RECAST

Lukas Heinrich

DIANA Meeting 2016/05/02

NEW YORK UNIVERSITY

RECAST

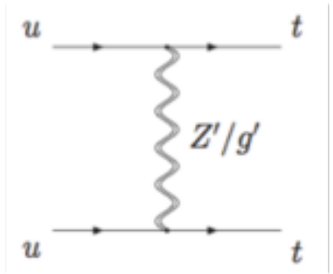
Experimental HEP faces idiosyncratic challenges re: archiving, reproducibility, re-executability of research output

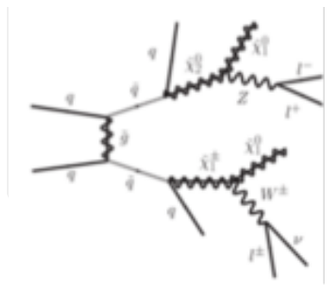
- few, large experiments / collaborations
- unique, but largely proprietary datasets (there'll ever only be one dataset of 8 TeV pp collisions)
- many interested third parties (read: phenomenologists): interested in re-execution, but lack access to data, knowledge to execute original data analysis

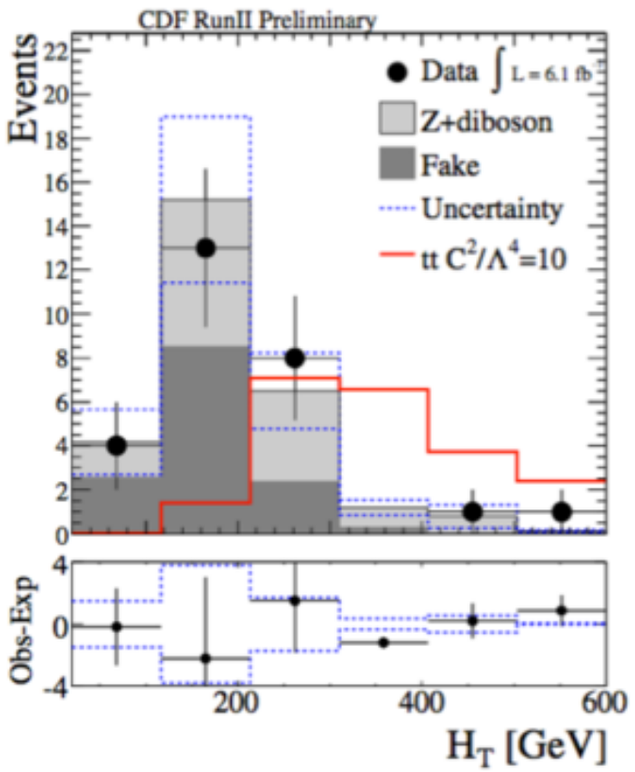
RECAST

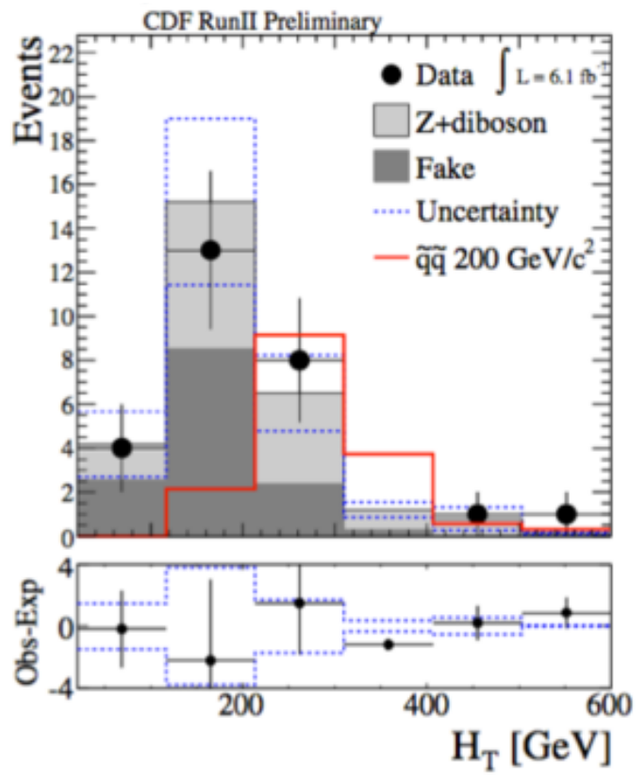
RECAST a service to address common requests from pheno: re-execute analysis w.r.t new signal signal model

- most resources go into simulating large background samples, taking data, stay fixed in this scenario
- just re-execute analysis chain for one new signal dataset









RECAST

Challenges in archiving analysis chain:

- wide range of software used, broad spectrum in quality, documentation, support, etc
- generally can't assume common interfaces, coordination

ROOT

numpy

Gaudi/Athena

MC Generators

collaboration-wide analysis software

case-specific frameworks (SUSY, Higgs)

HistFitter

one-off scripts

event selection

plotting

fitting code

(semi-)professionally developed / released / maintained

packages used by multiple analyses, midterm support

unique analysis code, know-how moves with analysis team

RECAST

Two ingredients to preservation:

1. capture individual analysis steps, including all required software dependencies, simple executable interface

2. capture workflow to execute analysis steps in correct order

Developed generic vocabulary to describe both. Goals:

- independent of execution backend
- clean separation of concerns, modularized
- flexible, extensible to support future changes
- robust, easily digestible data format

Capturing Activities

Capturing Activities

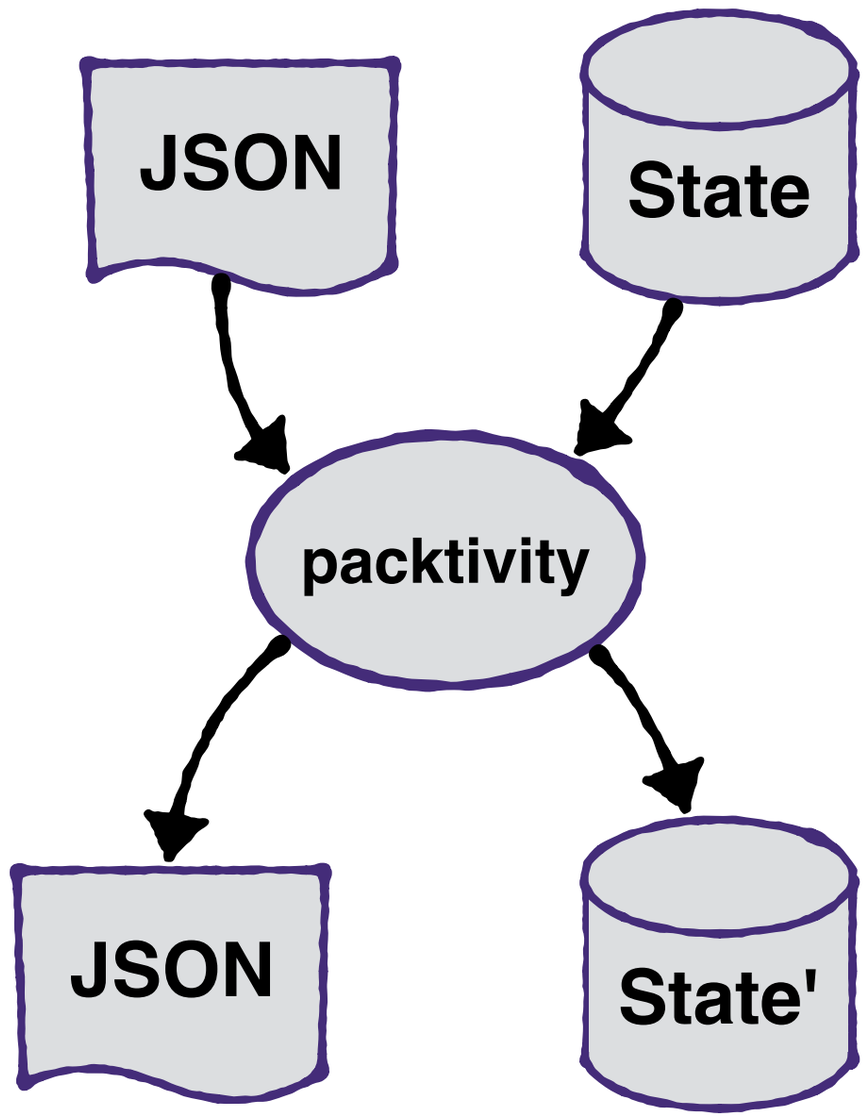
Packaged analysis step (*packtivity*) describes specific task, possibly parametrized by small number of arguments.

