Hybrid Reusable Computational Analytics Workflow Management with Cloudmesh

Gregor von Laszewski*
laszewski@gmail.com
University of Virginia
Charlottesville, VA, USA

J.P. Fleischer
jacquespfleischer@gmail.com
University of Virginia
Charlottesville, VA, USA

Geoffrey C. Fox
gcfexchange@gmail.com
University of Virginia
Charlottesville, VA, USA

Abstract

In this paper, we summarize our effort to create and utilize a *simple* framework to coordinate computational analytics tasks with the help of a workflow system. Our design is based on a minimalistic approach while at the same time allowing to access computational resources offered through the owner's computer, HPC computing centers, cloud resources, and distributed systems in general. The access to this framework includes a simple GUI for monitoring and managing the workflow, a REST service, a command line interface, as well as a Python interface. The resulting framework was developed for several examples targeting benchmarks of AI applications on hybrid compute resources and as an educational tool for teaching scientists and students sophisticated concepts to execute computations on resources ranging from a single computer to many thousands of computers as part of on-premise and cloud infrastructure. We demonstrate the usefulness of the tool on a number of examples. The code is available as an open-source project in GitHub and is based on an easy-to-enhance tool called cloudmesh.

CCS Concepts: • **Computer systems organization → Distributed architectures**; **Cloud computing**; • **Software and its engineering → Distributed systems organizing principles**.

Keywords: high performance computing, batch queue, service

1 Introduction

In this section we provide an introduction to our work while moving forward to motivate a Hybrid Reusable Computational Analytics Workflow Management Framework.

1.1 Reusable Computational Analytics

Reusable computational analytics (RCA) focuses on the creation of reusable programs, patterns, and services to conduct analytics tasks that are part of the scientific discovery process. RCA service need varies widely and may include multi-scale hardware resources as well as multi-scale scientific applications. To utilize such services and their resources in a reusable way, we need to have a mechanism to express them in an easy fashion that goes beyond just the definition in one programming language or framework, but allows the integration into many different programming languages and frameworks so that services that may be designed in one framework or language may be reusable in others.

1.2 Reusable Multi-scale Algorithms

In current scientific problems we encounter a rich set of applications that leverage a number of sophisticated methods that may require adaptations on multiple scales. The scales are influenced by their Domain size, accuracy and time requirement to solve them in a sufficient manner. It is of advantage to provide reusable components that can be controlled by parameters to simplify reuse.

1.3 Hybrid Cloud and Compute Resources and Serices

As we deal with multi-scale algorithms, not every analytics task needs to be conducted on a High Performance Computer (HPC). This is especially the case with the advent of desktop GPUs, which authors have termed in past *desktop supercomputing*. Also the availability of cloud computers and hyper-scale data centers play a significant role in today's analytics processes. This not only includes the use compute resources, but also services that are these days offered by cloud service providers. A well-known example for this is natural language processing.

1.4 Reusable and Adaptable HPC and Cloud Service Workflows

High-performance computing (HPC) has been, for decades, a very important tool for science. Scientific tasks can leverage the processing power of a supercomputer so they can run at previously unobtainable high speeds or utilize specialized hardware for acceleration that otherwise are not available to the user. HPC can be used for analytic programs that leverage machine learning applied to large data sets to, for example, predict future values or to model current states. For such high-complexity projects, there are often multiple complex programs that may be running repeatedly in either competition or cooperation. Leveraging computational GPUs, for instance, leads to several times higher performance when applied to deep learning algorithms. With such projects, program execution is submitted as a job to a typically remote HPC center, where time is billed as node hours. Such projects must have a service that lets the user manage and execute

*Corresponding author

without supervision. We have created a service that lets the user run jobs across multiple platforms in a dynamic queue with visualization and data storage.

Similar aspects are available for cloud services that abstract the infrastructure needs and focus on the availability of services that can be integrated in concert with HPC, as well as the users local resources (for example a PC).

2 Requirements

For the design of the framework, we have the following requirements.

