Journal of Physics: Conference Series **664** (2015) 062058 doi:10.1088/1742-6596/664/6/062058

Exploring Two Approaches for an End-to-End Scientific Analysis Workflow

Scott Dodelson[1,2,3], Steve Kent[1], Jim Kowalkowski[1], Marc Paterno[1] and Saba Sehrish[1]

[1] Fermi National Accelerator Laboratory, Batavia, IL 60510-0500, U.S.A.
[2] Kavli Institute for Cosmological Physics, University of Chicago, Chicago, IL 60637, U.S.A
[3] Department of Astronomy & Astrophysics, University of Chicago, Chicago, IL 60637, U.S.A

E-mail: ssehrish@fnal.gov

Abstract. The scientific discovery process can be advanced by the integration of independently-developed programs run on disparate computing facilities into coherent workflows usable by scientists who are not experts in computing. For such advancement, we need a system which scientists can use to formulate analysis workflows, to integrate new components to these workflows, and to execute different components on resources that are best suited to run those components. In addition, we need to monitor the status of the workflow as components get scheduled and executed, and to access the intermediate and final output for visual exploration and analysis. Finally, it is important for scientists to be able to share their workflows with collaborators. We have explored two approaches for such an analysis framework for the Large Synoptic Survey Telescope (LSST) Dark Energy Science Collaboration (DESC); the first one is based on the use and extension of Galaxy, a web-based portal for biomedical research, and the second one is based on a programming language, Python. In this paper, we present a brief description of the two approaches, describe the kinds of extensions to the Galaxy system we have found necessary in order to support the wide variety of scientific analysis in the cosmology community, and discuss how similar efforts might be of benefit to the HEP community.

1. Introduction

The scientific discovery process can be simplified and advanced by the integration of independently-developed programs run on diverse computing facilities into coherent workflows usable by scientists who are not experts in computing. Our goal in this work is to develop a framework that scientists can use so they can focus on science issues, rather than having to learn how to perform uninteresting computing tasks. Such a framework should:

- integrate independently developed programs;
- seamlessly run these programs on disparate computing facilities and monitor the progress of each program;
- facilitate both of the above in the form of easy to describe workflows,
- easily allow the addition of new components to, or modification of existing components in, the workflows;
- propagate relevant metadata and provide correct input and output connection;
- provide useful debugging information and error messages in case of failure; and

Content from this work may be used under the terms of the Creative Commons Attribution 3.0 licence. Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.
Published under licence by IOP Publishing Ltd

- allow publishing and sharing of results and workflows with collaborators.

Our exploration involves two approaches for the framework. One is based on the use and extension of Galaxy [4], a web-based portal for biomedical research, and the other is based on a programming language, Python. The two approaches are driven by two requirements of the framework; one is minimal intervention when workflows are running, and maximal intervention while workflows are developed.

In this paper, we present a brief description and comparison of the two approaches. Section 2 explains the LSST DESC use case in detail, sections 3 and 4 describes each approach, followed by comparison and conclusion in sections 5 and 6 respectively.

2. Cosmology use case

For our investigation, we used a cosmology problem selected from an LSST use-case catalog [6], "Correct galaxy number density for depth variation and track uncertainties". The immediate scientific goal is to transform the different LSST observations into a mask that encodes how deeply a given region was observed. The mask quantifies how many galaxies there should be in a given pixel. We broke down the problem into smaller steps as listed below.

(i) Workflow input
(ii) Image simulation
(iii) Galaxy catalog creation
(iv) Power spectrum calculation
(v) Cosmological parameter estimation
(vi) Plotting estimation results