Inputs:

- parameters as JSON data
- external state (e.g. work directory)

Output:

- modified state
- semantic JSON description of output / activity results



Capturing Activities

Three ~independent ingredients

process:

takes input JSON and produces complete description.
Simple example: command line string template with interpolation fields, that are filled by passed JSON data.
Reconstruction Tags

environment:

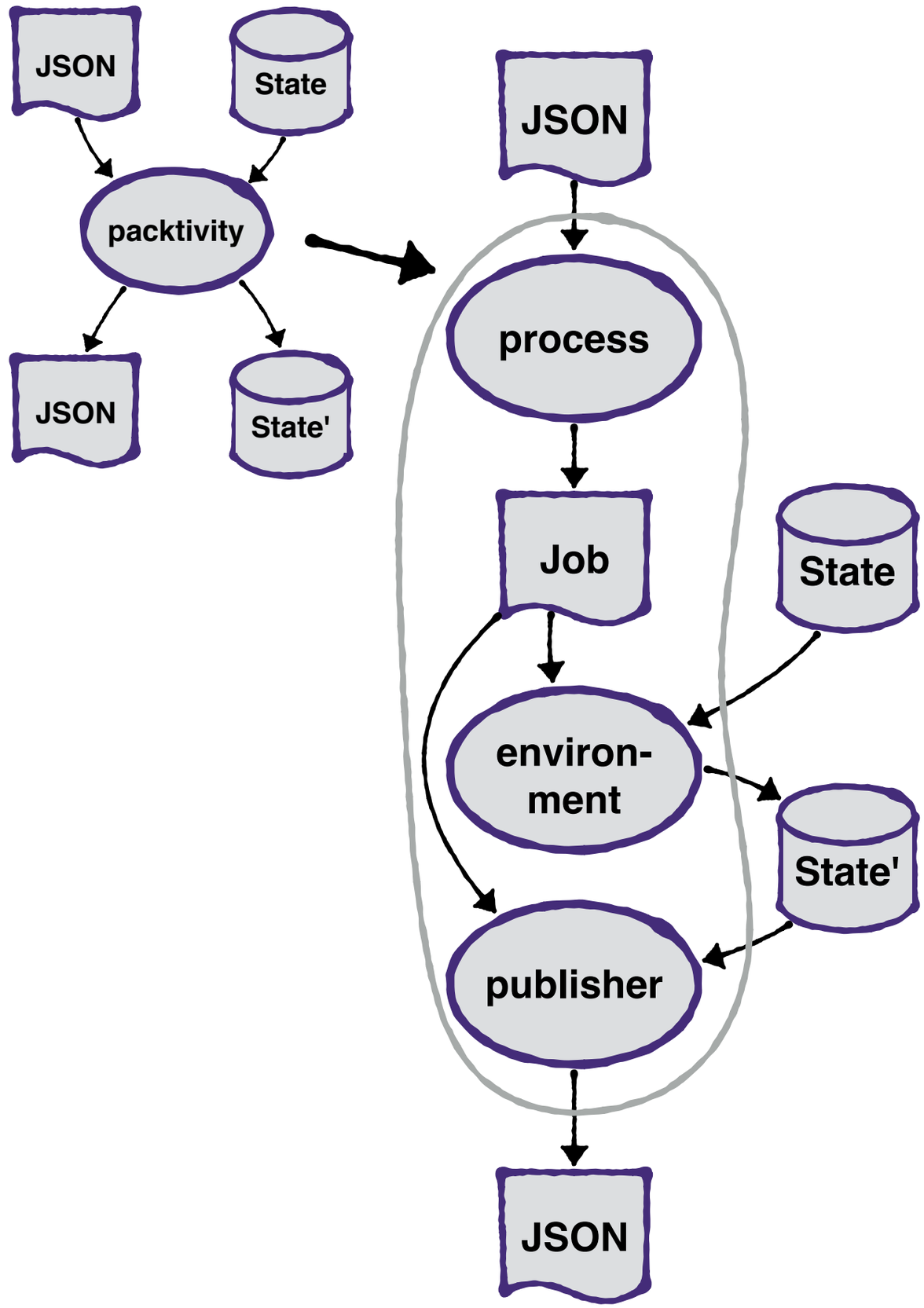
packaged execution environment, with all dependencies to run above job and write to external state.

Examples:

- Docker Image w/ attached external volume
- Full VM image w/ shared FS (EOS, AFS etc)
- Environment described by Umbrella (already uses json, but mixes process and environment notions)
- GRID Input tarball + cmtconfig

publisher:

human/machine readable JSON describing output. Often independent of job details: globbing files in workdir, declare certain input pars as output ("./cmd infile **outfile**")



Capturing Activities

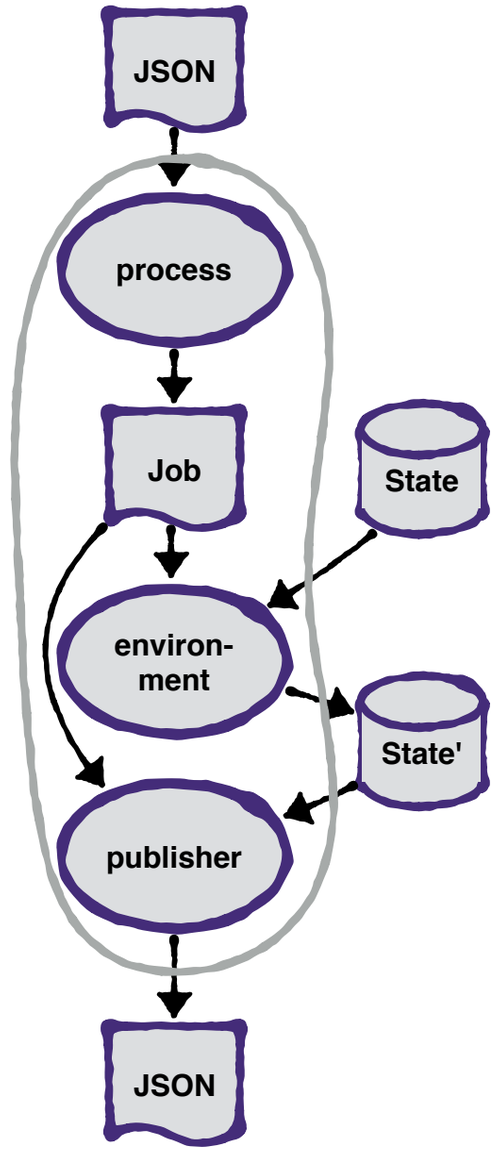
Example: fitting code of a Run-1 ATLAS SUSY analysis

```
process:
  process_type: 'string-interpolated-cmd'
  cmd: '/resources/pMSSM-TwoLep-Fit/post_fit.sh {fitresultsarchive} {workdir} {modelName} {resultsyaml}'
publisher:
  publisher_type: 'frompar-pub'
  outputmap:
    output: resultsyaml
environment:
  environment_type: 'docker-encapsulated'
  envscript: /resources/fitenv.sh
  image: lukasheinrich/dilepton_fit
  resources:
    - CVMFS
```

publisher that maps input arguments to output JSON

often, jobs can execute in same image but need different shells

for this env, we can request add. resource grid proxy, cvmfs, afs access



input JSON:

```
{
  "fitresultsarchive": "/workdir/fit.tgz",
  "workdir": "/workdir/fitwork",
  "modelName": 255123,
  "resultsyaml": "/workdir/fitres.yml"
}
```

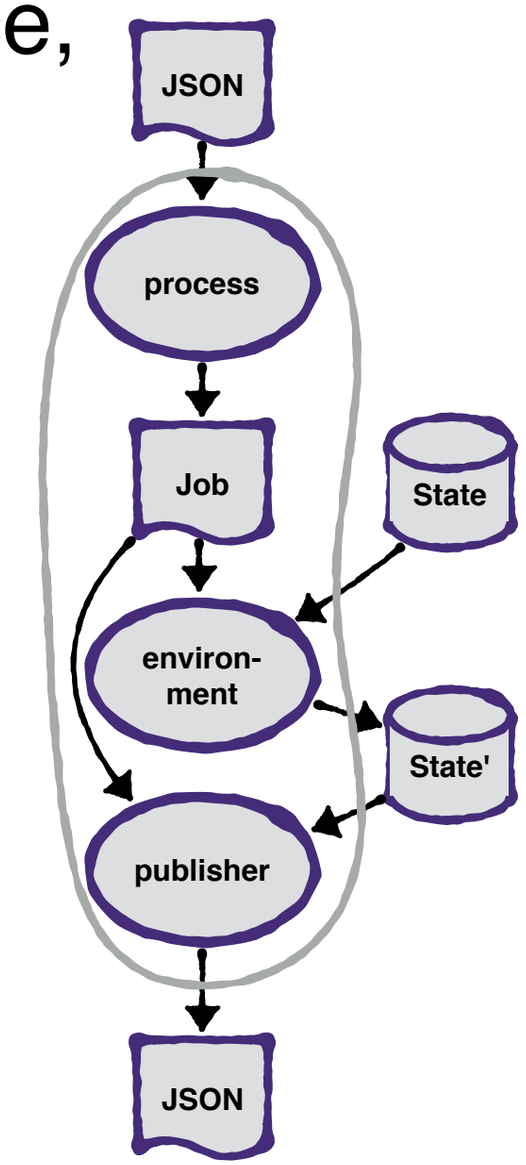
output JSON:

```
{
  "output": "/workdir/fitres.yml"
}
```