Simplicity. The design must be simple and the code base must be small. This helps the future maintenance of the code, but also allows the code to become part of educational opportunities such as in a Research Experience for Undergraduates (REU), capstone project, and class projects. The challenge here is that many other systems are too big to be used in introductory undergraduate and masters activities. However, the framework should be capable enough to also support research projects.

Workflow specification. The specification of the workflow must be simple. From the past we have learned that the introduction of programming control components such as loops, and conditions, in addition to DAGs, is essential to provide maximum flexibility.

Workflow Monitoring. The workflow framework must be able to monitor the progress of individual jobs as well as the overall progress of the workflow.

Workflow Interfaces. To specify and interface with the workflow, we must provide several interface layers. This includes specification through a YAML file, interfacing through REST calls, interfacing through a Python API, and interfacing with a very simple GUI component. For the workflow monitoring with the GUI, we must be able to easily define custom text to reflect user designable monitoring labels.

Language Independence. As we want to make the framework integrable in other frameworks, we need a simple mechanism to provide either API or interface portability. To keep the code base small, a REST API seems very well-suited.

OpenAPI. To further strengthen usability, the framework must have an OpenAPI interface. This allows the integration via 3rd party tools into other frameworks and languages, if desired.

Hybrid multicloud Providers. The service must be able to be deployable on an on-premise local computer or various cloud providers.

Generalized Job Interface. The framework must be able to interface with a wide variety of computation services. This includes the support of ssh, batch queues such as Slurm and LSF, and local compute resources including shell scripts, as well as support for WSL.

Support of various Operating Systems. The framework must be runnable on various client operating systems including Linux, macOS, Windows.

State Management. The state management must be recoverable and the client must be able to completely recover from a network failure. As such, the state will be queried on demand. This allows one to deploy the framework on a laptop, start the workflow, close or shutdown the laptop, and at a later time open the laptop while the workflow framework can be refreshed with the latest state.

3 Related Research

Many different workflow systems have been developed in the past. It is out of the scope of this document to present a complete overview of all the different systems. Instead, we compare some selected features of the systems and identify features that are important for us. The following features are important.

Batch parallelism. Long-running jobs on HPC systems are coordinated through Batch services.

Task parallelism. Tasks are a logical unit of work that is executed by a resource, service, or component. Task parallelism is often used in distributed resource frameworks.

Resource Reservation. In some cases, access to batch queues on HPC systems take a long time. In the case of many different tasks, it is sometimes useful to research a number of batch nodes and run many short-running programs on them if the need arises. This, however, can often be replaced just with properly coordinated workflows using a batch system. In fact, frameworks using such reservations internally implement them using the batch system.

Computational Grids. Grids provide service-level access to distributed HPC computing resources. However, Grids (popular a decade ago) are no longer predominantly deployed and the focus has shifted to Cloud computing.

Cloud. Presently, computing resources are also available in the cloud as HPC, batch, and compute services, including specialized SaaS offerings that allow to integrate analytics functions into workflows.

REST. The predominant specification for cloud services uses the REST framework relying on a stateless model. This contrasts with the WSRF model that uses stateful services.

Specification. Some work has also focused on the specification of the workflows.

GUI. Some frameworks provide extensive GUIs.

In Table 1, we provide a number of examples for the various features we found in workflow systems.

4 Design

To fulfill our requirements, we have developed a framework for workflow-controlled computing called Cloudmesh compute cluster, otherwise referred to as *Cloudmesh-cc*. The architecture of the framework is depicted in Figure 1.