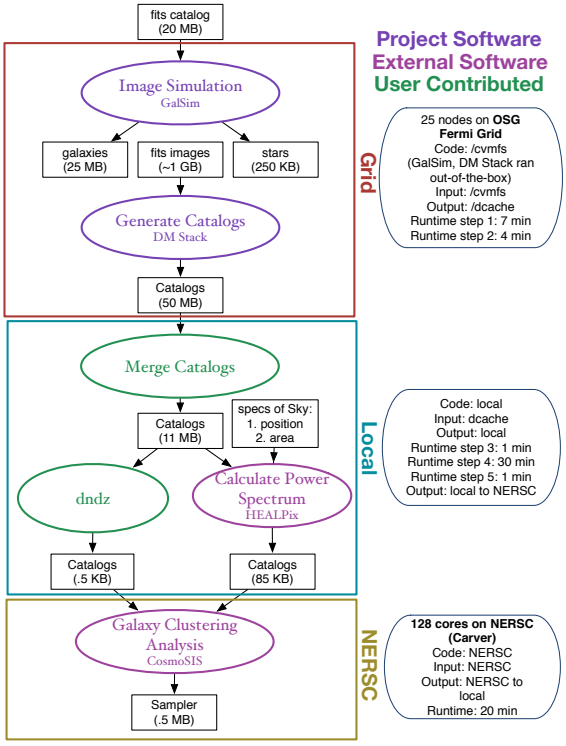

Figure 1. Detailed steps in our workflow

Detailed description of each of the steps is as follows.

Preparing Input: There is an initial step to prepare the input catalogs based on the input parameters provided by the user interface and divides the area of sky to be simulated into a set of "coadd tiles" - each approximately 0.22 degrees on a side. A "coadd tile" represents a portion of the sky that may have been observed multiple times by the LSST, which thus has greater sensitivity than a single observation. Parameters appropriate for each tile are computed and used as input to the next step. These parameters define the specific area on the sky to be covered and describe the performance of the telescope, camera, atmosphere, and night sky.

Image Simulation: In this step, one simulated image per coadd tile is created. The simulated images include both stars and galaxies as shown in the Figure 2. Star positions and brightness are generated randomly. The galaxy positions, sizes, brightnesses, and orientations are taken from an existing catalog of galaxies taken from the COSMOS survey [2]. The COSMOS survey images a small portion of the actual sky using multiple telescopes, including the Hubble Space

Telescope, and is well matched to the sensitivity that will be obtained by LSST. Images of individual objects are created using GalSIM [10]. Stars have a simple Gaussian profile with width determined by atmospheric conditions. Galaxies are given a "Sersic profile", which is specified by seven parameters total chosen to match the COSMOS galaxy measurements. Noise arising from Poisson photon statistics and instrumental effects is added.

A typical image might have 9000 galaxies and 1700 stars. Of these, approximately 8000 galaxies (and 500 stars) are detected and measured.

Generate Catalogs: The second step runs the LSST Data Management programs (DM stack [5]) on the simulated coadd tile images. The program detects, classifies, and measures parameters of the stars and galaxies. Because of the added noise and possible overlaps in images, the output catalog will only approximately match the input catalog.

Merge Catalogs: The third step combines the object catalogs from each of the coadd tiles, trimming them to eliminate duplicate detections where the tiles overlap, and matching up with the "truth" catalog in order to assign photometric redshifts (which are not computed otherwise in this simulation). The output is a single catalog of about 200,000 galaxies in a square degree that are ready for processing by the CosmoSIS code [11].

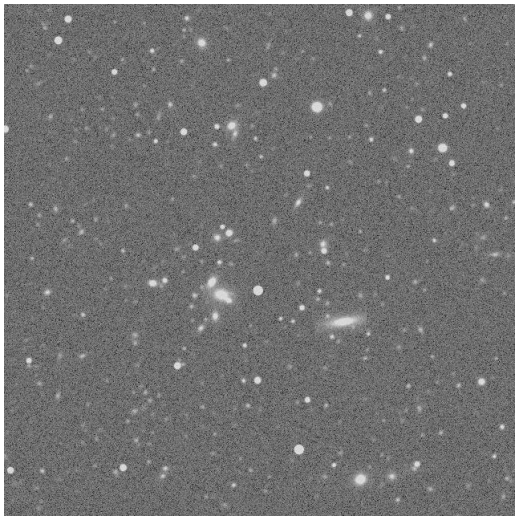

Figure 2. Input Image; it shows 0.1% of the total area of sky covered by the simulation. Fuzzy blobs are galaxies; round, compact things are perfect stars.