Capturing Activities

Packaging activities, according to process, env, publisher schema gives us:

- Simple JSON API to call parametrized analysis activities
    - probably can't rely on more common interface, JSON ubiquitous / future-proof
- Let's us add new component descriptions (new kinds of container techniques, publishers) down the road, w/o changing existing
- JSON for input and output, notions of external state sets us up nicely for using packtivities as building blocks for workflows



Capturing Workflows

Capturing Workflows

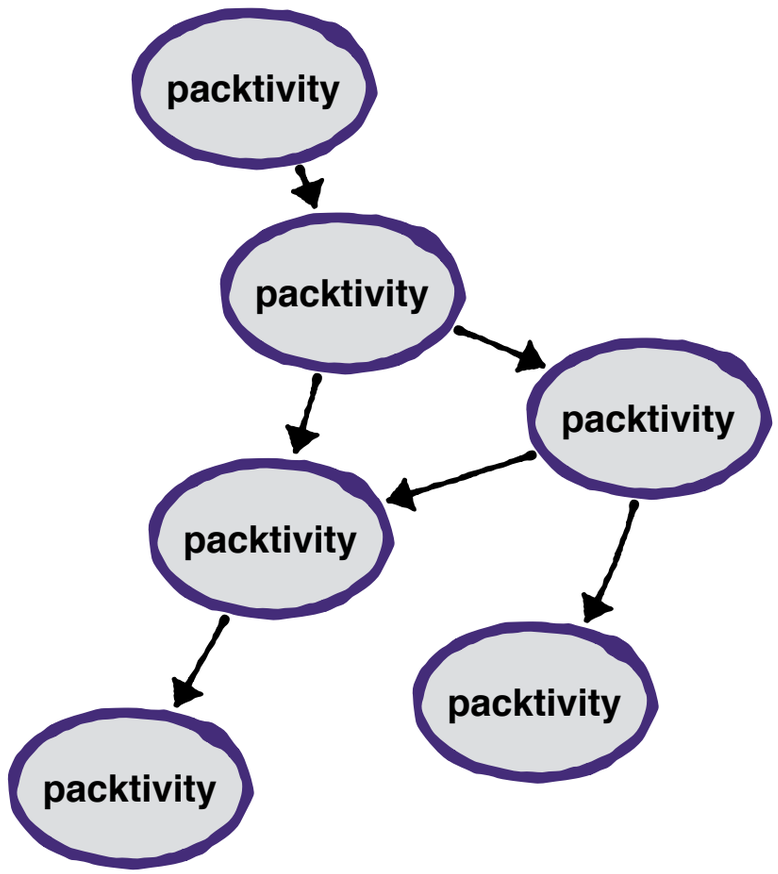
How do we use captured activities to execute more complex analyses? Two (three) options:

1. write meta-program that steers execution using your favorite language
   - pro: workflow can have arbitrary complexity
   - con: no introspection intro structure of workflow, handle parallelism manually, hard to re-use shared workflows, more analysis-specific code to maintain
2. Use declarative workflow description (language/schema)
   - con: can only describe supported workflow types
     - basic assumption: often similar workflow patterns are re-used ("map-reduce", simple chain, simple combinatorics)
   - pro: gives as static, analyzable/queryable description of workflow structure, re-use shared workflows easily, jobs can be distributed, parallelized from inspection
3. (Give up, hope for documentation, run everything by hand)
   - well, let's not do that…

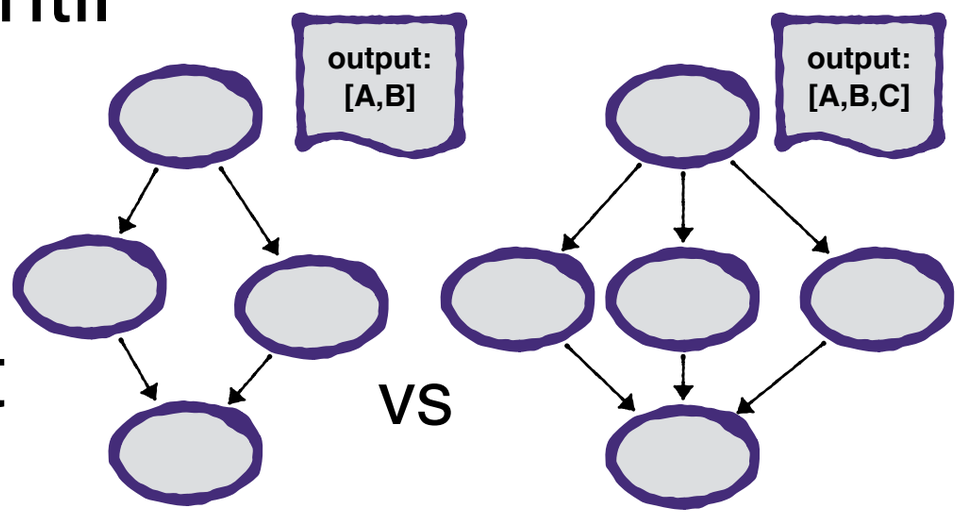
Capturing Workflows

Workflow Model:

- Directed Acyclic Graph:
  nodes=packtivities, edges=dependencies
- Fits W3C PROV standard terminology
- "structured job queue", jobs submitted to a backend when upstream ready



- Basic Problem:
  - entire DAG is often not known at until execution time. depends on activity results
  - simple example: download dataset process all files in parallel, merge.



Capturing Workflows

Dynamic Workflows

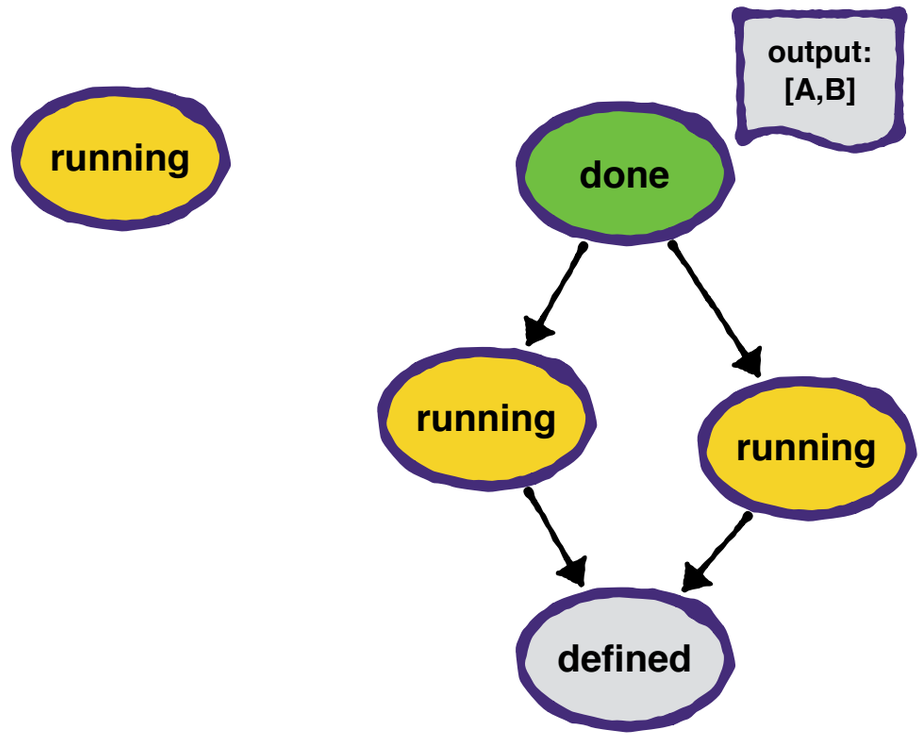
- described by a DAG + list of extension rules. Two components

  predicate: signal if extension can be applied based on current DAG state.