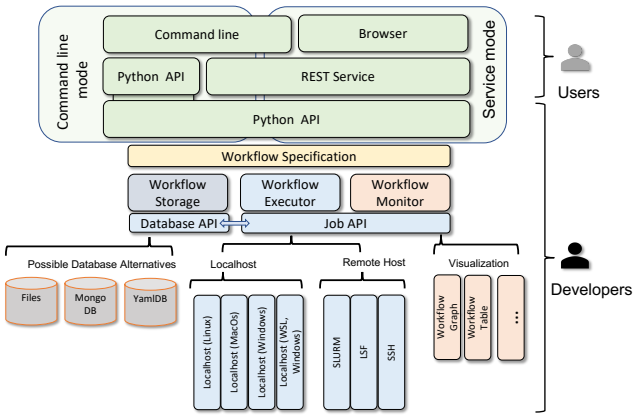

Figure 1. Architecture of the Cloudmesh-cc Framework

The framework is based on a layered architecture so it can be improved and expanded on at each layer. We distinguish two kinds of users, developers and end users, that define their workflows to run on compute resources. For the end user, we provide a command line and a very simple browser interface. Simplicity is important as it does not require spending exorbitant amounts of time to learn how to use the workflow framework. To support developers, we have designed a minimal Python API that is also used to implement a REST Service. Currently, the REST Service is supposed to run on the client, but a service deployment could also be conducted while assuring that proper authentication and authorization are used. This is easily possible as we use a well-known REST service framework (FastAPI) that can integrate with common security frameworks to secure services. The API, workflow specification, command line interface, and browser are well documented with the help of Sphinx and OpenAPI.

The workflow specification plays an important role in not only defining a workflow but also keeping the status of a currently executed workflow. Here we have completely separated the status of the workflow and synchronize the state of the workflow with pull requests to the service that executes the computation. This allows the system to be shut down at any time while the running jobs are completely independent of the client application accessing the state. Thus, the client appears to be stateless and fetches the state of the submitted jobs on demand. It will return the latest state found on the job execution services.

The workflows are defined with YAML. The workflow is stored in a database that can be implemented using a variety of backends such as YamlDB (which is file-based), MongoDB, or pickle. In our current implementation, we simply use a file-based implementation as it is not required to set up and manage more complex databases. This is an important capability as we found that scientists and beginning students do not want to engage in the hassle of setting up and managing a datastore such as MongoDB.

As we use a YAML file to represent the status of the workflow, it is easy to create monitoring components (for example, as part of a Web browser). Various sophisticated graph display frameworks could be used. For now, we have simply exposed the graph in table format using datatables.net and the graph as SVG while leveraging Graphviz.

One of the most important parts of the framework is how we manage jobs and monitor their status. For this, we have introduced an abstract job class that is integrated into our workflow class. The job class can define jobs, start them, and cancel them, to name only the most important management methods. However, each job defines a status file in which the actual progress of the job is recorded. This status file is managed on the compute resource where the job is run and is queried on demand to return the status to the client. This way, the status of all jobs can be monitored. As our goal is not to run jobs that execute in milliseconds, but rather in the second range, such status reporting and propagation is well-suited for us. We have defined a special status progress update specification that is universally applicable. These jobs can be bash scripts, Python scripts, Jupyter notebooks, or Slurm scripts.

Workflows are compilations of jobs to be run on nodes. These workflows report information on the status of jobs as they are run, whether locally or remotely. These types of jobs can be mixed with others into a single workflow. This allows us, as already mentioned, to view the progress of a workflow as it is being run in quasi-realtime. The workflow can be monitored as a graph or a DataTable. These interfaces report the statuses of the jobs. Additionally, the order in which the jobs are run can be specified, enabling prerequisite jobs and the segmentation of a workflow.

To address the requirement of simplicity, the overall code is less than 3600 lines of code, with additional 2600 lines for extensive test cases.

The framework is managed as an open-source repository in GitHub and uses Python as the implementation language. The code is compatible with Windows, macOS, and Linux.

5 Specifying Workflows

The workflow definition for cloudmesh is rather simple and intuitive. An example is provided in Figure 2. Here, a DAG with three nodes are specified ($start \rightarrow fetch - data \rightarrow compute \rightarrow analyze \rightarrow end$). The workflow executes three

Table 1. Example of workflow frameworks with selected features.

