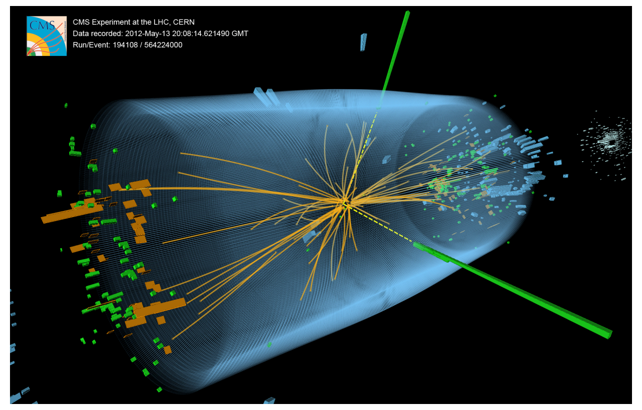
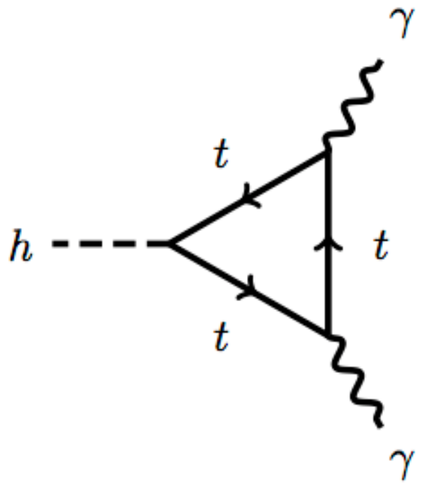
Design and Execution of make-like distributed Analyses based on Spotify's Pipelining Package Luigi

> make



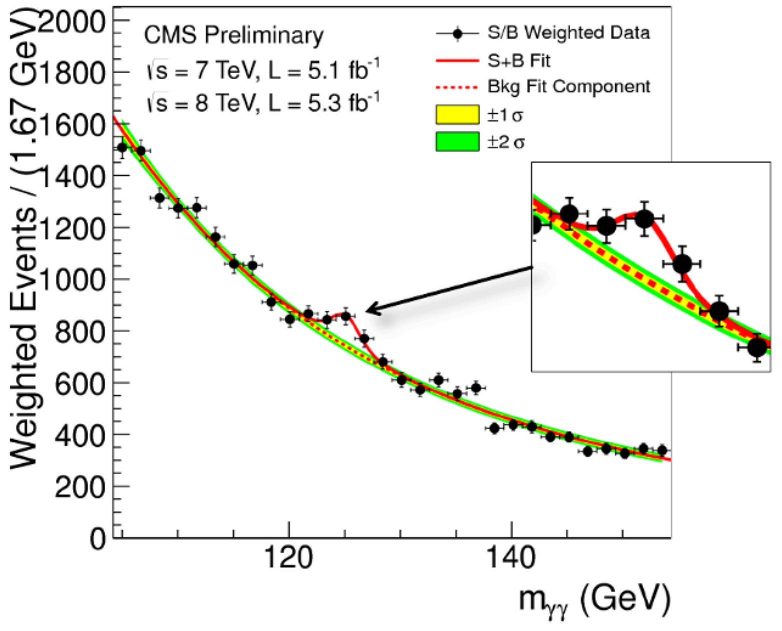
experiment

+

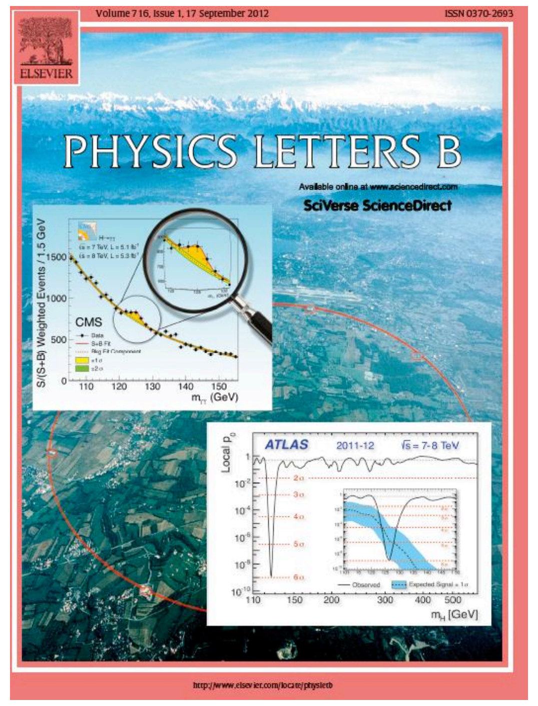


theory

→



→



Marcel Rieger*,
Martin Erdmann, Benjamin Fischer, Robert Fischer

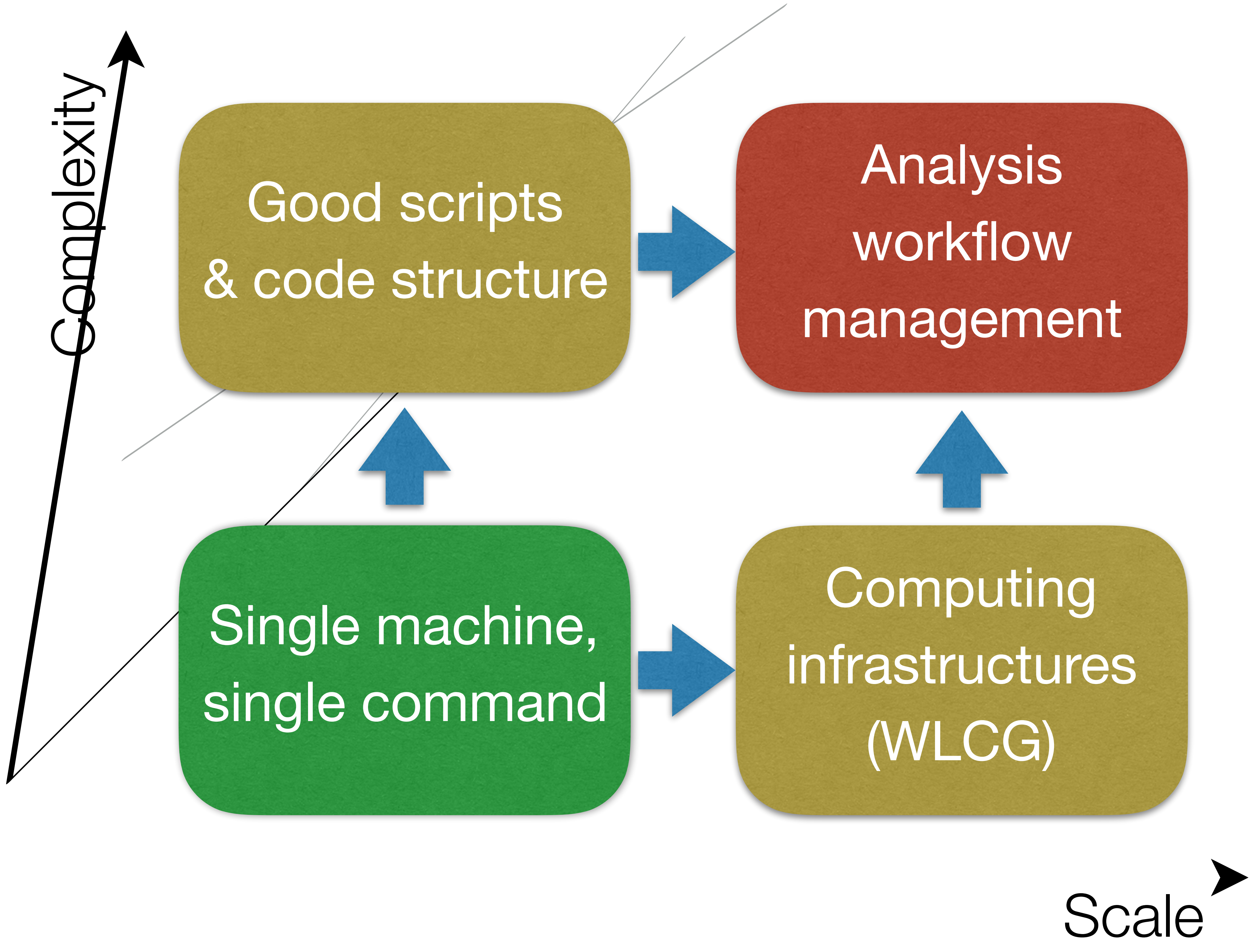
12/10/2016

III. Physikalisches Institut
RWTH AACHEN UNIVERSITY

Landscape of Analyses in HEP

- Scale: measure of resource consumption and amount of data
- Complexity: measure of granularity and inhomogeneity of workloads

- Future analyses likely to be large *and* complex, bottlenecks:
  - Entangled and undocumented structure & requirements between workloads, only exists in the "physicist's head"
  - Bookkeeping of code, data, versions, …
  - Manual execution and steering of jobs
  - Error-prone & time-consuming