  Example: wait for dataset download to be finished

  body: extend DAG: add nodes, edges, *or new extension rules*

  Example: inspect download node output, create appropriate # of nodes, add edges



t = t0 #rules=1

t = t1 #rules=0

- Simple processing loop:
  - process until all nodes done no extensions left / applicable

```
#starting the loop
while nodes_left_or_rule(state):
    update_dag(state)
    process_dag(backend,state)

    #we're done for this tick, let others proceed
    yield state
```

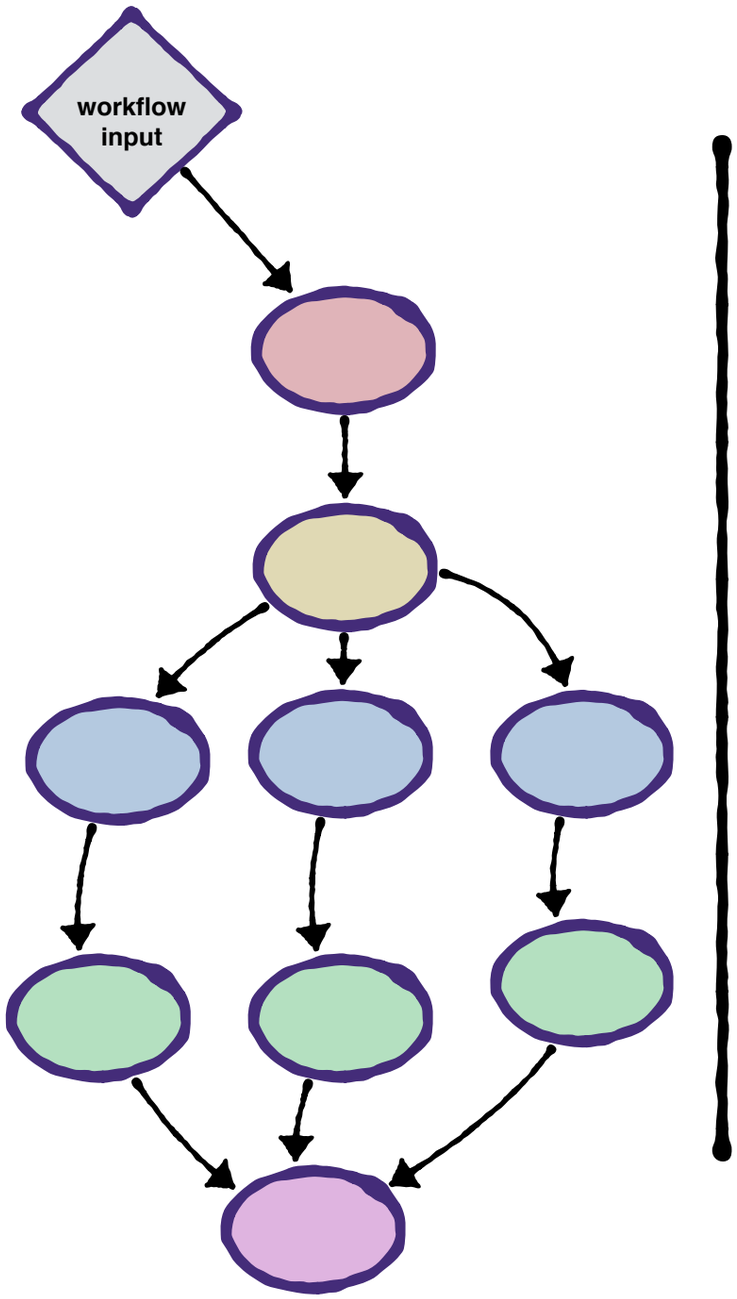
Capturing Workflows

Next step: describe extension rules in declarative form according to a JSON Schema.

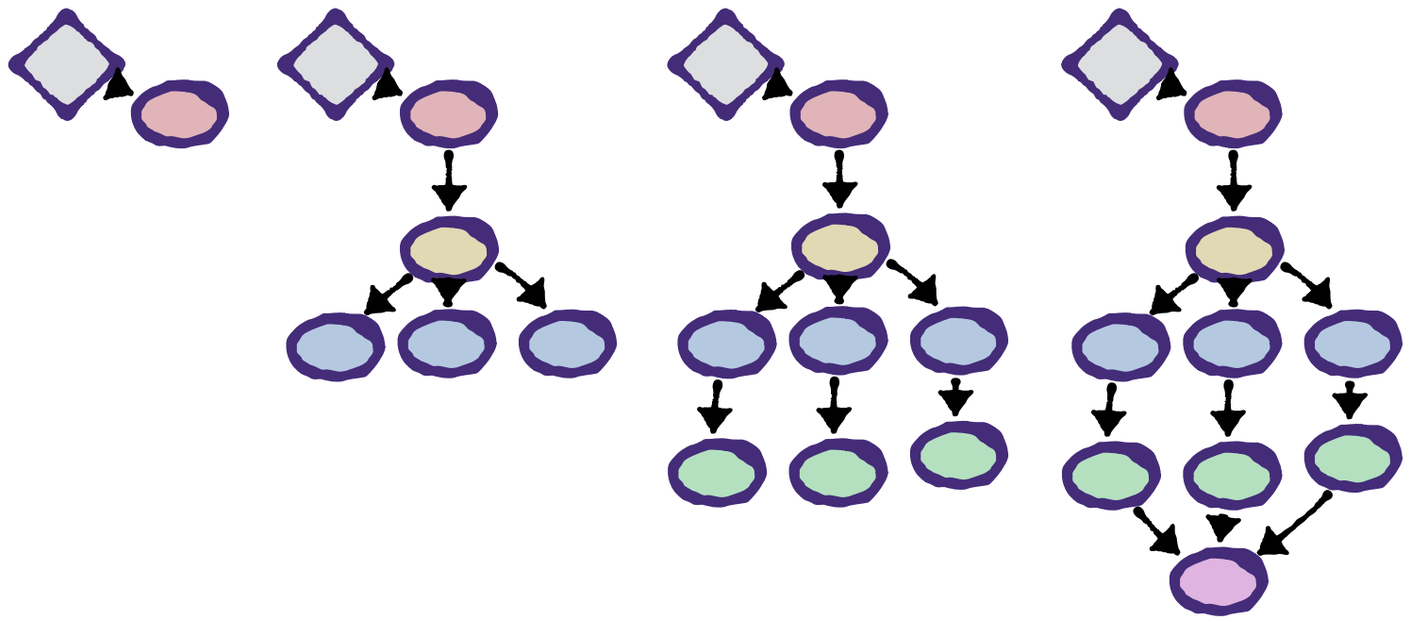
- We call each extension a workflow "stage"
  - simple predicates: a stage can be scheduled while all schedules activities of dependent stages are done
  - simple bodies: library of workflow patterns that take result JSON from dependencies' activities and creates new input JSON for newly scheduled activities, adds edges as necessary
    - examples: **map** (process array output w/ one activity each), **reduce** (collect array outputs with single activity), **combinatorics** (cartesian products, lockstep iteration/zip of arrays) etc.
    - operate solely on JSON data of activities, completely factorized from jobs. Can build quite complex workflows w/ very few basic building blocks (HEP workflows tend to be lots's of parallel processing / map-reduce). Suspect a saturation of patterns w/ time.

Capturing Workflows

Example: simple Monte Carlo Event generation pipeline



1. Input: Couplings
2. Generate Generator Cards from couplings
3. Create Integration Grid (one-time cost)
4. Generate hard interactions based on Grid in parallel
5. Run Parton Shower
6. Merge files



Capturing Workflows

Example: simple Monte Carlo Event generation pipeline

JSON reference to packtivity definition

```yaml
- name: prepare
  dependencies: ['init']
  scheduler:
    scheduler_type: singlestep-stage
    step: {$ref: 'steps.yml#/prepare'}
    parameters:
      par1: {stages: init, output: par1}
      par2: {stages: init, output: par2}
      param_card: '{workdir}/param.dat'

- name: grid
  dependencies: ['prepare']
  scheduler:
    scheduler_type: singlestep-stage
    step: {$ref: 'steps.yml#/grid'}
    parameters:
      param_card: {stages: prepare, output: param_card}
      gridpack: '{workdir}/grid.tar.gz'

- name: madevent
  dependencies: ['grid']
  scheduler:
    scheduler_type: multistep-stage
    step: {$ref: 'steps.yml#/madgraph'}
    parameters:
      gridpack: {stages: grid, output: gridpack}
      nevents: {stages: init, output: nevents}
      seed: {stages: init, output: seeds, flatten: true}
      lhefile: '{workdir}/lhefile_{index}.lhe'
    scatter:
      method: zip
      parameters: ['seed']
```

