Initial tasks

Final results

"nominal"   "tune(up|down)"   "jec(up|down)"   "pileup(up|down)"   ...

Used in training?
Example: Resolution of systematic uncertainties

**Key idea**
Tasks *know* which uncertainties

- *they implement*
- *they depend on* (through upstream tasks)
Example: Resolution of systematic uncertainties

Key idea

Tasks know which uncertainties
- they implement
- they depend on (through upstream tasks)
Example: Resolution of systematic uncertainties

Key idea

Tasks know which uncertainties

- they implement
- they depend on (through upstream tasks)

Initial tasks

- Reuses all "nominal" outputs above
- SelectEvents

Final results

- "nominal"
- "tune(up|down)"
- "jec(up|down)"
- "pileup(up|down)"

Tasks involved:
- CreateCutflowHistograms
- ReduceEvents
- ProduceColumns
- MergedReducedEvents
- MergeReductionStats
- MergeSelectionMasks
- MergeSelectionStats
- CreateHistograms
- MLTraining
- MergeMLEvents
- PrepareMLEvents
- MergeHistograms
- MergeShiftedHistograms
- PlotCutflowVariables1d
- PlotCutflowVariables2d
- PlotShiftedVariables1d
- PlotVariables2d
- PlotVariables1d
- WritePyhfWorkspace
- WriteDatacards
Example: Resolution of systematic uncertainties

Key idea

Tasks *know* which uncertainties

- *they implement*
- *they depend on* (through upstream tasks)