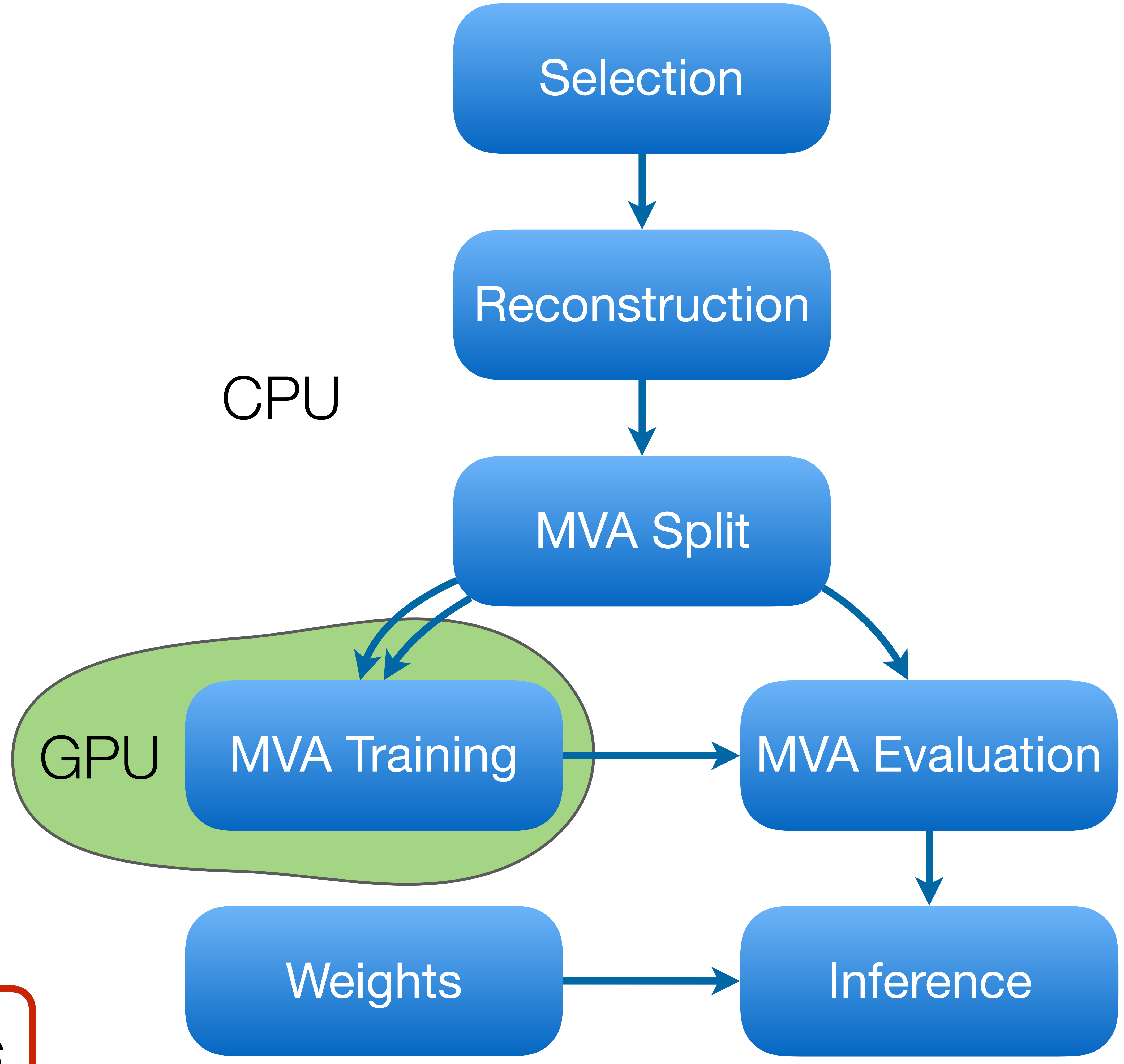


→ *Analysis workflow management essential for future measurements!*

Abstraction: HEP Analysis

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

→ *Large overlap with features of workflow systems*



example

Comparison of Workflow Management Systems (WMS)

	Existing WMS e.g. MC Management	Generic Analysis WMS
Development Process	final objective **known in advance**	**iterative**, final composition **a priori unknown**
Workflow Structure	**chain structure,** mostly one-dimensional	**tree structure,** arbitrarily branched
Evolution	**static** over time, recurrent execution	**dynamic,** fast R&D cycles
Infrastructure	**specially tailored,** e.g. storage systems, DBs	**incorporate existing,** quickly adapt to changes
Applicability	**tuned** to particular use case	**flexible**, able to model **every possible workflow**

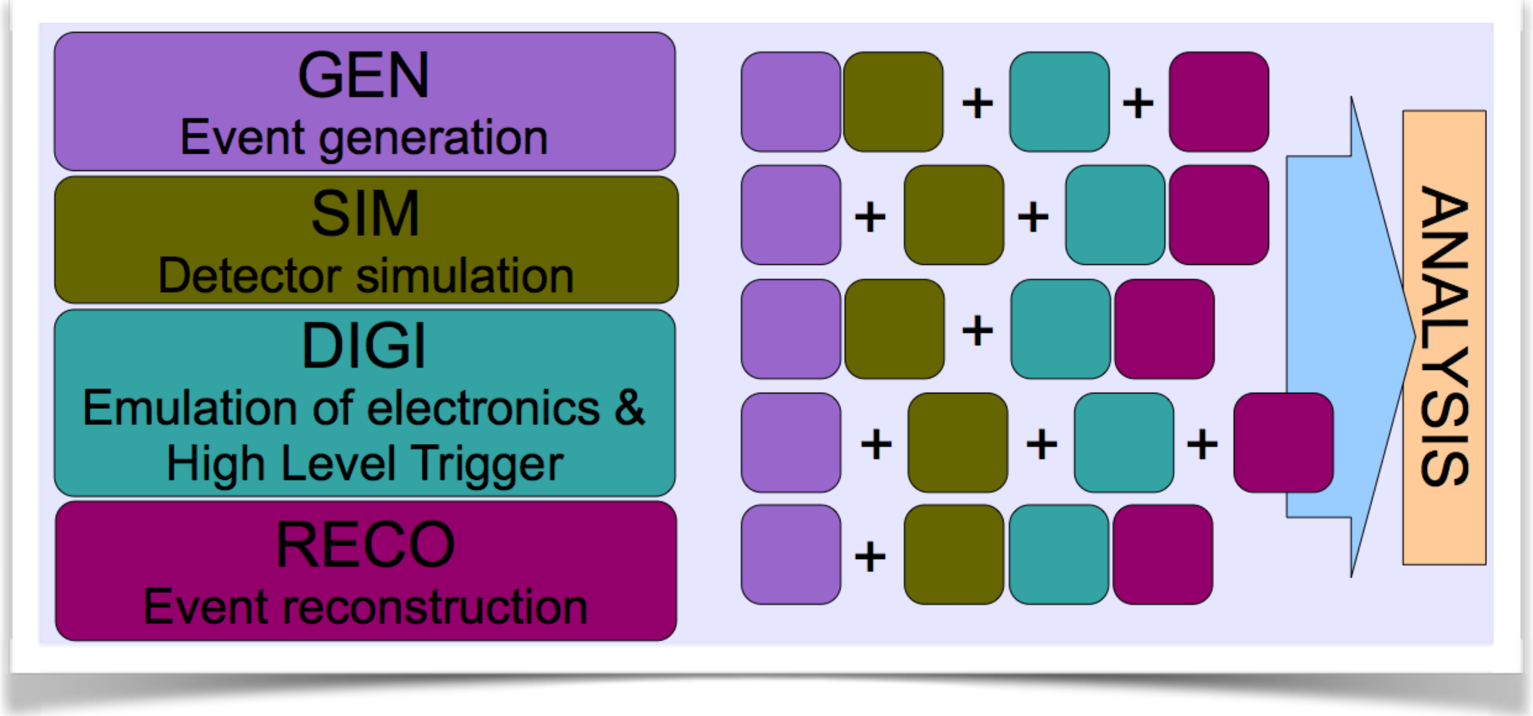
→ Existing WMS highly specialized for designated use case

→ Requirements for HEP analyses mostly orthogonal

→ *Toolbox for flexible analysis conception*

Typical analysis *tree*:

Typical MC *chain*:



Luigi

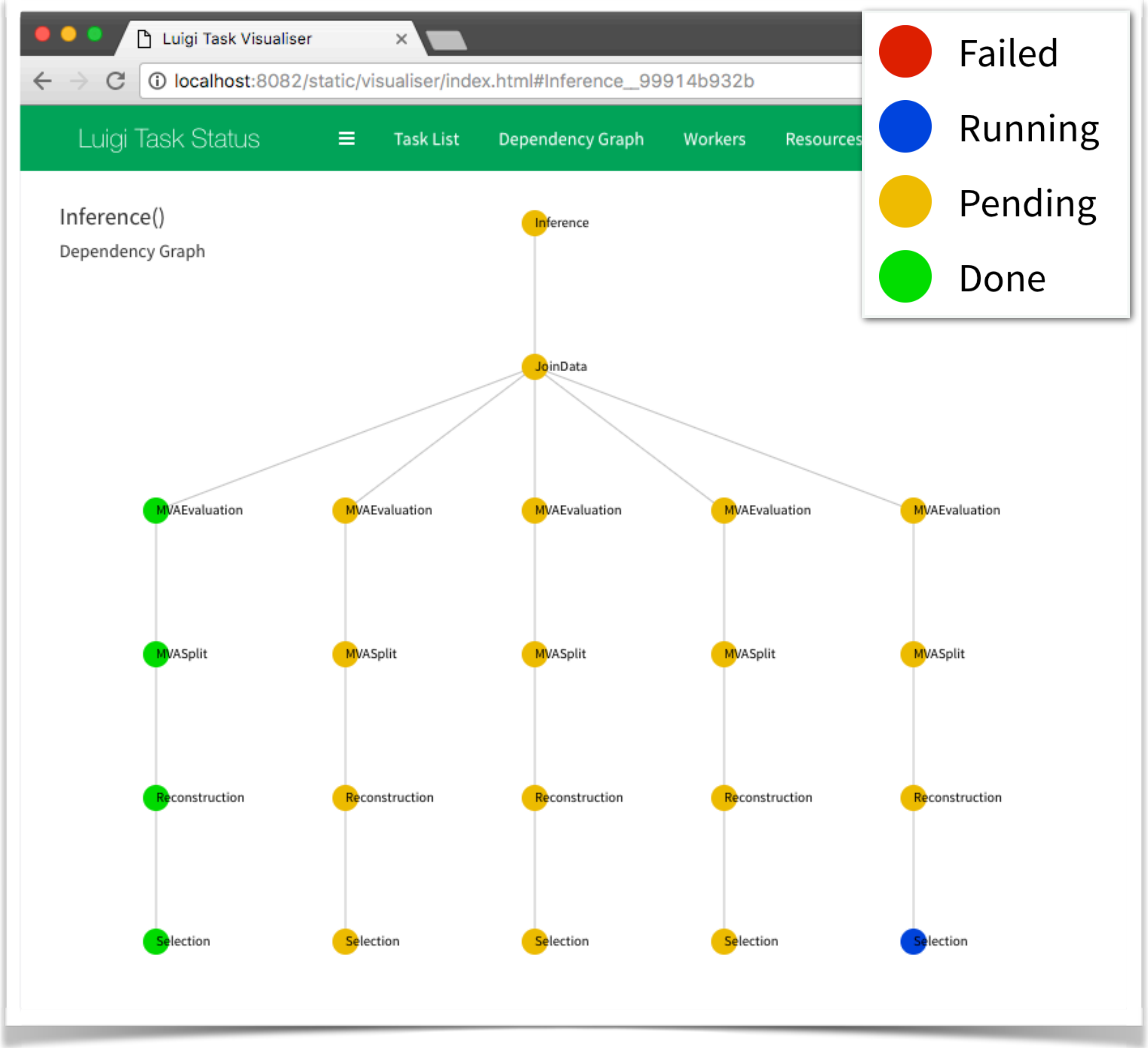
- Python package for building complex pipelines
- Development started at Spotify, now open-source and community driven