predicate: wait for prepare to finish before scheduling grids

reference upstream outputs to assemble new input JSON

multiple packtivities per stage

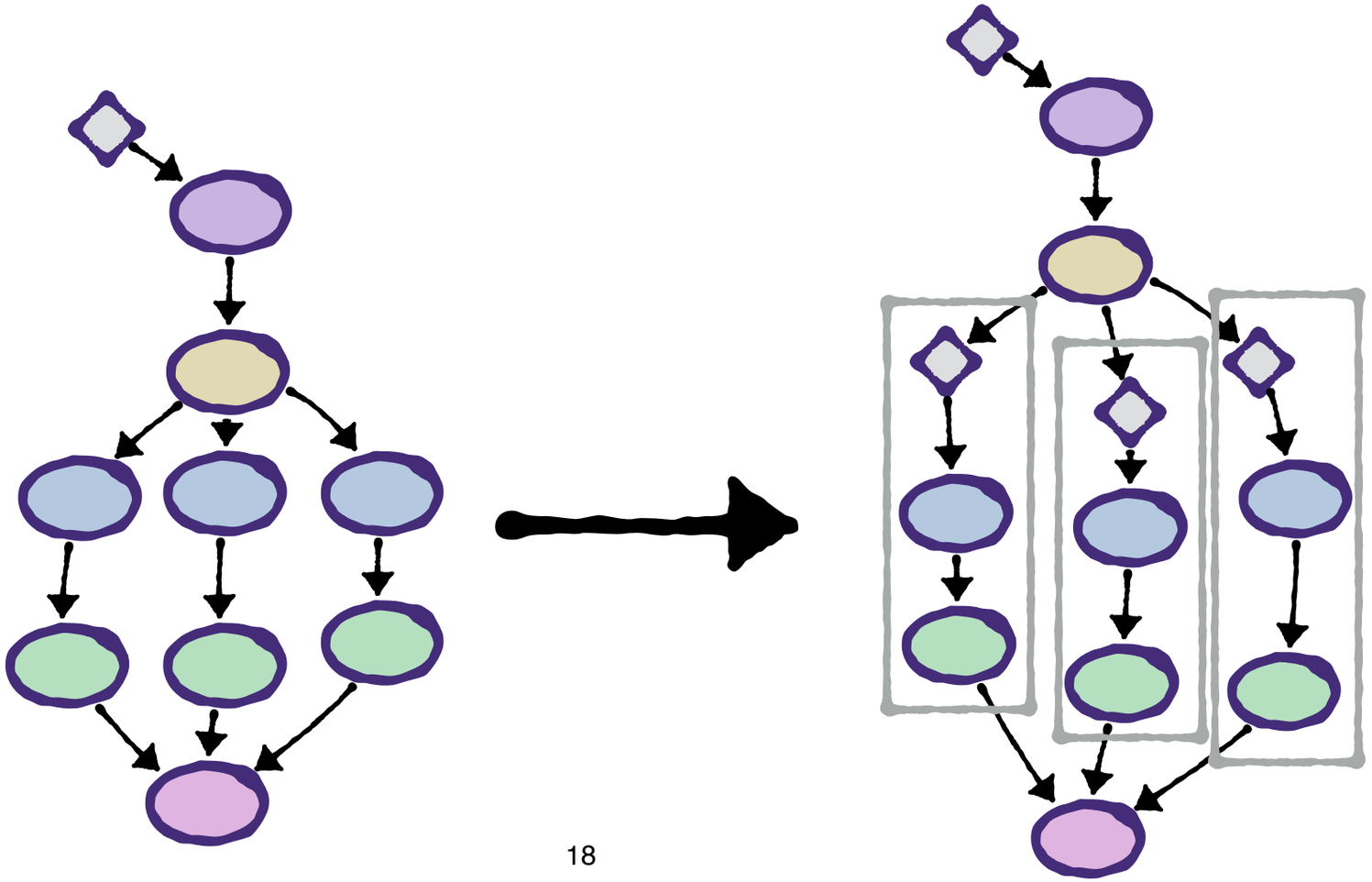
workflow.yml

```yaml
prepare:
  process:
    process_type: 'string-interpolated-cmd'
    cmd: './paramfromyaml.py --madgraph HC_UFO -i "{{kHzz: {par1}, kAzz: {p
  publisher:
    publisher_type: 'frompar-pub'
    outputmap:
      param_card: param_card
  environment:
    environment_type: 'docker-encapsulated'
    image: 'lukasheinrich/higgs-mc-studies'

grid:
  process:
    process_type: 'string-interpolated-cmd'
    cmd: './scripts/setupgrid.sh {param_card} {gridpack}'
  publisher:
    publisher_type: 'frompar-pub'
    outputmap:
      gridpack: gridpack
  environment:
    environment_type: 'docker-encapsulated'
    image: 'lukasheinrich/higgs-mc-studies'

madgraph:
  process:
    process_type: 'string-interpolated-cmd'
    cmd: './scripts/rungrid.sh {gridpack} {nevents} {seed} {lhefile}'
  publisher:
    publisher_type: 'frompar-pub'
    outputmap:
      lhefile: lhefile
  environment:
    environment_type: 'docker-encapsulated'
    image: 'lukasheinrich/higgs-mc-studies'
```

steps.yml

Capturing Workflows

- Many HEP data pipelines share a lot of upstream data processing (reconstrution from RAW data, basic event selection / thinning, ATLAS: derivations). Only diverge at the very end.
- Want to be able to compose workflows from subworkflows.
  - results in recursively nested workflows, "scoped" extensions



Capturing Workflows

Example: simple Monte Carlo Event generation pipeline

```yaml
stages:
  - name: prepare
    dependencies: ['init']
    scheduler:
      scheduler_type: singlestep-stage
      step: {$ref: 'steps.yml#/prepare'}
      parameters:
        par1: {stages: init, output: par1}
        par2: {stages: init, output: par2}
        param_card: '{workdir}/param.dat'

  - name: grid
    dependencies: ['prepare']
    scheduler:
      scheduler_type: singlestep-stage
      step: {$ref: 'steps.yml#/grid'}
      parameters:
        param_card: {stages: prepare, output: param_card}
        gridpack: '{workdir}/grid.tar.gz'

  - name: madevent
    dependencies: ['grid']
    scheduler:
      scheduler_type: multistep-stage
      workflow: {$ref: 'subchain.yml'}
      parameters:
        gridpack: {stages: grid, output: gridpack, unwrap: true}
        nevents: {stages: init, output: nevents, unwrap: true}
        seed: {stages: init, output: seeds, flatten: true}
      scatter:
        method: zip
        parameters: ['seed']

  - name: rootmerge
    dependencies: ['*.analysis']
    scheduler:
      scheduler_type: singlestep-stage
      step: {$ref: 'steps.yml#/rootmerge'}
      parameters:
        mergedfile: '{workdir}/anamerged.root'
        inputfiles: {stages: '*.analysis', output: analysis_output}
```

same scheduler, but just specify workflow instead of step. Input JSON become subworkflow input

depend on all "analysis" stages in subfworkflows. Uses JSONPath

rootflow.yml

```yaml
stages:
  - name: madevent
    dependencies: ['init']
    scheduler:
      scheduler_type: singlestep-stage
      step: {$ref: 'steps.yml#/madgraph'}
      parameters:
        gridpack: {stages: init, output: gridpack}
        nevents: {stages: init, output: nevents}
        seed: {stages: init, output: seed}
        lhefile: '{workdir}/lhefile.lhe'

  - name: pythia
    dependencies: ['madevent']
    scheduler:
      scheduler_type: singlestep-stage
      step: {$ref: 'steps.yml#/pythia'}
      parameters:
        settings_file: /analysis/mainPythiaMLM.cmnd
        hepmcfile: '{workdir}/outputfile.hepmc'
        lhefile: {stages: madevent, output: lhefile}

  - name: delphes
    dependencies: ['pythia']
    scheduler:
      scheduler_type: singlestep-stage
      step: {$ref: 'steps.yml#/delphes'}
      parameters:
        detector_card: /analysis/template_cards/modified_delphes_card_ATLAS.tcl
        outputfile: '{workdir}/outputfile.root'
        inputfile: {stages: pythia, output: hepmcfile}

  - name: analysis
    dependencies: ['delphes']
    scheduler:
      scheduler_type: singlestep-stage
      step: {$ref: 'steps.yml#/analysis'}
      parameters:
        fromdelphes: {stages: delphes, output: delphesoutput}
        analysis_output: '{workdir}/anaout.root'
```