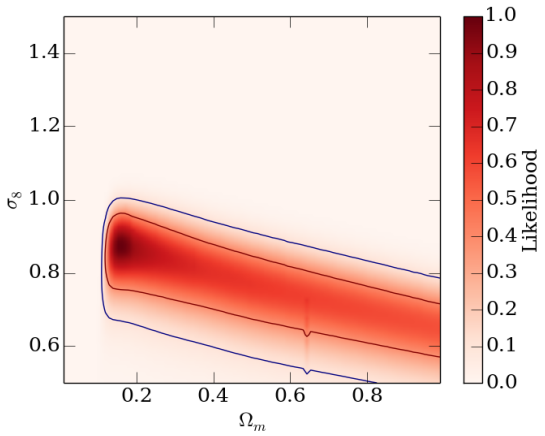

Figure 3. Output plot of correlated likelihood function.

dndz: In this step, we calculate the normalized distribution of the photometric redshifts of the galaxies found in step 3. This is a simple process of creating histogram, typically using a 40 equal-sized bins in z, from $z = 0$ to $z = 2$.

Calculate Power Spectrum: In this step, we calculate the (pseudo) power spectrum C_ℓ as a function of ℓ for the over-density of galaxies in a specified region of the sky. It operates on the combined catalogs of the galaxies found step 3. It uses a HEALPix [9] grid with NSIDE=4096 (for a resolution of about 0.86 arcmin), and HEALPix's ANAFAST routine to calculate the power spectrum. Finally is subtracts a simulation of the over-density from Poisson noise, with the same mean as the data, from the power spectrum of the data, and writes out the noise-subtracted power spectrum.

CosmoSIS: The final step is the cosmological parameter estimation using CosmoSIS [11]. CosmoSIS is a modular system for cosmological parameter estimation, based on Markov Chain

Monte Carlo and related techniques. In this instance, CosmoSiS is used to compare the power density with a theoretical prediction, based on values of the cosmological parameters σ_8 and Ω_m. CosmoSiS calculates the likelihood of the observed data as a function of σ_8 and Ω_m.

Plotting: The likelihood values produced by CosmoSIS can be plotted, using several Python scripts that are part of the CosmoSIS product. It produces three plots: a two-dimensional plot the likelihood as a function of Ω_m vs. σ_8 as shown in figure 3, and a plot of the marginalized 1-dimensional likelihood for each of Ω_m and σ_8.

Figure 1 also shows the resource requirement for each step. It shows three different blocks; each block encapsulates the steps that will be run on one type of computing resource. The first two steps are configured to be run on FermiGrid. All the software and user code resides in the CERN-VM File system (CVMFS) [1]. The FermiGrid nodes have access to the CVMFS, and all the input and output files are read/written to *dCache* [3]. Once processing is done, files are copied back from dCache to the local host, where next few steps are executed. The output from local machines is then moved to NERSC, where the final computation step. The size of the input/output and runtime at each stage is also shown in figure 1.

3. Web-based approach

Our web-based approach used Galaxy, "an open web-based platform for data intensive biomedical research". We used Galaxy because of our experience working on a prior project - A Portal for Data Analysis Services for Cosmological Simulations (PDACS) [8]. In order to use it for our purpose, we made several additions to Galaxy: (1) new tools, (2) new data types to represent tool outputs and to propagate metadata, (3) job submission support to FermiGrid and HPC resources at NERSC, and (4) integration of the NERSC and FermiGrid job monitoring interfaces.

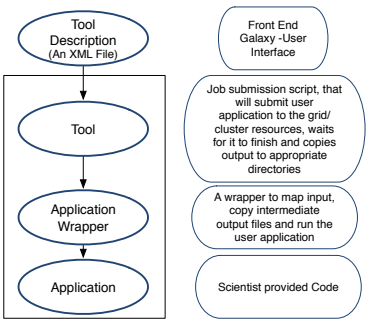

Figure 4. Steps to add a new application to the Galaxy framework.