Building blocks

1. Workloads defined as *Task* classes
2. Tasks *require* other tasks & output *Targets*
3. *Parameters* customize and control task behavior

- Task execution → builds up dependency tree, only computes what is necessary
- Web interface, error handling, command line tools, collaborative features, …

→ *Suitable tool to manage complexity*



III. Physikalisches Institut RWTH AACHEN UNIVERSITY

Luigi

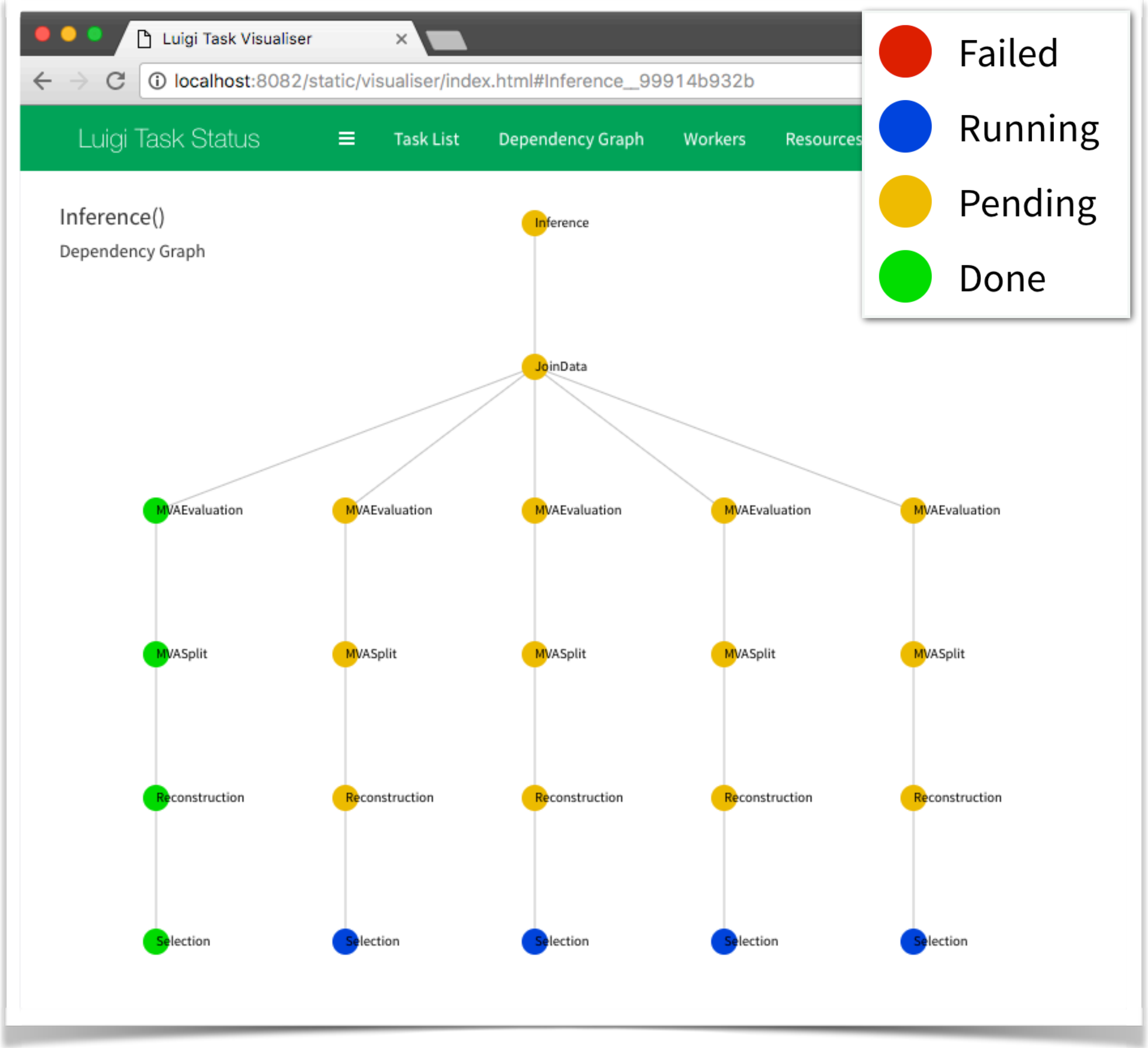
- Python package for building complex pipelines
- Development started at Spotify, now open-source and community driven

Building blocks

1. Workloads defined as *Task* classes
2. Tasks *require* other tasks & output *Targets*
3. *Parameters* customize and control task behavior

- Task execution → builds up dependency tree, only computes what is necessary
- Web interface, error handling, command line tools, collaborative features, …

→ Suitable tool to manage complexity



Luigi

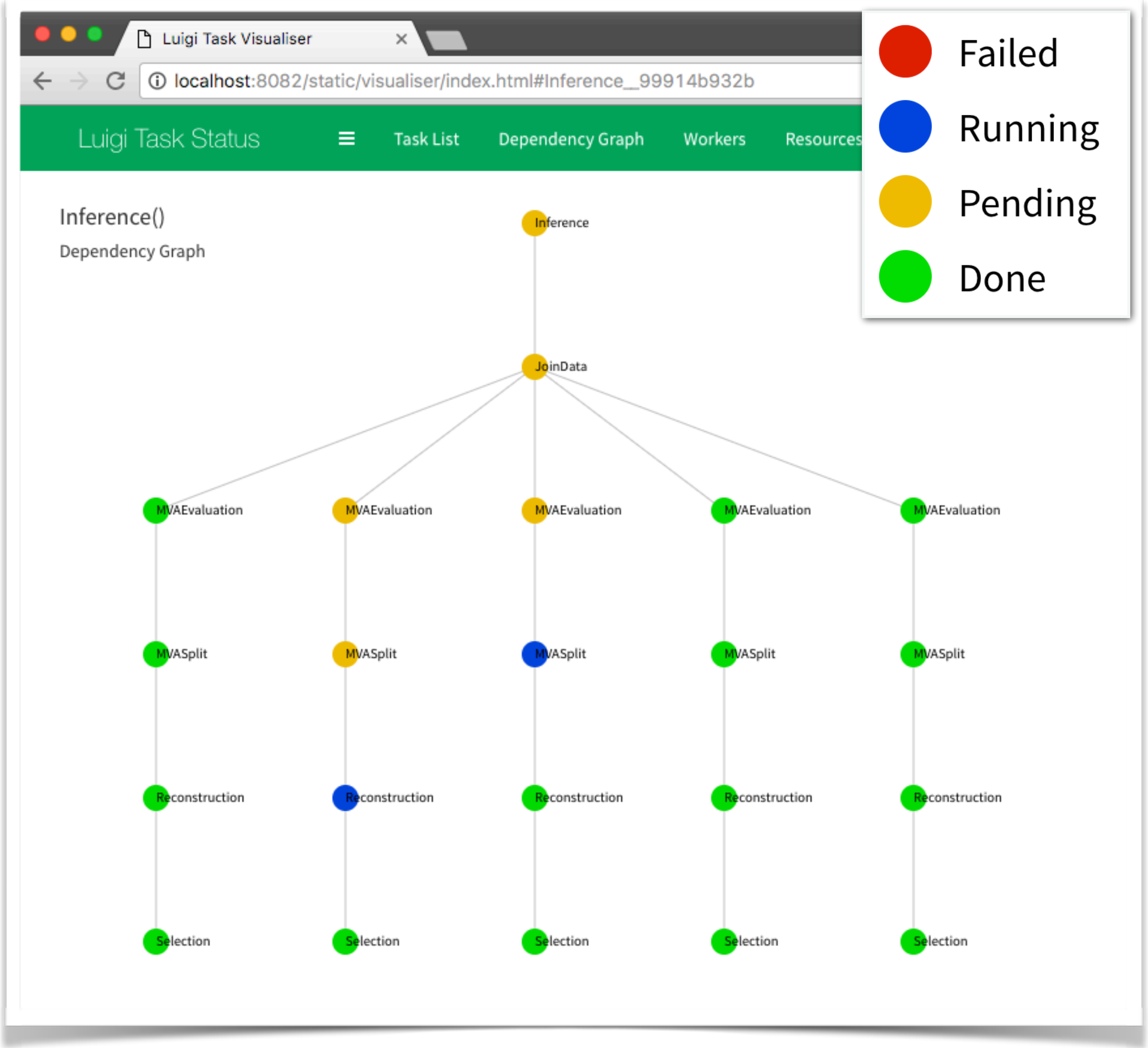
- Python package for building complex pipelines
- Development started at Spotify, now open-source and community driven

<u>Building blocks</u>

1. Workloads defined as *Task* classes
2. Tasks ***require*** other tasks & output *Targets*
3. *Parameters* customize and control task behavior