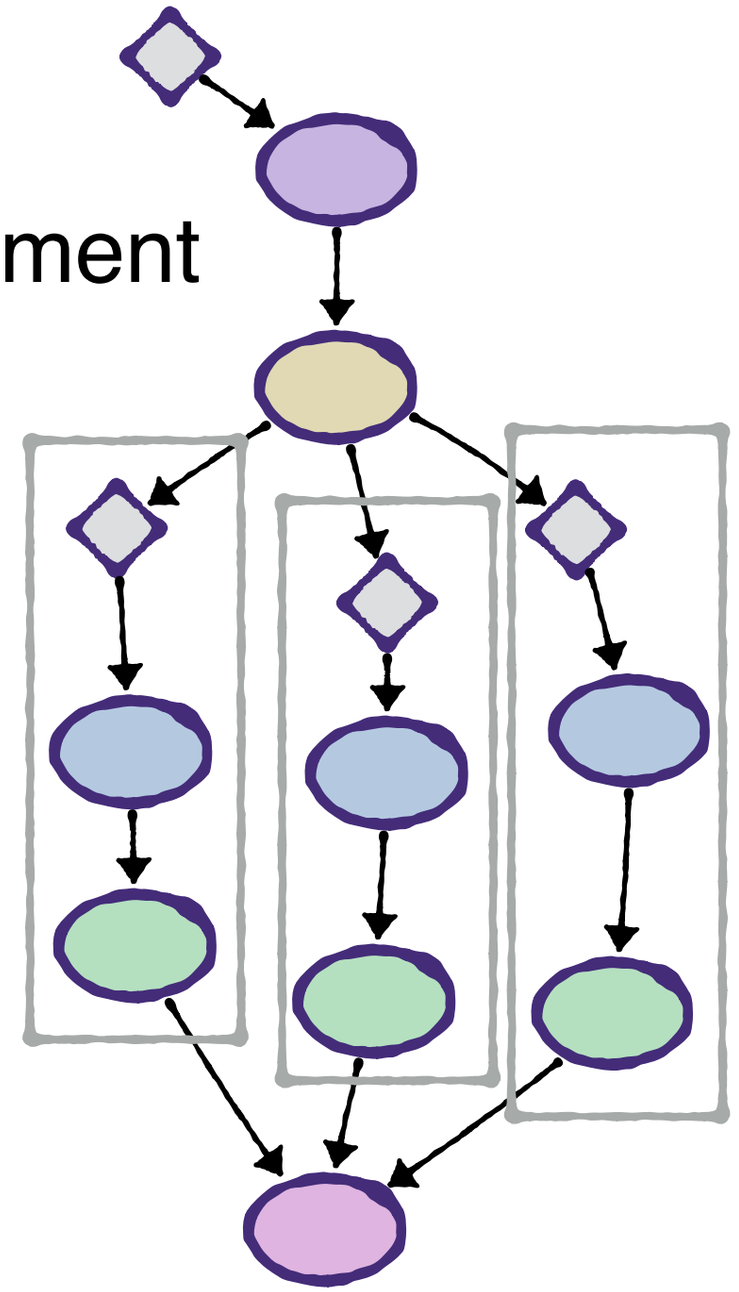
subworkflow does not need to respect any naming scoping resolves correct stages

subchain.yml

Capturing Workflows

Taking Stock:

- Defined two sets of JSON schemas
  - Wrapping Activities + execution environment into "packtivity" that has JSON API
  - Generic workflow definition that uses APIs + a number of patterns to assemble complex workflows

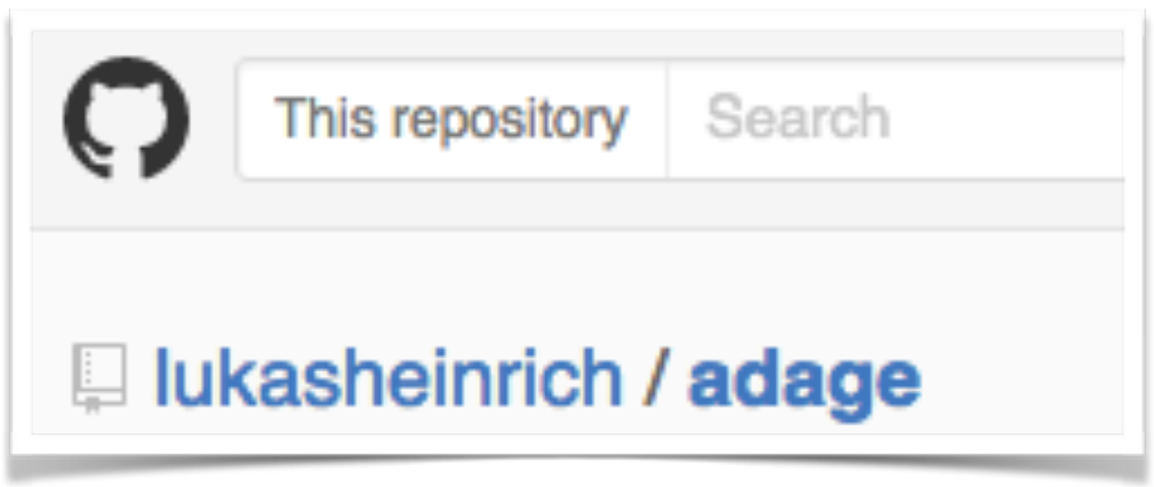


Executing Workflows

Executing Workflows

Reference / Demo implementation to execute workflows.
Three packages:

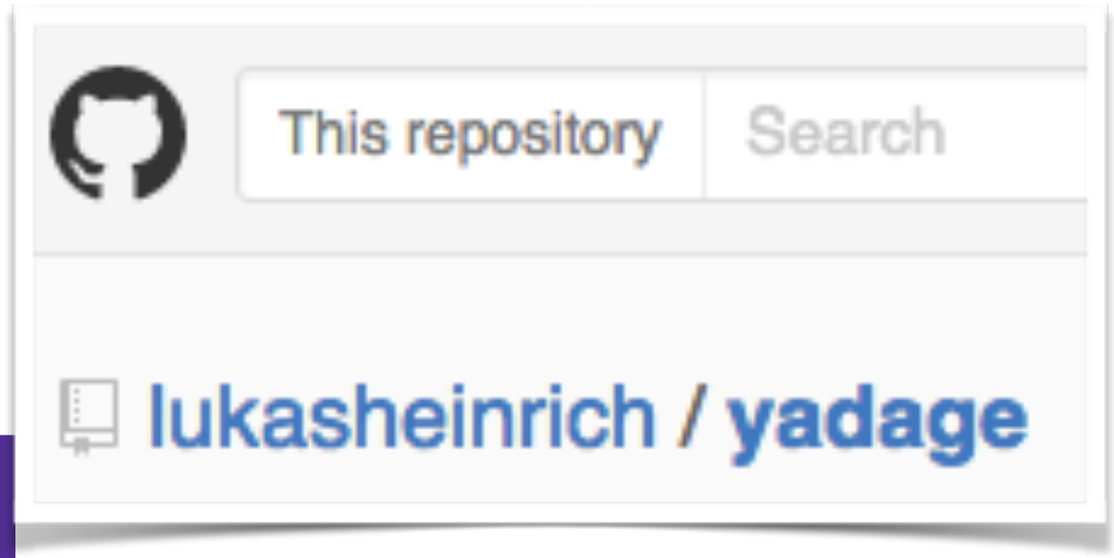
- **adage**: low-level steering of DAG update/process loop for tasks/extension rules defined as python callables.
  - Pluggable backend support. Currently: multiprocessing pool, distributed computing via Celery Worker pool. Obvious new backends: HTCondor cluster, etc…
- API access to event loop via single coroutine. Ticks in loop could also be steered via web service w/ workflow state stored in DB

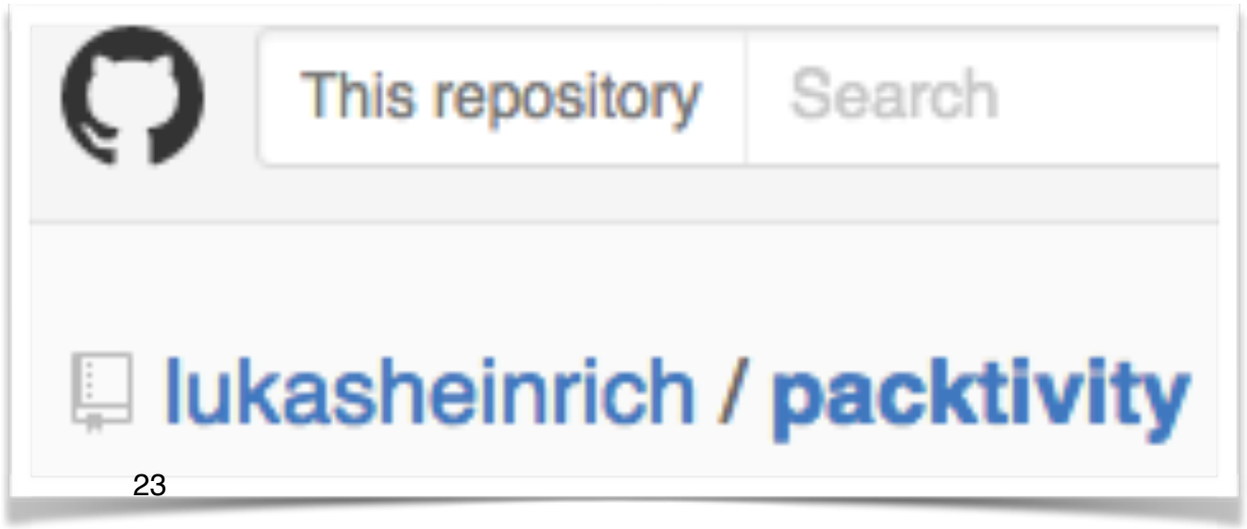


Executing Workflows

Reference / Demo implementation to execute workflows.
Three packages:

- **packtivity**: read-in packtivity definitions, make JSON API available as python callables. Has process/publisher/ environment handlers (e.g. craft docker command, attach external state etc)
- **yadage (yaml+ adage)**: parsing workflow definitions from JSON/YAML, library of standard DAG extensions, scoping