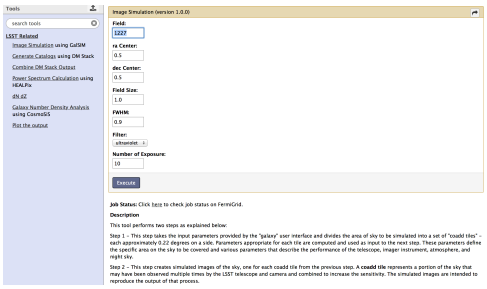

Figure 5. Example of an application interface in Galaxy. This shows the web interface for Image Simulation step.

Adding new tools to the framework: One adds new applications to the framework by wrapping it in a *Galaxy Tool*, as shown in Figure 4. The first step is to provide an XML file that describes the Galaxy interface, i.e. a tool description file containing input and output, their correct types, and an executable to be run. In the simplest situation, only this is required. However, if application is to be run at FermiGrid, or HPC resources at NERSC, then a *Tool file*, i.e. an additional job submission script is to be provided as well, which will be used in the XML file in place of application executable. Additionally, if necessary, a wrapper should be provided to abstract the inputs.

We created tools for all the applications shown in Figure 1. We provided tool description files, and tool files for the steps that run on FermiGrid and NERSC. The steps that were run locally only needed the description files. For the image simulation step, we also provided the

wrapper to limit the user controlled input parameters. Figure 5 shows the front end of Image Simulation step.

Adding new data types for metadata propagation and provenance tracking: Our scientific applications typically have a rich set of metadata needed as provenance, to ensure that constructed workflows will be reproducible. Galaxy uses *datatype* and *format* of a file to recognize data. If the datatype is already added to Galaxy, (the *metadata*) describing the contents of the file can be readily extracted and stored in Galaxy's metadata database. Users can view such metadata through Galaxy's web interface. The format of the file can be used to help prevent assembly of incorrect workflows.

We added the support for new datatypes necessary for this workflow, and the associated metadata can be forwarded along a workflow. The richer collection of metadata allows the user to identify the application that created the data, the inputs used to create that data set, and the configuration parameters used by the application that created it.

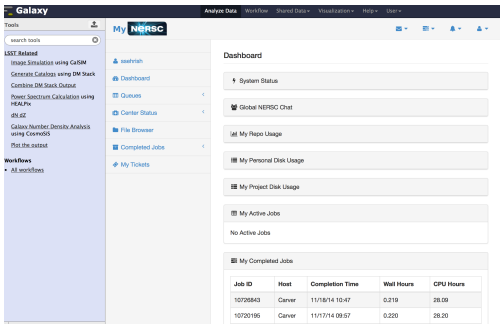

Figure 6. An example of NERSC job monitoring through Galaxy web interface.

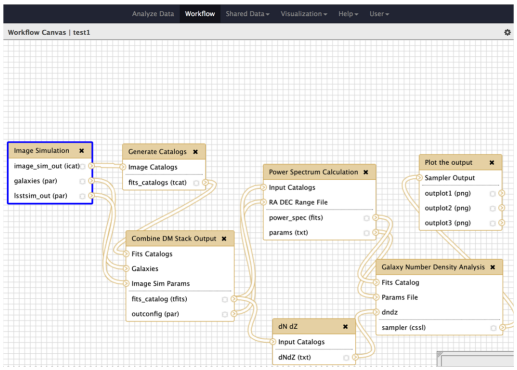

Figure 7. The Galaxy workflow editor; all the tools are connected with matching input and output types.

Job Submission and monitoring: Galaxy provides job runners and support for common schedulers to aid running jobs on different computing facilities. We set up job submission for FermiGrid and NERSC's Carver supercomputer. For FermiGrid, we used the `jobsub` command, which behind the scenes uses HTC Condor. For running on NERSC, we used NewtAPI [7] to submit jobs to Carver's queues.

Since there are web interfaces and monitoring pages available for both FermiGrid and NERSC, we integrated them into the Galaxy interface. Figure 6 shows NERSC's job monitoring page when a user is logged in.