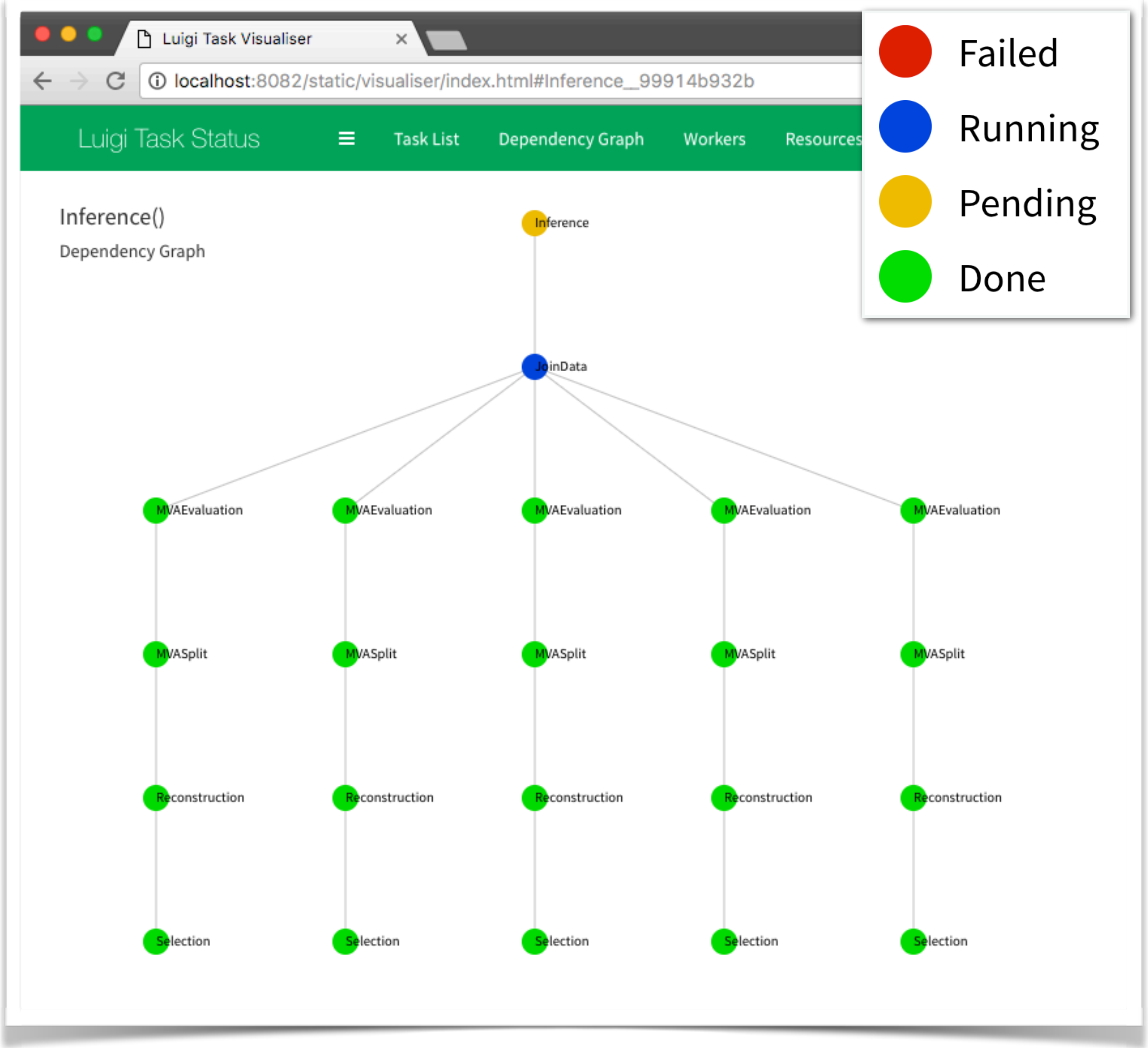
- Task execution → builds up dependency tree, only computes what is necessary
- Web interface, error handling, command line tools, collaborative features, …

→ *Suitable tool to manage complexity*



III. Physikalisches Institut
RWTH AACHEN UNIVERSITY

- Python package for building complex pipelines
- Development started at Spotify, now open-source and community driven



<u>Building blocks</u>

1. Workloads defined as *Task* classes
2. Tasks ***require*** other tasks & output *Targets*
3. *Parameters* customize and control task behavior

- Task execution → builds up dependency tree, only computes what is necessary
- Web interface, error handling, command line tools, collaborative features, …

→ *Suitable tool to manage complexity*

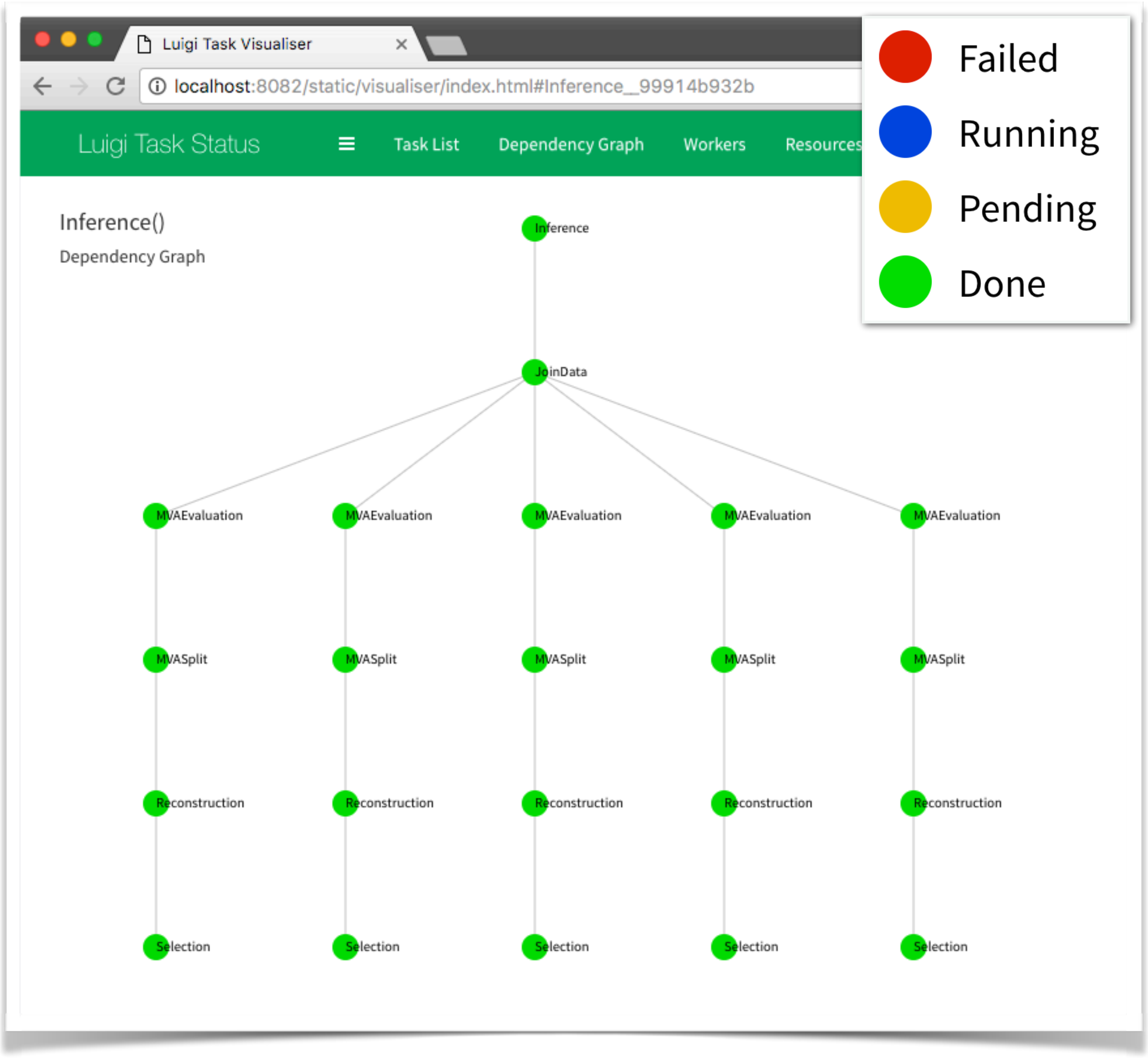
- Python package for building complex pipelines
- Development started at Spotify, now open-source and community driven

<u>Building blocks</u>

1. Workloads defined as *Task* classes
2. Tasks ***require*** other tasks & output ***Targets***
3. ***Parameters*** customize and control task behavior

- Task execution → builds up dependency tree, only computes what is necessary
- Web interface, error handling, command line tools, collaborative features, …

→ ***Suitable tool to manage complexity***



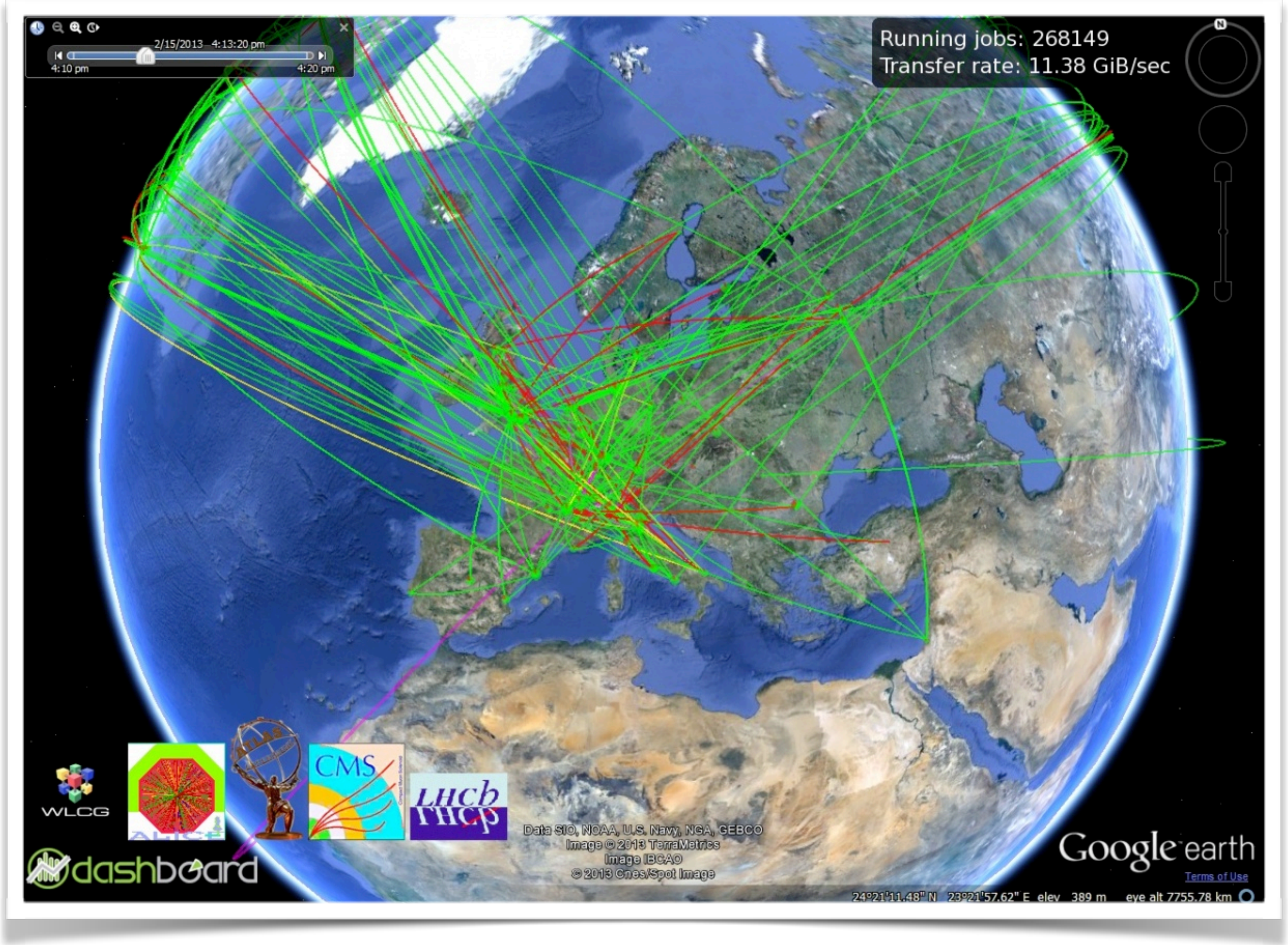
III. Physikalisches Institut
RWTH AACHEN UNIVERSITY