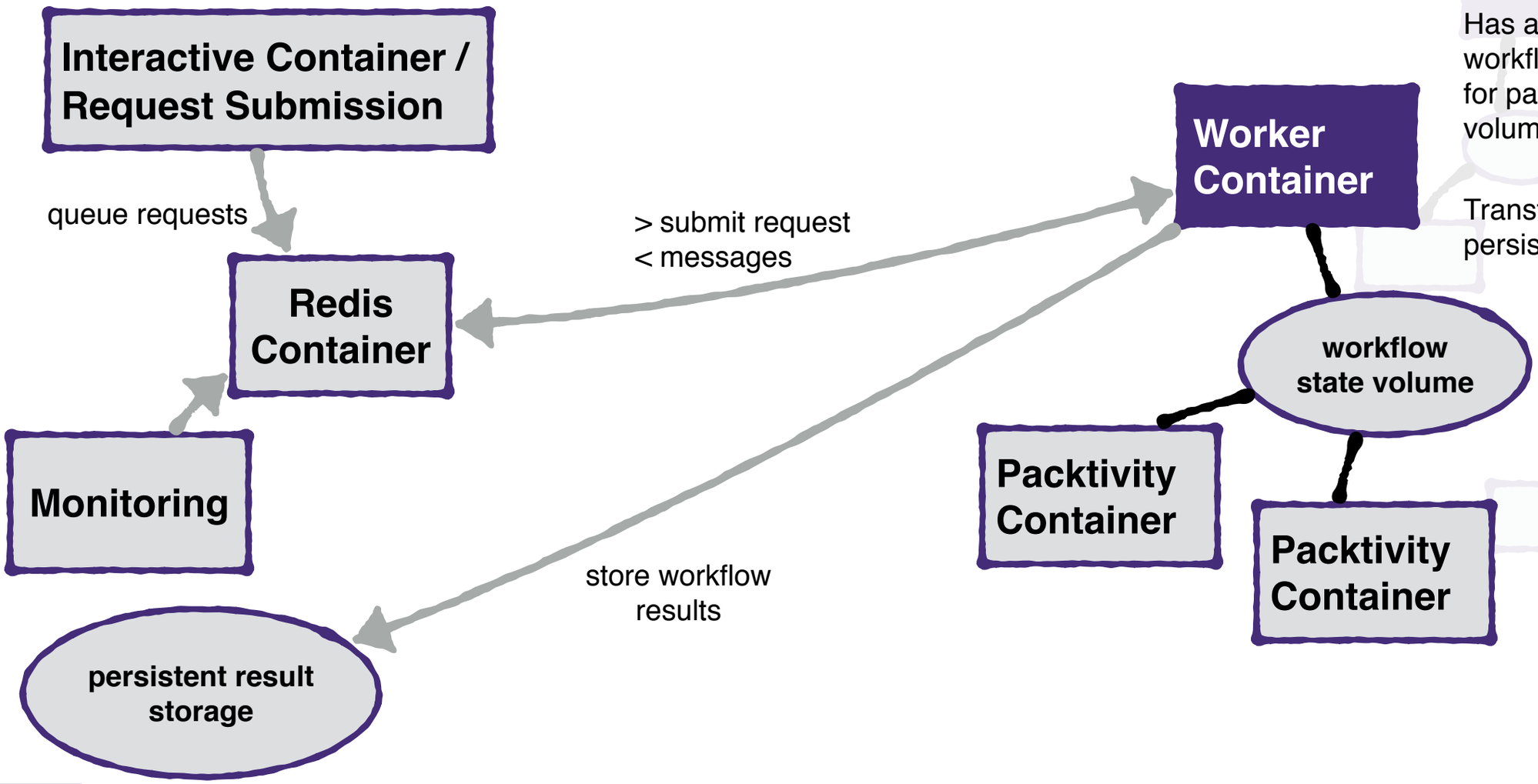




Executing Workflows

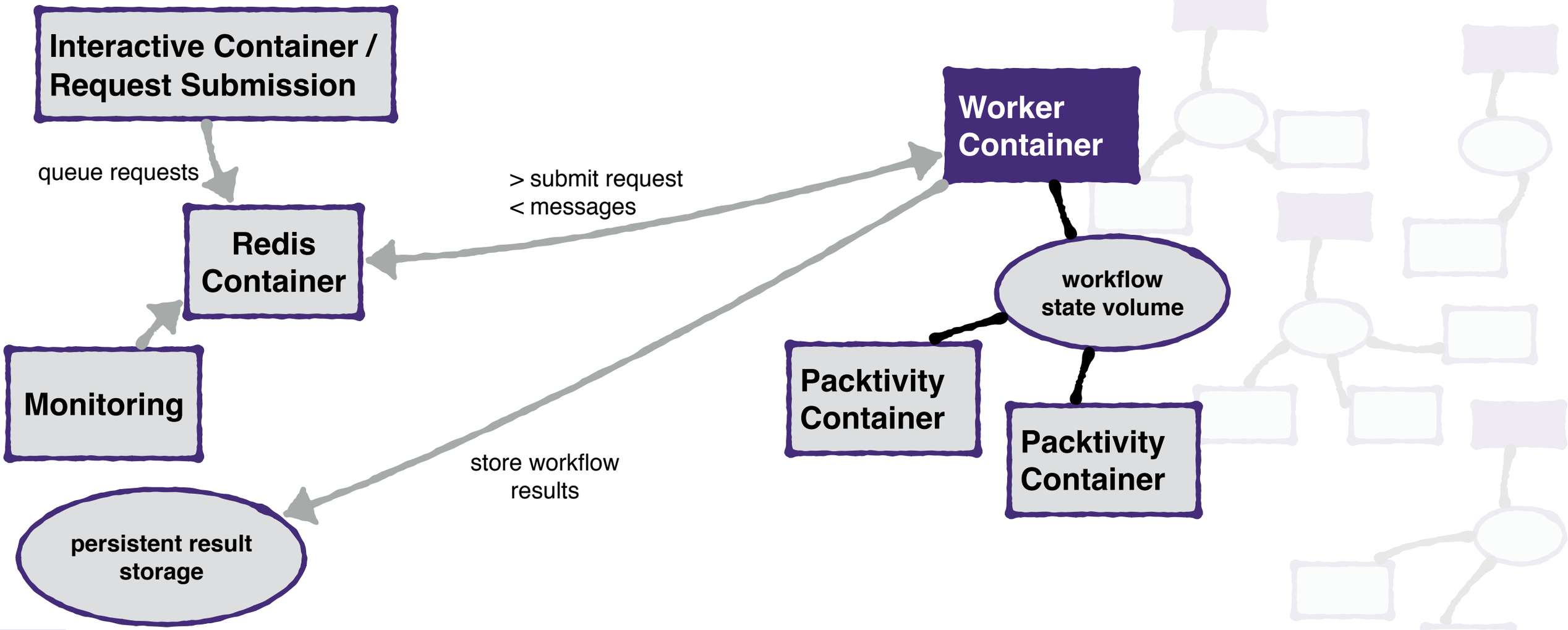
Executing RECAST workflows in a scalable deployment:

- Demo Cluster on CERN OpenStack: Cluster of ~30 VCPUS with Docker installed. Celery Job Queue to request Workflow processing



Executing Workflows

- 1-Click deployment of new nodes via user-scripts (plan to transition to OpenStack Heat).
- Worker nodes, glorified Docker Hosts (we add a bit of Monitoring, CVMFS, GRID Host Certs), could transition to CERN Container Project