Name	Selected Features	Description
LSF [14]	Batch, HPC	Batch Queue Manager.
SLURM [24]	Batch, HPC	Batch queue workflow manager.
Cloudmesh [28]	HPC, Cloud, on-Premise, Task	Suite of Python code for cloud computing.
Airflow [1]	HPC, Cloud, Remote Tasks	"Airflow is a platform created by the community to programmatically author, schedule, and monitor workflows."
Loosely coupled Metacomputer [26, 27]	HPC, on-Premise	Early work on workflow management system connecting multiple supercomputers.
Karajan CoGkit [31]	Service, Language, HPC, on-Premise	Sophisticated Workflow management system with Dag and loops for Grid computing and loosely coupled compute resources through services.
Snakemake [25]	Language	"The Snakemake workflow management system is a tool to create reproducible and scalable data analyses. Workflows are described via a human-readable, Python-based language. They can be seamlessly scaled to server, cluster, grid, and cloud environments, without the need to modify the workflow definition. Finally, Snakemake workflows can entail a description of the required software, which will be automatically deployed to any execution environment."
Gridant [6]	Language	Client-based workflow management toolkit to orchestrate complex workflows on the fly without substantial help from the service providers. It integrates on premise and Grid resources.
Keppler [15]	Service, GUI	"Kepler is designed to help scientists, analysts, and computer programmers create, execute, and share models and analyses across a broad range of scientific and engineering disciplines."
Pegasus [23]	DAG, Service, HPC, Cloud	A DAG-based workflow management tool for scientific workflow.
Swift [32]	Language, Resource Reservation	A language for distributed parallel scripting.
Parsl [7]	Language, Resource Reservation	A language for distributed parallel scripting.
Radical Pilot [17]	Resource Reservation	Pilot system that enables scalable workflows
Gcloud workflow [8]	Cloud	Google Cloud documentation for creating a workflow with the command line interface.
Azure REST [21]	Cloud	Microsoft Azure REST API documentation to interface with Azure Logic Apps.
GitHub REST Cancel a Workflow Run [11]	GitHub Actions	GitHub REST API documentation to cancel a GitHub Workflow.
Azure Enterprise Workflow [9]	GUI, Workflow	Enterprise Workflows with Azure Logic Applications
WSDL [10]	Specification	Microsoft Azure schema reference for the Workflow Definition Language in Azure Logic Apps.
WSRF [20] AWS Workflow [4]	Specification	OASIS Web Services Resource Framework Amazon Web Services documentation definition of a workflow.
AWS SWF [5]	Task	Amazon Web Services documentation for managing simple workflows with tasks.
AWS Step Functions [3]	GUI, Task	Amazon Web Services documentation for visual workflow service.
AWS HPC [2]	Cloud, HPC	Amazon Web Services documentation for setting workflow dependencies using batch processing in HPC on the cloud.
Azure Batch [18]	Cloud, Batch	Microsoft Azure documentation for running batch service workflows.
Business Automation with REST [13]	Business Process	IBM REST API documentation for its Business Automation Workflows.
Infogram [30]	Service	A Peer-to-Peer Information and Job Submission Service.

scripts (test-[fetch-data,compute,analyze].sh). It contains a specific start and end node.

5.1 Dependencies

Each dependency is specified in the *dependency* section while providing sequences of names in a list such as -start,fetch-data or, in our case where 3 more nodes are defined, additional nodes are appended with commas, as in ,compute,analyze,end.

In case one wants to execute nodes in parallel, we can simply define them through a list such as

```
dependencies:
  - start,a,end
  - start,b,end
  - start,c,end
```

if the names of the nodes were *a*, *b*, and *c*.

5.2 Nodes

Nodes can be customized in various ways within the workflow configuration YAML file, including their job types (python, sh, jupyter, or slurm), their virtual Python environment (by specifying venv), their appearance on the graph, and other characteristics. This is controlled through a number of attributes used by the nodes in the DAG. The attributes are summarized in Table 2.

```
workflow:
  nodes:
    start:
       name: start
    fetch-data:
       name: fetch-data
       user: gregor
       host: localhost
       kind: local
       status: ready
       label: '{name}\nprogress={progress}'
       script: test-fetch-data.sh
    compute:
       name: compute
       user: gregor
       host: localhost
       kind: local
       status: ready
       label: '{name}\nprogress={progress}'
       script: test-compute.sh
    analyze:
      name: analyze
      user: gregor
      host: localhost
      kind: local
      status: ready
      label: '{name}\nprogress={progress}'
      script: test-analyze.sh
    end:
       name: end
  dependencies:
    - start,fetch-data,compute,analyze,end
```

Figure 2. Workflow YAML Configuration file.