Adding Scalability: Luigi and the WLCG

- Example for implementation of abstract run & storage locations

GFAL2

1. Submit tasks as jobs to *computing elements*
   - Simple usage, transparent Luigi integration
   - Actual run location (local, CE) not hard-coded, decision made at execution time
   - Mandatory features like pilot jobs, automatic resubmission, or batch submission



2. Store targets on ***storage elements*** (e.g. dCache)
   - Built on top of GFAL2 Python bindings, transparent Luigi integration
   - Mandatory features like automatic retries, local caching, or batch transfers

→ *WLCG implementations provide scalability in the HEP context*

Adding Scalability: Luigi and the WLCG

- Example for implementation of abstract run & storage locations

1. Submit tasks as jobs to *computing elements*
   - Simple usage, transparent Luigi integration
   - Actual run location (local, CE) not hard-coded, decision made at execution time
   - Mandatory features like pilot jobs, automatic resubmission, or batch submission
2. Store targets on ***storage elements*** (e.g. dCache)
   - Built on top of GFAL2 Python bindings, transparent Luigi integration
   - Mandatory features like automatic retries, local caching, or batch transfers

GFAL2

```
> pyl Reconstruction --v test1 --local
> pyl Reconstruction --v prod1 --ce RWTH
```

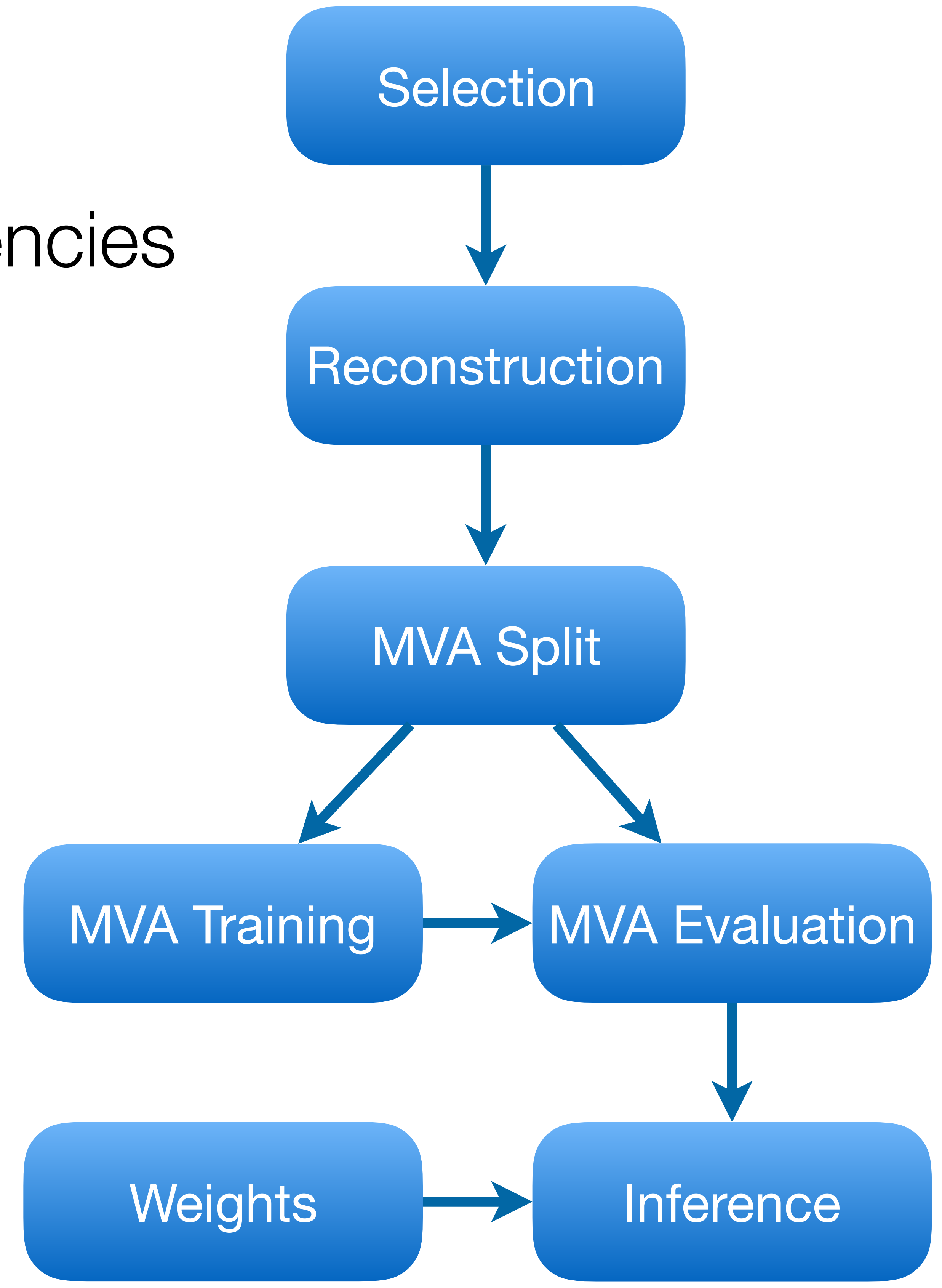
```
target = DCacheTarget("/path/to/file.txt")

with target.open("w") as f:
    f.write("some result")
```

→ *WLCG implementations provide scalability in the HEP context*

III. Physikalisches Institut
RWTH AACHEN UNIVERSITY

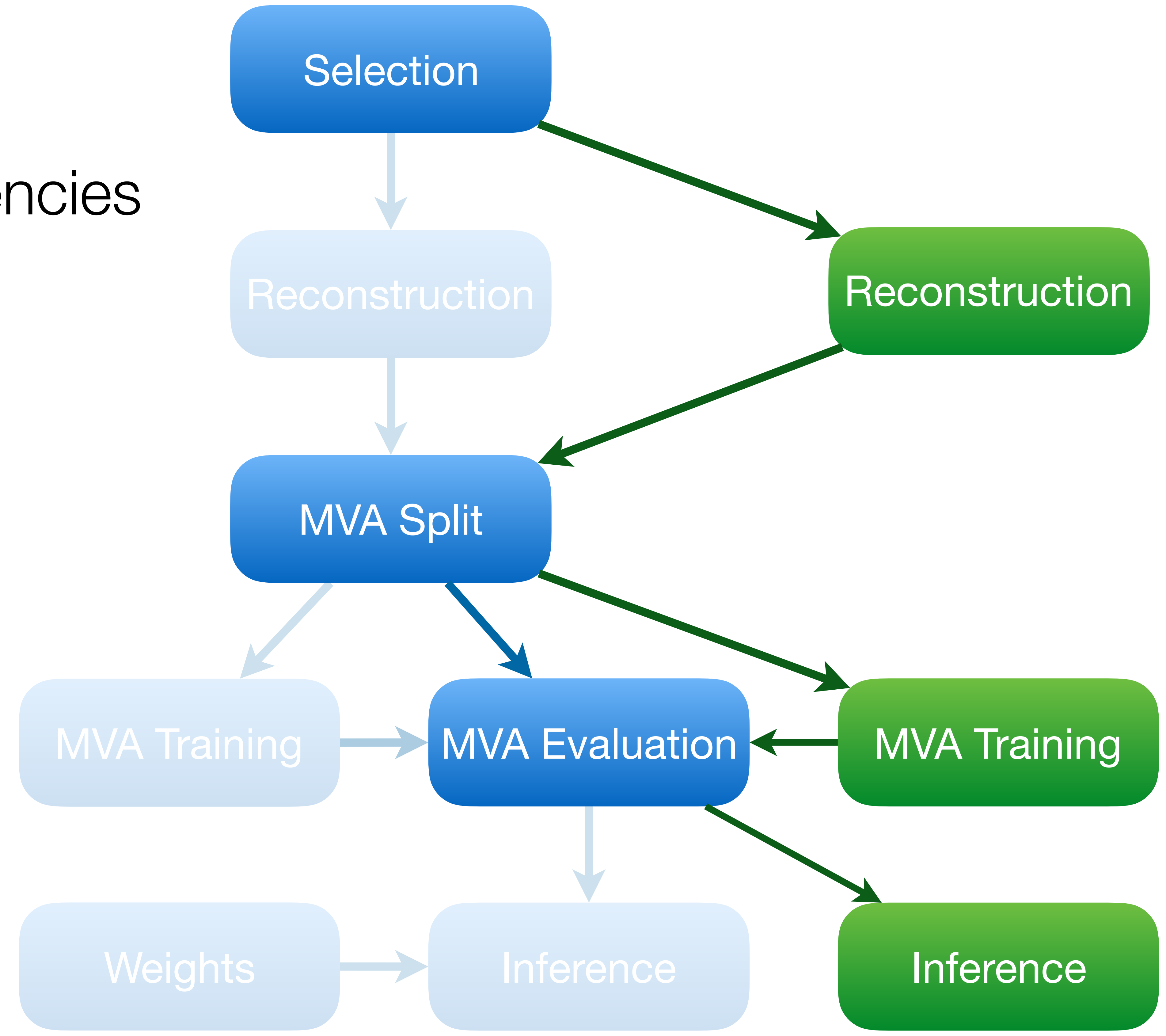
Direct Consequences and Benefits

- Toolbox providing building blocks for analyses, *not a framework*
  - → Permissive, non-restrictive *design pattern*
    (e.g. no constraint on language or data structure)
- *All* information transparently encoded in tasks, targets & dependencies
  - → Results *reproducible* by developer, groups, reviewers, …
  - → Documentary benefits, enables *analysis preservation*
- `make`-like execution across distributed resources
  - → Reduces overhead of manual management
  - → Improves cycle times & error-proneness
- Expansion of the concept of *collaboration*
  - → Clear structure lowers entry barrier
  - → Modularization allows re-use of tasks & intermediate results