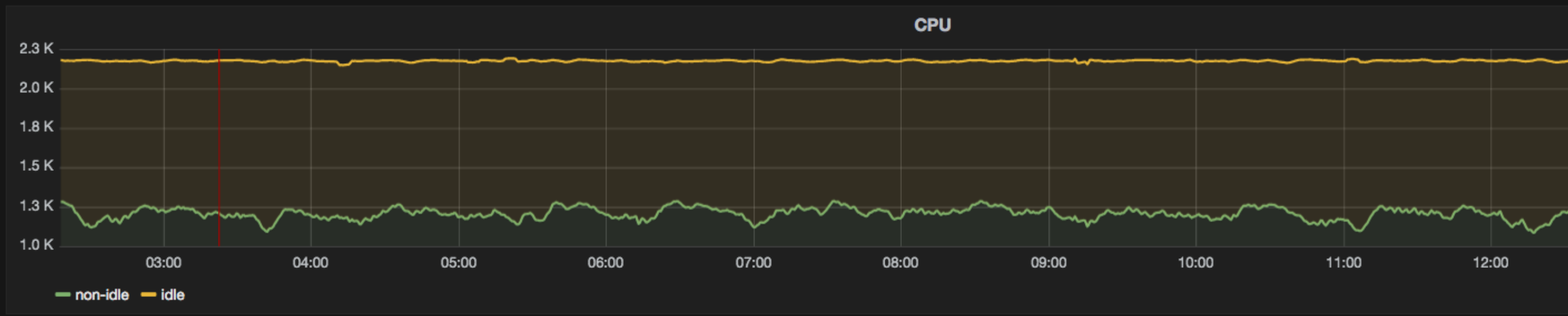
Cluster Overview

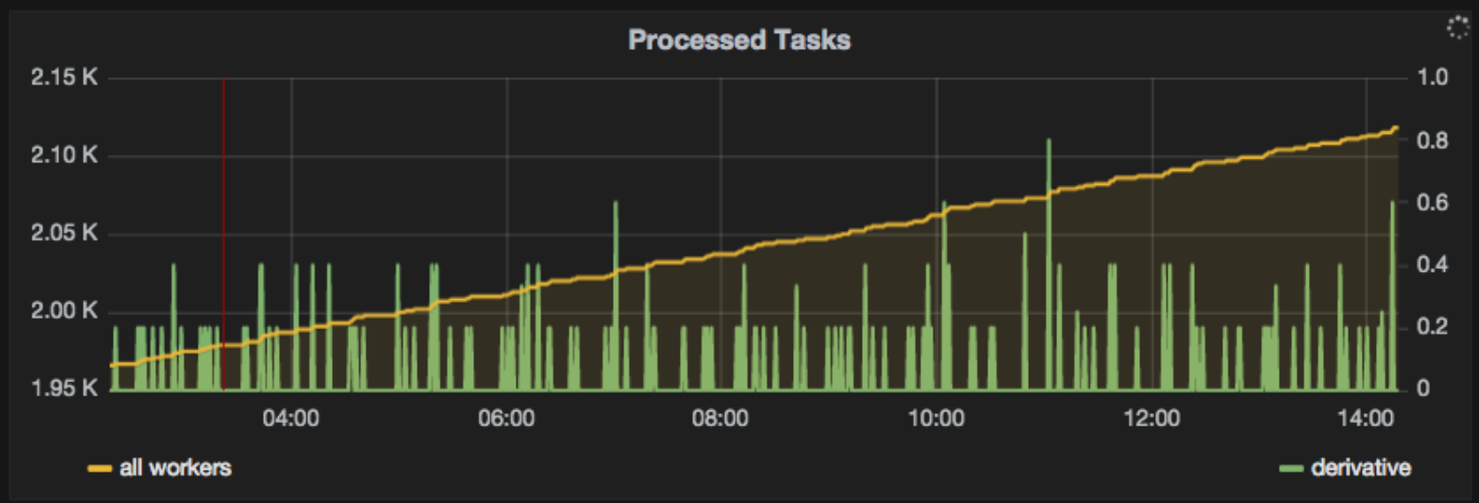
Zoom Out

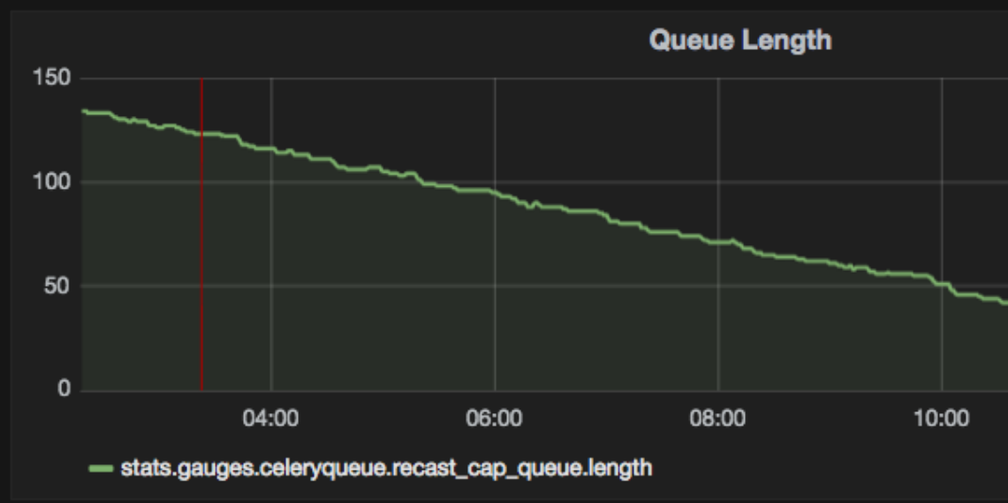
12 hours ago to a few

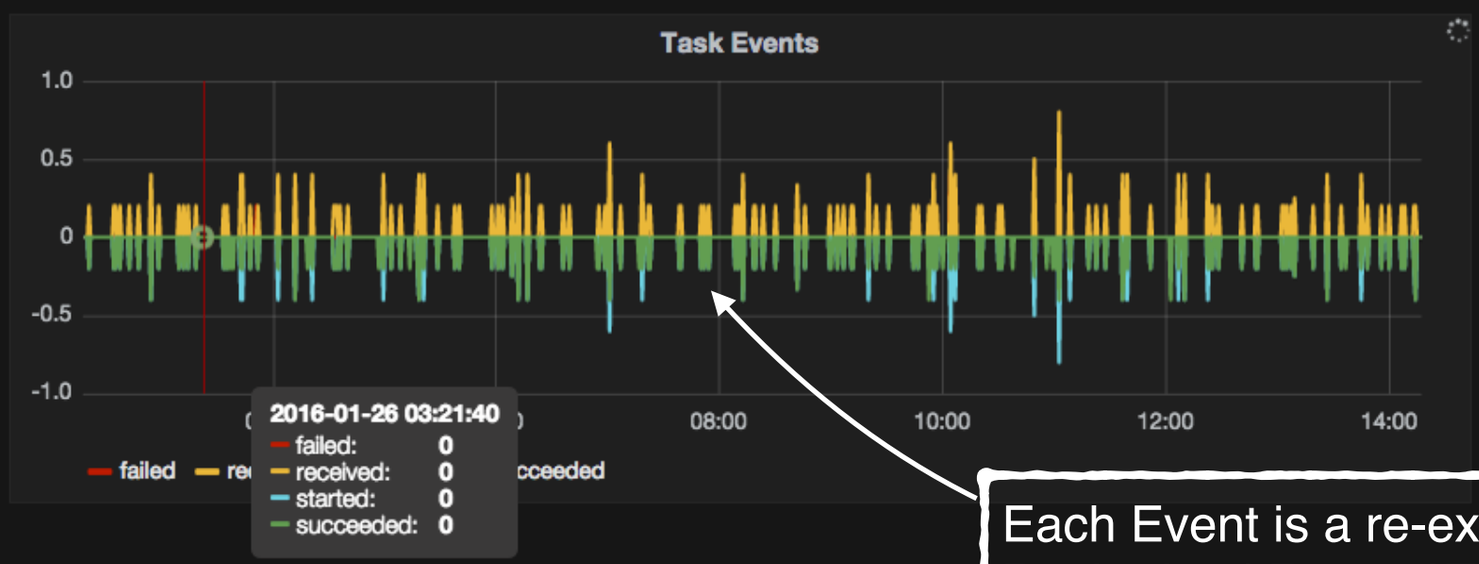
server: All

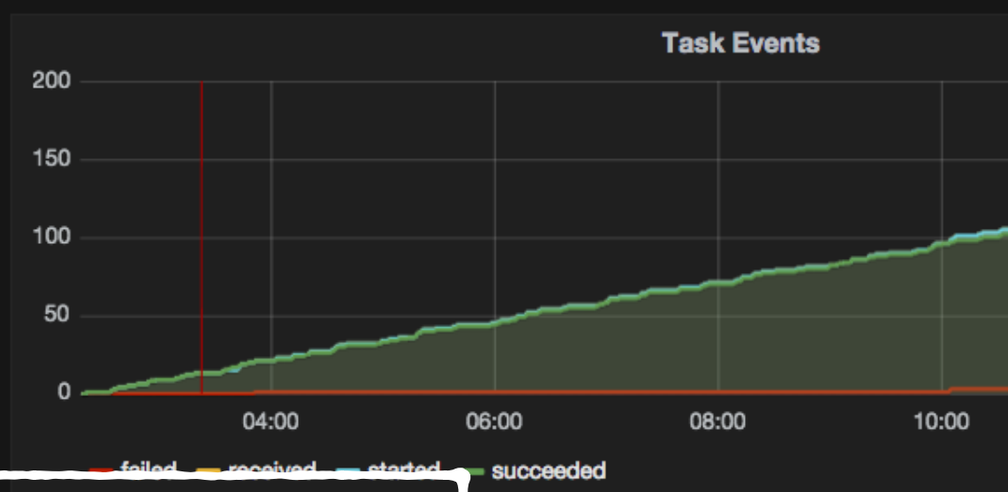
cpucomp: iowait + irq + nice + softirq + steal + system + user











Each Event is a re-execution of a Dockerized Run-1 ATLASAnalysis Chain

CERN Analysis Preservation

CERN Analysis Preservation Portal

- New effort to archive analysis information in Invenio-based backend
- Naturally acts on JSON records, ideal place to store Workflow Definition, exposes API to query for workflows of a given analysis
- Can now store workflows according to our schemas, RECAST can pull workflow & re-execute it with new input
- Will make it easy to cross-reference shared workflow stages across analyses