Table 2. Node attributes.

Attribute	Description
name	A unique name of the job, must be the same as defined in the : line
user	The username for the host
host	The hostname
kind	The kind of the job, which can be local, ssh, wsl, or slurm
status	The status of the job in integer value between 0 and 100
label	A custom-designed label
script	The script name to be executed

5.3 Node labels

One particular useful attribute is that of a label. If no label is used, the name of the node is used as the label. However, if a label is specified, one can also use attribute names and timers to create labels with implicit state information. This is done by introducing variables through curly braces when defining the labels inside the label defined in nodes within the YAML workflow file.

For example, a label could be defined as showcased in Figure 3. The appropriate values will be dynamically replaced during the execution of the workflow. This creates a node on the graph that looks similar to the node showcased in Figure 4.

Initially, the created and elapsed labels are N/A if the workflow has not yet started, but they are replaced during runtime. This can be observed by running a workflow in graph view in the web interface.

In this format, we must use two dashes -- to separate the various components. However, when rendered, the dashes will be replaced with a colon. Thus you can easily use years, months, days, hours, minutes, and seconds can be arranged as desired, as long as the corresponding letters remain consistent: (%Y %m %d %H %M %S). The time format must be specified immediately following the period after a format-supported time variable. If no format is specified following the period after the variable, the datetime defaults to the American format. See Table 3 for a summary of options for defining time based attributed to being replaced in the label.

Nodes can be customized in various ways within the workflow configuration YAML file, including their job types (python, sh, jupyter, or slurm), their virtual Python environment (by specifying venv), their appearance on the graph, and other characteristics.

```
workflow:
  nodes:
    start:
      label: 'start\n
             Created={created.%Y/%m/%d, %H--%M--%S}\n
             Workflow Started={t0.%Y/%m/%d, %H--%M--%S}\n
             Now={now.%Y/%m/%d, %H--%M--%S}\n
             Elapsed={dt0.%M--%S}'
```

Figure 3. Example of job label in YAML configuration file.

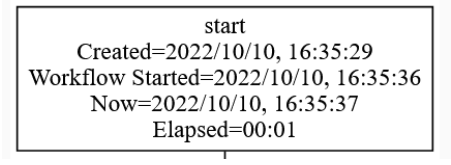

Figure 4. An example node with labels.

5.4 Shapes and Styles

As we use graphviz for rendering, we have also added the ability to change the shape and style of each node in the graph. The available shapes and styles are listed in the Graphviz documentation [12].

Figure 5 is an example of a node in YAML format that uses a box shape and an empty style. The empty style defaults

Table 3. List of possible labels for nodes on the graph.

Name	Description
progress	progress of job from 0-100
now	current time
now.%Y%m%d,%H--%M--%S	now in particular format (this can be used for other times as well)
created	time when workflow was created
t0.%Y%m%d,%H--%M--%S	workflow start time
t1.%Y%m%d,%H--%M--%S	workflow end time
dt0.%Y%m%d,%H--%M--%S	elapsed time since workflow began
dt1.%Y%m%d,%H--%M--%S	total time of workflow once complete
tstart.%Y%m%d,%H--%M--%S	job start time
tend.%Y%m%d,%H--%M--%S	job end time
modified.%Y%m%d,%H--%M--%S	job modified time
os.	operating system environment variable (like os.HOME)
cm.	cloudmesh variable that is read from `cms set`

to `filled`, which allows the node to change color when the job status is changed.

```
workflow:
  nodes:
    start:
      label: 'start\n
              Created={created.%Y/%m/%d, %H--%M--%S}\n
              Workflow Started={t0.}\n
              Elapsed={dt0.}'
      kind: local
      user: grey
      host: local
      status: ready
      exec: 'echo hello'
      name: start
      shape: box
      style: ''
```

Figure 5. Example of YAML config file that uses shape and style.