Direct Consequences and Benefits

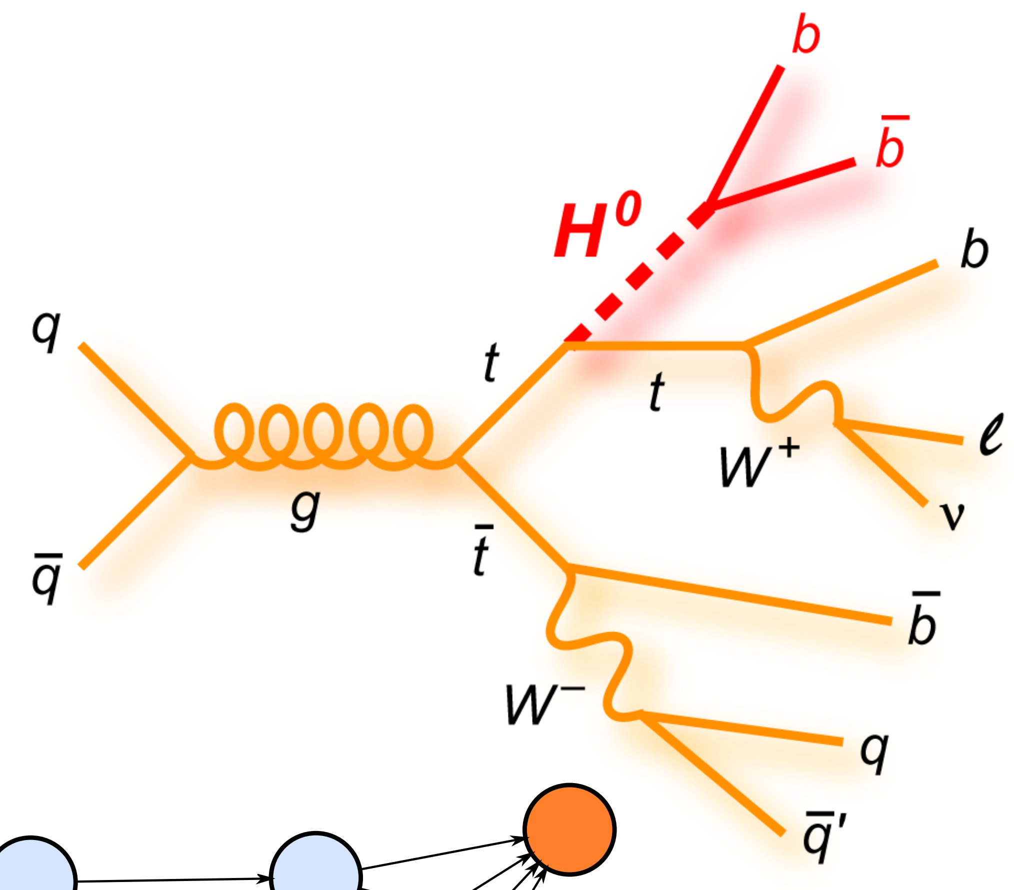
- Toolbox providing building blocks for analyses, *not a framework*
  - → Permissive, non-restrictive *design pattern*
    (e.g. no constraint on language or data structure)
- *All* information transparently encoded in tasks, targets & dependencies
  - → Results *reproducible* by developer, groups, reviewers, …
  - → Documentary benefits, enables *analysis preservation*
- `make`-like execution across distributed resources
  - → Reduces overhead of manual management
  - → Improves cycle times & error-proneness
- Expansion of the concept of *collaboration*
  - → Clear structure lowers entry barrier
  - → Modularization allows re-use of tasks & intermediate results



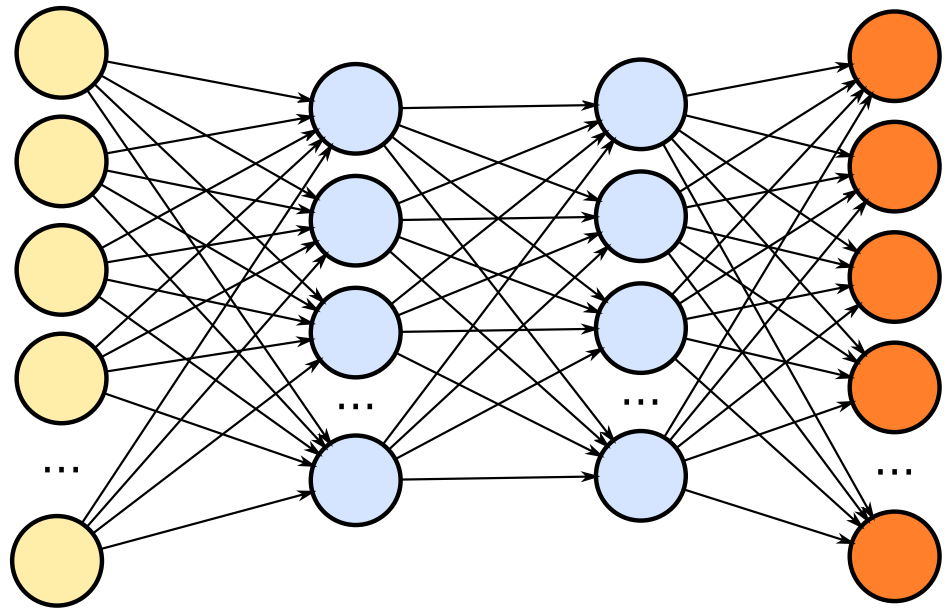
III. Physikalisches Institut
RWTH AACHEN UNIVERSITY

Example Application: ttH Analysis @ CMS

- *ttH*: measurement of Higgs ↔ quark Yukawa coupling
  - large-scale: ~30k input files, ~50 TB of storage, ~1000 unique Tasks
  - complex: irreducible backgrounds, ~40 systematic variations, DeepLearning/BDTs, multiple categorization schemes



- Run locations: 7 CEs, local machines, GPU machines
- Storage locations: 2 SEs (dCache), local disk, Dropbox, CERNBox



- Aware of entire workflow structure at all times, fast evaluation & revision

- Group of 5 people, clear allocation of duties and their interface
- Yet, entire analysis ***operable by everyone*** at all times