Putting it all together: Figure 7 shows the Galaxy workflow editor, where users can drag and drop different tools, and connect the tools together to form a workflow. If correct datatypes for each tool are encoded in Galaxy, then users cannot create workflows with incorrect input/output. The input parameters for each of the tool in workflow can be configured by clicking on the tool and making changes to the inputs in the right hand side window.

4. Programming language-based approach

The second option we have explored is a command line based solution, where users describe their workflow as a sequence of steps to be carried out. We used Python as the language to describe workflows for the demonstration system. We provide new data types to ensure correct type checking and new wrappers for each application.

Adding new applications and data types: Applications and data types are represented as a Python class, i.e. a new class is created for each new application and for each input data type.

Listing 1 shows example of the code needed to support an application. In Listing 1, first two lines show import from two data types. `Tfits` represents "FITS" table format, and `Cltype` represents "power spectrum calculation" output, which is a fixed column format with specific names.

A `runapp` method is implemented for each application class as shown in line 9. Each `runapp` method consists of three fundamental steps: `checkinput` ensures the correct input type for this particular application, a "run" function (here `rundndz`) represents the execution of application, and `writeoutput` ensures that a data type object representing output of `dndz` application is created to be used in the subsequent steps.

Listing 1. Example of application wrapper and runapp method.

```
1  from datatypes import Tfits
2  from datatypes import Cltype

4  class rundndz:
5      def __init__(self, env):
6          self.env = env
7
8      def runapp(self, fitscat):
9          self.checkinput(fitscat)
10         fname = self.rundndz(fitscat)
11         dndzout = self.writeoutput(fname)
12         return dndzout
```

Putting it all together: After creating classes for all the applications and required data types, a workflow can be described in a Python script as shown in Listing 2. The import statements represent datatypes and applications in the workflow. The main body of the code has a pattern to it, i.e. instantiating a class and then calling a method from that class for each of the application in the workflow. An instance of an "application class" represents a configured application in the specified context (e.g. grid, local) ready to run. e.g. `lsstsim`. A call to `runapp` represents invoking the configured application in its context, providing the specified inputs and creating the specified outputs.

Listing 2. A Complete Workflow Description.

```
1  #!/usr/bin/env python

3  from datatypes import Par
4  from lsstsim import lsstsim
5  from dm import dm
6  from merge import merge
7  #more import statements for each type

9  conf = Par('config.par')
10 lsim = lsstsim('grid')
11 images, galaxies, params = lsim.runapp(conf)

13 dmcat = dm('grid')
14 catalogs = dmcat.runapp(images)

16 catalog, radec = merge('local').runapp(catalogs, galaxies, params)
17 dndzout = rundndz('local').runapp(catalog)
18 psout, pars = powerspec('local').runapp(catalog, radec)
19 cosmosisout = runcosmosis('nersc').runapp(psout, pars, dndzout)
20 likelihood = postprocess('local').runapp(cosmosisout)
```

In our prototype system, the Python script actually executes the workflow, and a drawback is that closing Python window will kill the workflow. Future work would include having the execution of the Python script write out a job description to be executed by one of several workflow execution systems.

5. Comparison

Implementing this one example use case revealed benefits to each of the approaches. Both allow scientists to run complicated workflows involving a variety of computational resources at physically different locations: GRID (at FNAL), HPC (at NERSC), and local interactive systems.

Adding a new application using the Python-based workflow system is straightforward; a Python wrapper class is written forming a proxy to the real application. Adding new applications to Galaxy, however, requires expert-level knowledge the configuration and functioning of the system. Applications communicate using specific kinds of data. Describing and incorporating new data types are important features of any workflow system; they permit correct interpretation of data and also for validation of workflows. Adding data types to Galaxy requires expert-level knowledge of the system protocols and GUI interactions. The Python workflow system requires creating a new class that describes data type that will be used by applications.

The built-in Galaxy facilities for sharing workflow and results with collaborators are feature-rich and easy to use. Currently, there are no direct collaboration facilities in the Python-based system.

Galaxy does not provide explicit debugging infrastructure necessary for a good development environment and error messages can be difficult to interpret. With the Python system, however, users are in full control of the environment and can interact with the underlying tools throughout development cycle.