5.5 Reporting Progress

When running scripts/jobs inside a workflow, the scripts must leverage some format of *cloudmesh.progress* to notify the user and the backend client monitoring system. If progress is not reported, the Workflow class cannot tell if the scripts are done.

The examples that are provided with cloudmesh-cc are already augmented with *cloudmesh.progress*. Thus, if a user is running jobs through cloudmesh cc workflows, they must integrate progress strings into the log files that are monitored. This is available for shell, batch, Python, and Jupyter scripts.

For shell and Slurm scripts, the script must contain a progress update lines as follows:

```
echo "# cloudmesh status=running progress=1 pid=$$"
```

at the beginning of the script, and

```
echo "# cloudmesh status=done progress=100 pid=$$"
```

at the end of the script.

For Python scripts and Jupyter notebooks, it is easiest to use our built-in progress method and import it from the cloudmesh.common module as the example in Figure 6.

```
from cloudmesh.common.StopWatch import progress
from cloudmesh.common.Shell import Shell
filename = Shell.map_filename('./py_script.log').path
progress(progress=1, filename=filename)
... execute your analysis here ...
progress(progress=100, filename=filename)
```

Figure 6. Progress report with the cloudmesh Python API.

When recording progress, the progress should be ascending between 1 and 100. The last value must report 100 or the node will not be completed and the workflow gets stuck. The output of the progress will be written into a `{workflowname}.log` file that will be probed continuously by the client to report the progress to the user.

5.6 Generating Progress Image as Graph

The framework has a built-in capability to export the progress of the workflow in a DAG.

In Figure 7, we depict a simple example workflow to be executed, where each task is executed sequentially. The status of the execution can be displayed as a table or as a graph. In Figure 7, we showcase how the graph changes its appearance over time while using no label in the start and end node and a defined label in the nodes for fetch-data, compute, and analyze:

```
label: "{name}\nprogress={progress}"
```

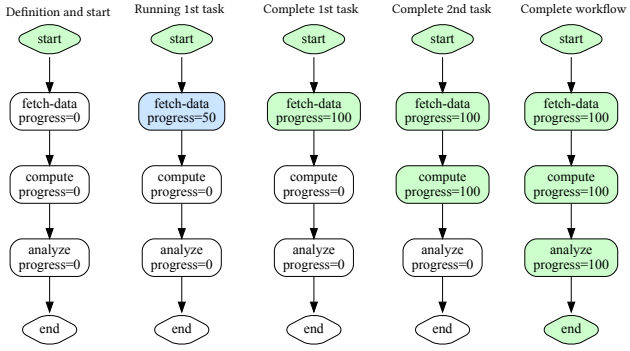

Figure 7. The gradual process of a simple workflow.

6 Cloudmesh-cc Interfaces

In this section, we explain the various ways of interfacing with cloudmesh-cc that are part of our design (see Figure 1).

It is important to note that we have a *command line mode* that interfaces directly with the backends while not requiring a service. This includes an easy-to-use Python API.

In addition, we have used this API to implement a *service mode*, so we can stand up a REST service as well as a GUI that

can be accessed through a Web browser. Please note that the service mode can also be accessed through the command line in a terminal. To distinguish how we operate cloudmesh-cc, we use the term *mode* to delineate it from a command that is entered in a terminal to query the status of the workflow.

We will now discuss these interfaces in more detail and also showcase how we can access them.

6.1 Python API

Cloudmesh-cc is implemented in Python. Extensive documentation is available in GitHub and, using GitHub Actions, it is automatically updated [29]. We distinguish two main classes that are easy to use. The first is a *Job class* that can be adapted to include new computational resource types for executing jobs. The second is a *Workflow class* to coordinate the execution of multiple jobs. Selected methods of the Job class include are listed in Figure 8. Important to note is that the code for the scripts are all managed locally, and a synchronization step is invoked prior to one running the job to assure that the latest script and code to be executed on the compute resource is available.

The most important part of using the Workflow Python API is showcased in Figure 8, where we explain how easy it is to set up a workflow with the Python API.