- Setup allows for ***execution with a single command***

→ ***Successful proof of usability & suitability***

workflow image: Kevin Boucher, lansa.com

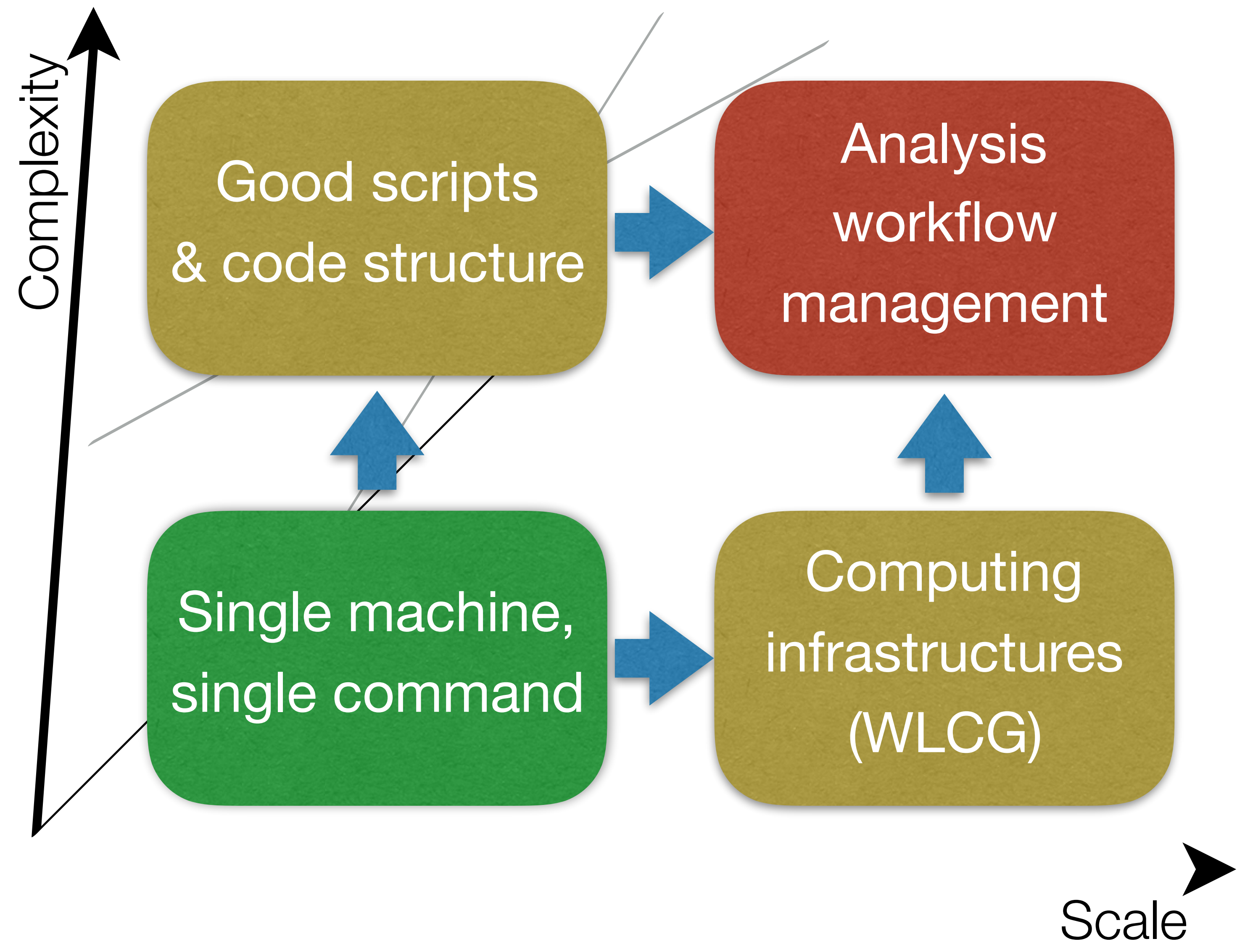
Summary

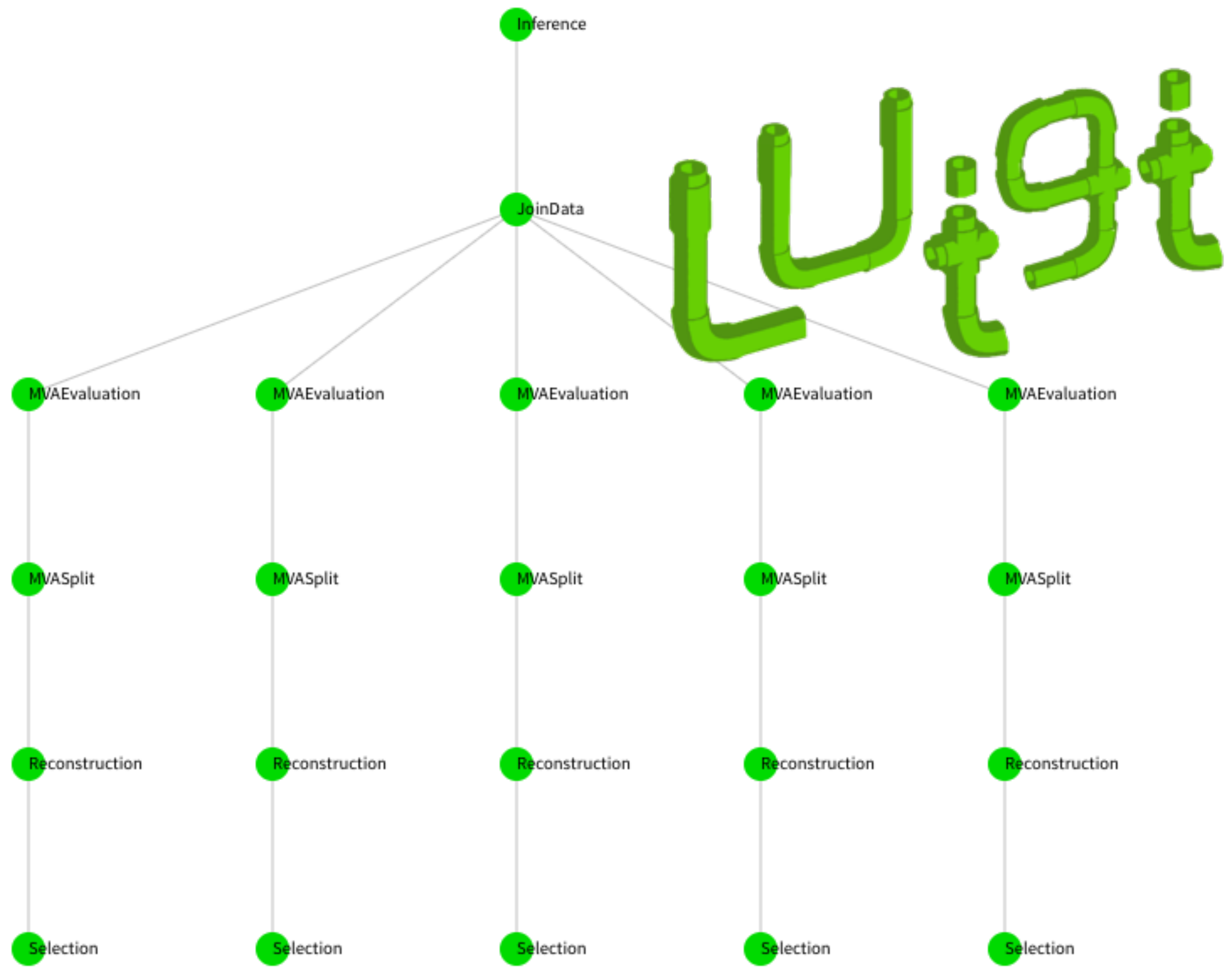
- HEP analyses likely to increase in scale and complexity
  → Analysis workflow management ***essential for success*** of future measurements
- Divergent requirements of existing, specialized management systems and those for “end-user” analyses
  → Need for a ***toolbox*** providing a ***design pattern***, ***not a framework***

- Luigi provides a promising way to model even ***complex*** workflows
- WLCG extension introduces ***scalability*** in the HEP context

- Increased ***transparency & reproducibility*** → ***analysis preservation***
- Encourages collaboration beyond code sharing

- Successfully applied in actual *ttH* analysis with CMS





Backup

Luigi - An Introduction

- Package for building complex pipelines
- Development started at Spotify, now open-source and community driven
- Simple core API:
  - Workloads are written in `Task`'s → `class Reconstruction(luigi.Task):`
  - Tasks are configured with `Parameter`'s → `dataset = luigi.Parameter(default="ttH125")`
  - Tasks can `require` other tasks, defines (multiple) dependencies → `def requires(self):`
  - Tasks produce `Target`'s, output representations with an `exist()` method → `def output(self):`
  - Actual workload defined in `run()` method, completely flexible via python code → `def run(self):`

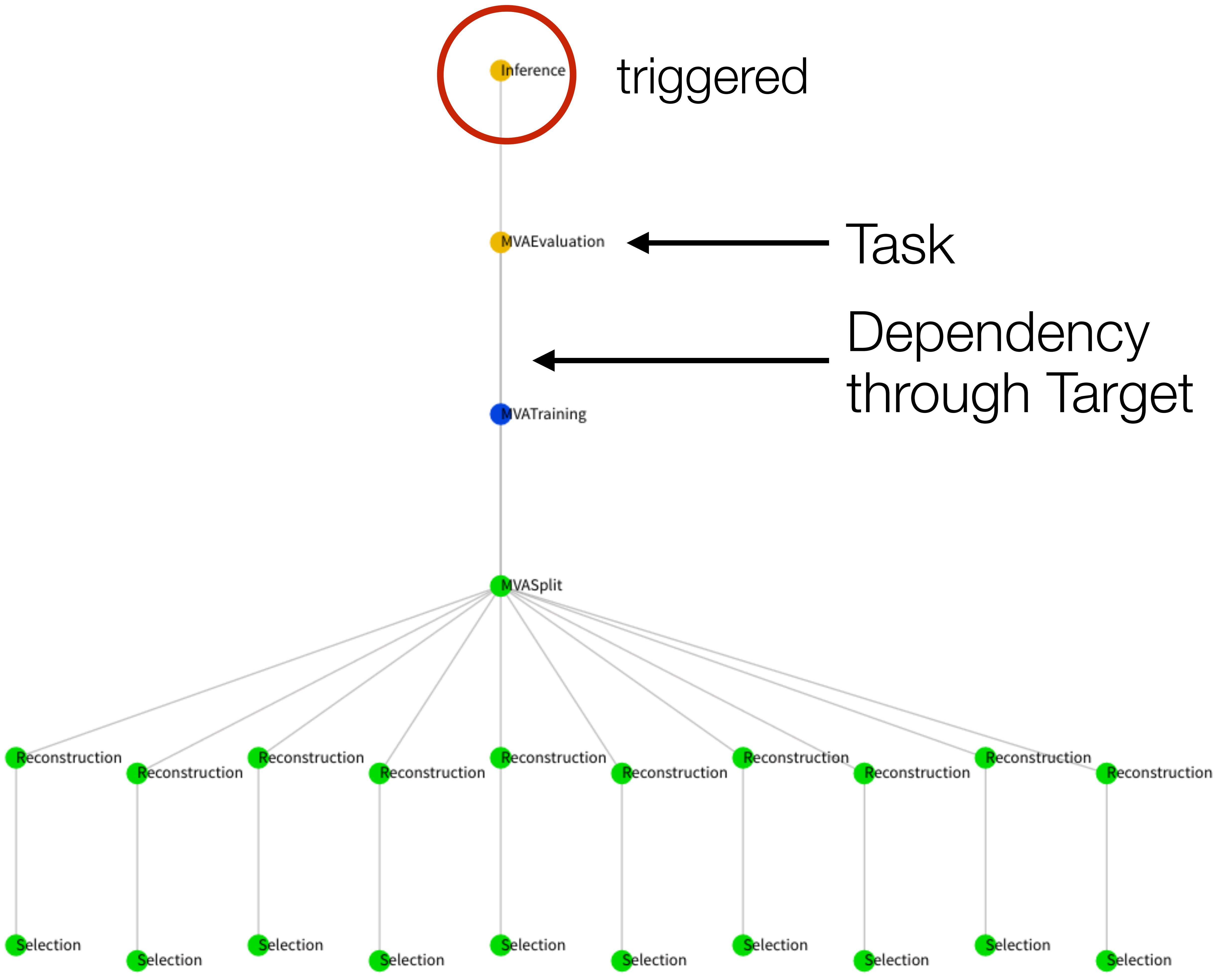
```python
# reco.py

import luigi

from analyses.ttH.tasks import Selection

class Reconstruction(luigi.Task):

    dataset = luigi.Parameter(default="ttH125")

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

    def output(self):
        return luigi.LocalTarget("reco_%s.root" \
                                 % self.dataset)

    def run(self):
        # do whatever a reconstruction does
        ...
```

```
> python reco.py Reconstruction --dataset ttH125
```