Galaxy provide a straightforward means for incorporating existing web monitoring pages into the system. These pages can be viewed directly from the Galaxy application. For the Python-based system, users have to rely on external command line tools for job monitoring.

Applications that generate output files from external resources, such as those available on the GRID, require special handling within Galaxy. All produced files appear as history items within a user work space, and are managed on the Galaxy server. There is no direct way of browsing all the available files. In addition, any files of interest must be downloaded to the end-user machine. In contrast, all the output files go to a specified directory in the Python-based system, therefore a simple listing of the directory can provide direct access to the data.

If a workflow is well-tested and fairly static with a moderate number of configurable parameters, then using a system like Galaxy is a good option. If a workflow is under development and needs iterative modifications, testing, and debugging, then a command-line based solution such as the Python-based system is likely going to be a better choice.

6. Observations and conclusion

The cosmology use case we implemented is similar in structure to many HEP workflows: multiple data processing stages with intermediate files for communication between the stages. The job submission and data handling tools used in our exploration are commonly used within the HEP Intensity Frontier experiments affiliated with Fermilab. For example, we used `jobsub` to submit jobs to FermiGrid, CVMFS and dCache to setup software and facilitate data storage, and `ifdh` to copy files to and from dCache. Each approach we considered has features useful to the HEP community.

A web-based system like Galaxy has features to coordinate large configurable workflows composed of commonly used applications. This may be especially suitable interface for an experiment's production system, where neither new application code is added or modified. The workflows in experiment's production system are well-tested and well-defined, and operators (who may not be expert in programming) can submit and monitor the workflows alerting configurable parameters as necessary. The centralized configuration, submission, and monitoring over multiple compute facilities is also an attractive features these kinds of workflows.

In a pre-production environment, however, where algorithms are constantly being developed,

debugged, and tested, an interactive system with fast turn around time is the much preferred solution. In this situation a programming language-based system is a more appropriate choice, since it provides high-level abstractions that readily permit developers to add, modify, and debug new application components, rapidly integrating them into new workflows.

References

[1] CernVM File system. `http://cernvm.cern.ch`.
[2] COSMOS Catalog. `http://irsa.ipac.caltech.edu/data/COSMOS/`.
[3] dCache. `http://www.dcache.org`.
[4] Galaxy. `http://www.galaxyproject.org`.
[5] LSST Data Management. `http://dm.lsst.org`.
[6] LSST DESC Use cases Document. `https://confluence.slac.stanford.edu/display/LSSTDESC/Software`.
[7] NERSC Web Toolkit API. `https://newt.nersc.gov`.
[8] Ryan Chard, Saba Sehrish, Alex Rodriguez, Ravi Madduri, Thomas D. Uram, Marc Paterno, Katrin Heitmann, Shreyas Cholia, Jim Kowalkowski, and Salman Habib. PDACS: A Portal for Data Analysis Services for Cosmological Simulations. In *Proceedings of the 9th Gateway Computing Environments Workshop*, GCE '14, pages 30–33, Piscataway, NJ, USA, 2014. IEEE Press.
[9] K. M. Górski, E. Hivon, A. J. Banday, B. D. Wandelt, F. K. Hansen, M. Reinecke, and M. Bartelmann. HEALPix: A Framework for High-Resolution Discretization and Fast Analysis of Data Distributed on the Sphere. *Ap.J.*, 622:759–771, April 2005.
[10] B. T. P. Rowe, M. Jarvis, R. Mandelbaum, G. M. Bernstein, J. Bosch, M. Simet, J. E. Meyers, T. Kacprzak, R. Nakajima, J. Zuntz, H. Miyatake, J. P. Dietrich, R. Armstrong, P. Melchior, and M. S. S. Gill. GALSIM: The modular galaxy image simulation toolkit. *Astronomy and Computing*, 10:121–150, April 2015.
[11] J. Zuntz, M. Paterno, E. Jennings, D. Rudd, A. Manzotti, S. Dodelson, S. Bridle, S. Sehrish, and J. Kowalkowski. CosmoSIS: modular cosmological parameter estimation. *ArXiv e-prints*, September 2014.