6.2 Command Line Mode

The command line mode is an implementation that does not use a backend service. It runs in the terminal until the workflow is completed. All states of the jobs are managed on the compute service on which the job is run and the status is replicated into the client on demand. This allows easy reporting of the status through a table and a graph display using client-based rendering tools. The various commands to interact with workflows on the command line are shown in Figure 11.

6.3 Service Mode

The Python API has been used to implement a REST service. To easily interact with the REST service we have also added a couple of convenient commands one can issue from a terminal. They are listed in Figure 12. It is clear from these methods that starting, stopping, and getting the status of the service is very easy. In contrast to the command line mode, this service mode has an additional keyword *service* to interact with the rest service.

Certainly one can also directly interact with the REST service API for this service. For ease of use, we have exposed the interface through an OpenAPI specification that is available to the user as shown in Figure 13.

Thus, other tools such as curl or other languages supporting URL requests can be used. A curl example to list the workflow specifications uploaded to the service is as follows:

```
$ curl -X 'GET' 'http://127.0.0.1:8000/workflows' -H 'accept: application/json
```

```
class Job:
   ...
   def clear():
      """Clear job progress."""
   def create(filename=None, script=None, exec=None):
      """Create a template for the script with progress."""
   def get_log(refresh=True):
      """Get the log of the job."""
   def get_pid(refresh=False):
      """Get the pid that the job is running within determined by
         the compute resource it is running on."""
   def get_progress(refresh=False):
       """Get the progress of the job from the compute resource."""
   def get_status(refresh=False):
       """Get the status of the job."""
   def kill():
       """Kill the job."""
   def run():
       """Run the job."""
   def sync():
       """Synchronise the current directory with the remote. Copies
          the shell script to the experiment directory and ensures
          that the file is copied with the sync command."""
   def watch(period=10):
       """Watch the job and check for changes in the given period."""
```

Figure 8. Pseudo code for the Job class with selected methods.

```
class Workflow:
    ...
    def add_dependencies(dependency):
        """Add a job dependency to the workflow (and the graph)."""
    def add_dependency(source, destination):
        """Add a job dependency to the workflow (and the graph)."""
    def add_job( ... ):
        """Add a job to the workflow with appropriate parameters."""
    def display(filename=None, name='workflow', first=True):
        """Show the graph of the workflow."""
    def job(name):
        """Return the details of a job within the workflow."""
    def load(filename, clear=True):
        """Load the workflow."""
    def remove_job(name, state=False):
        """Remove a particular job from the workflow."""
    def remove_workflow():
        """Delete workflow from the local file system."""
    def run_parallel( ... ):
        """Run a workflow in a parallel fashion."""
    def run_topo(order=None, dryrun=False, show=True, filename=None):
        """Run the workflow in a topological order."""
    def save(filename=None):
        """Save the workflow."""
    def save_with_state(filename, stdout=False):
        """Save the workflow with state."""
    def sequential_order():
        """Return a list of the topological order of the workflow."""
    property table:
        """Return a table of the workflow."""
    def update_progress(name):
        """Manually update the progress of a job according to its log
           file."""
    def update_status(name, status):
        """Manually update a job's status."""
```

Figure 9. Pseudo code for the Job class with selected methods.

As we use a REST service, we can also easily upload the workflow through a Python-enabled REST call. We will use Python requests to demonstrate this upload feature. To showcase the various ways to access the service, we focus on uploading a tar file that contains the workflow as well as

```
w = Workflow(name="workflow-analyze")
# load a preexisting workflow
# w.load(filename="source.yaml")

# add jobs and dependencies explicitly
w.add_job(name="fetch-data",
          exec="hostname",
          host="supercomputer-a", # defined in .ssh
          label="{name}\nprogress={progress}",
          kind="local",
          status="ready",
          progress=0)
w.add_job(name="compute", command="... TBD ...")
w.add_job(name="analyze", command="... TBD ...")
w.add_dependencies(
          dependency="start,fetch-data,compute,analyze")
w.run_topo()
```