III. Physikalisches Institut
RWTH AACHEN UNIVERSITY

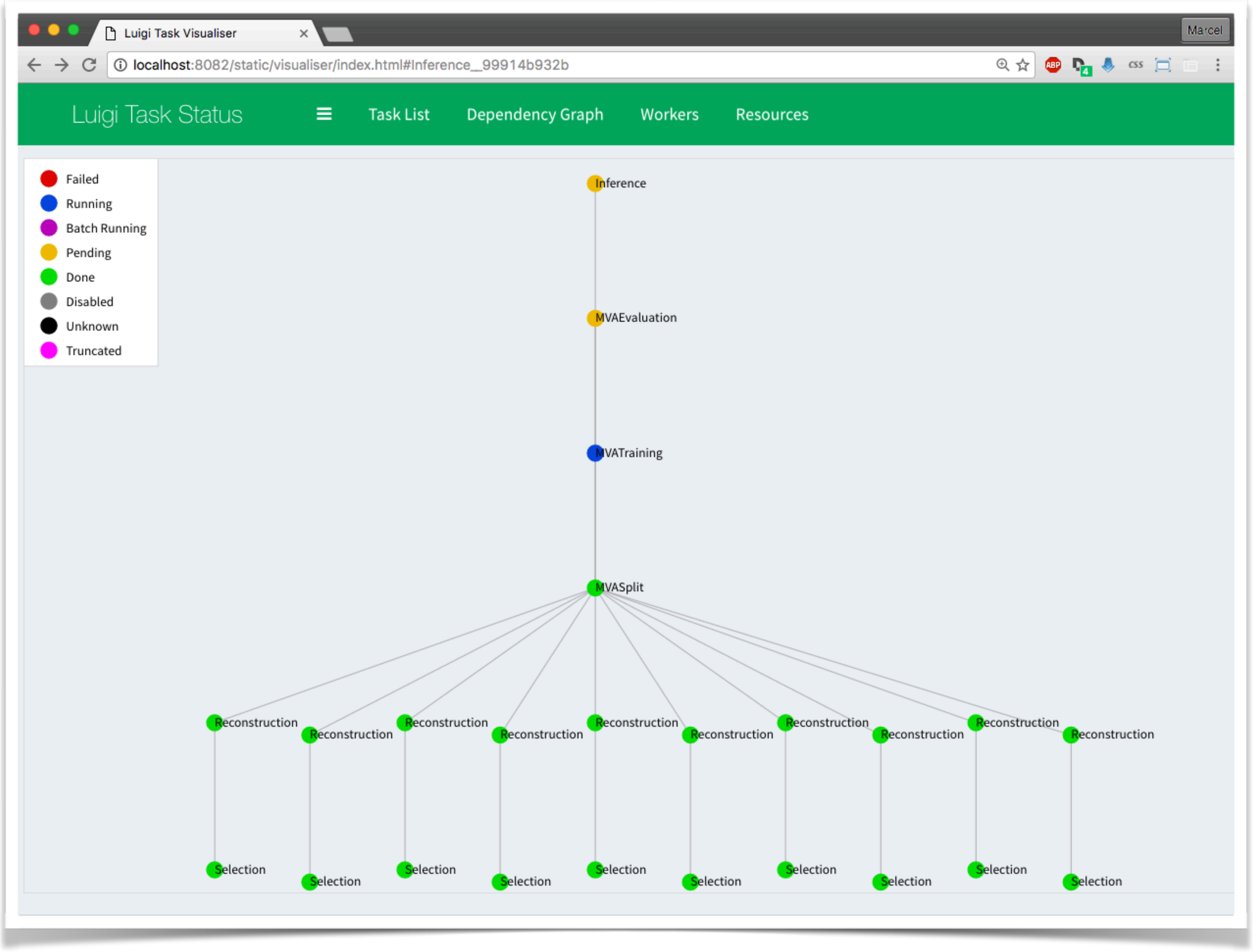
Luigi - make-like Execution

- Luigi's execution system is `make`-like, it only processes what is really necessary
  1. For the triggered task, create the dependency tree
  2. Determine tasks to actually run:
     2.1. Walk down the tree
     2.2. For each path, stop when all output targets of a task *exist*
  3. Run tasks in *n workers*
- Very clear & scalable through simple structure
- Error handling & automatic re-scheduling
- Command line integration & tools
- Central scheduling & visualization



Luigi - Central Scheduler

- Not a “scheduler” in HEP language, scheduling takes place on worker
- Think of it as a “global task lock”
- Optional, but powerful when working in teams / collaborations
  - Same task should not run twice
  - Saves resources but also ensures target/ data integrity
- Dependency, status & resource visualization
- Control of running workers (add, abort, …)
- Custom status messages & task history



III. Physikalisches Institut

RWTH AACHEN UNIVERSITY

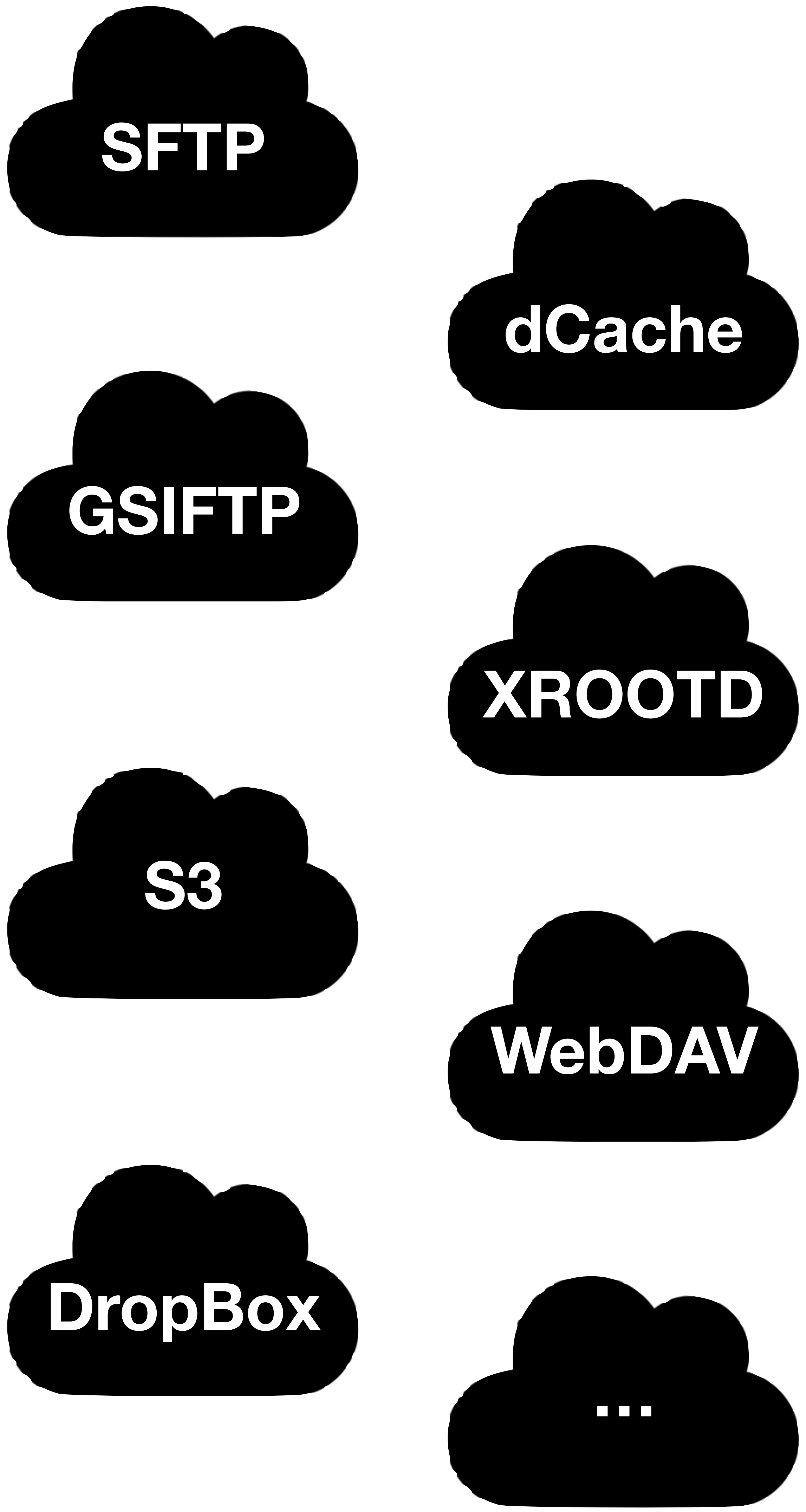
HEP Layer - GFAL Targets

- When running on the WLCG, use of storage elements is a necessity
- Fortunately, there is GFAL (Grid File Access Library)
  - Developed by Data Management Clients group at CERN
  - Command line tools & **python bindings**
  - Handles all file transfer protocols of the HEP community

→ Combine GFAL with Luigi targets

- Simple API, batch transfers, validation, auto-retry, local caching, …
- Usage equivalent to local targets

```
def output(self):
    return DCacheTarget("/path/to/file.txt")

def run(self):
    self.output().parent.touch()

    with self.output().open("w") as f:
        f.write("measurement results: ...")
```



III. Physikalisches Institut
RWTH AACHEN UNIVERSITY

Application: Implementation of Systematics

	"ShiftTask"			"AnalysisTask"
Systematics	Selection	Reconstruction	Evaluation	Inference
nominal	✔ (implements)	✔ (implements)	✔ (implements)	✔ (requires)
JES	☆	✔ (implements)	✔ (bubbles up)	✔ (requires)
JER	☆	✔ (implements)	✔ (bubbles up)	✔ (requires)
PDF	☆	☆	✔ (implements)	✔ (requires)
Q^2	☆	☆	✔ (implements)	✔ (requires)

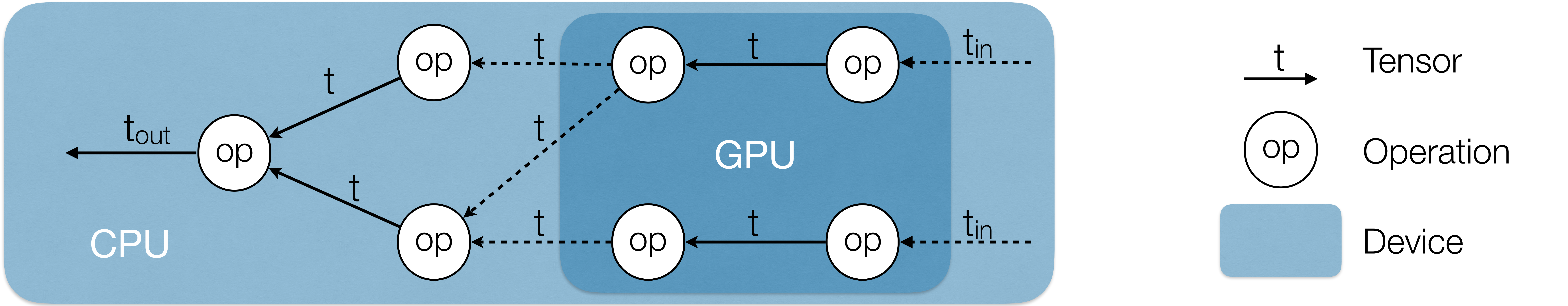
direction of processing →

- ✔ implements
- ✔ bubbles up / effective: nominal
- ✔ requires
- ☆ saved

→ "implement as late as possible"

tfdeploy (1)

- tensorflow *graphs* consist of *operations* and *tensors*



- Examples: $t_3 = add(t_1, t_2)$, $t_2 = softmax(t_1)$
- Ops are bound to devices (CPU/GPU), tensors are transferred if needed
- tfdeploy:
  - Implements tree structure in pure python
  - Tensors = numpy arrays
  - Ops = vectorized numpy calls, need to implement **all** tensorflow ops
  - Works in all environments, even in C++ with Python C-API, helpful for sharing

tfdeploy (2)

riga / tfdeploy

Unwatch 10 | Star 161 | Fork 13

Code | Issues 1 | Pull requests 0 | Projects 0 | Wiki | Pulse | Graphs | Settings

Deploy tensorflow graphs for fast evaluation and export to tensorflow-less environments running numpy. — Edit

README.md

tfdeploy

build passing | docs latest | pypi package 0.2.4

Deploy tensorflow graphs for *fast* evaluation and export to *tensorflow-less* environments running numpy.

Evaluation usage

```
import tfdeploy as td
import numpy as np

model = td.Model("/path/to/model.pkl")
inp, outp = model.get("input", "output")

batch = np.random.rand(10000, 784)
result = outp.eval({inp: batch})
```

Modular Analysis with VISPA & PXL