Figure 10. Pseudo code for the Job class with selected methods.

```
cms cc workflow add [--name=NAME] [--job=JOB] ARGS...
cms cc workflow add [--name=NAME] --filename=FILENAME
cms cc workflow delete [--name=NAME] [--job=JOB]
cms cc workflow list [--name=NAME] [--job=JOB]
cms cc workflow run [--name=NAME]
                    [--job=JOB]
                    [--filename=FILENAME]
cms cc workflow [--name=NAME] --dependencies=DEPENDENCIES
cms cc workflow status --name=NAME [--output=OUTPUT]
cms cc workflow graph --name=NAME
```

Figure 11. Command line interface to the workflow in terminal mode.

```
cms cc start [-c] [--reload] [--host=HOST] [--port=PORT]
cms cc stop
cms cc status
cms cc workflow service add [--name=NAME] FILENAME
cms cc workflow service list [--name=NAME] [--job=JOB]
cms cc workflow service add [--name=NAME] [--job=JOB] ARGS...
cms cc workflow service run --name=NAME
```

Figure 12. Command line interface to the workflow in service mode.

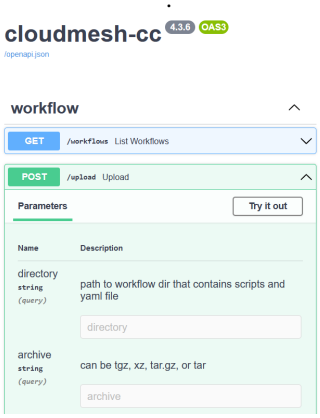

Figure 13. Browser API GUI for Cloudmesh Compute Cluster.

the scripts or programs used in the workflow. The tar file is called *workflow-example.tar*.

```
class RESTWorkflow
    ...
    def add_job(workflow_name, **kwargs):
        """Add a job to the workflow."""
    def delete_workflow(workflow_name, job_name=None):
        """Delete a workflow by using REST."""
    def get_workflow(workflow_name, job_name=None):
        """Retrieve a workflow by using REST."""
    def list_workflows():
        """Return a list of workflows that is found within
           the server."""
    def run_workflow( ... ):
        """Run a workflow by using REST."""
    def upload_workflow( ... ):
        """Upload a workflow by using REST."""
```

Figure 14. Pseudo code for the Job class with selected methods.

```
import requests
r = requests.post(
    'http://127.0.0.1:8000/workflow?archive=workflow-example.tar')
print(r.text)
```

Figure 15. Upload to the REST service with Python requests.

```
$ curl -X 'POST' 'http://127.0.0.1:8000/
                  workflow?archive=workflow-example.tar'
       -H 'accept: application/json' -d ''
```

Figure 16. Upload to the REST service with curl.

To also allow programming against the REST service in Python, a Python API similar to that of the command line mode is available. Figure 14 showcases this API.

To simplify interaction with the REST service, we have created a special *RESTWorkflow class* that is similar to the command module API, but instead uses the REST interface to the service rather than direct communication with the command client API.

The user can also simply use the requests module in Python to interface with the API. Figure 15 demonstrates how to use requests to upload a workflow by using an archive file that contains the YAML configuration file and the scripts.

6.4 Webservice GUI

A convenient Web service is included in Cloudmesh cc. It allows the user to manage and visualize the status of workflows through a Web browser interface. At this time, the focus is that the interface can be run by a single user on the local machine. This allows remote executions of workflow nodes run completely independent from cloudmesh cc and interaction is possible in asynchronous mode.

As the service is using also an OpenAPI 2.0 specification, the workflow can also be uploaded implicitly through the specification GUI. Navigate to `http://127.0.0.1:8000/docs` and use the POST Upload method. Then click `Try it out` and enter the location of the tar file, followed by clicking `Execute`.

Obviously, any REST service or REST API can be used, allowing the user to interface to it from different programming languages or frameworks.

The web server provides a more customizable, easy-to-use interface for the Workflow class which can be started, viewed, and stopped with the appropriate command suc as

```
$ cms cc start
$ cms cc view
$ cms cc stop
```

The view can also be achieved by opening the link `http://127.0.0.1:8000/`.

The browser provides an interface to view preexisting workflows in both a DataTable format and as a graph format. Both views will update in a live, automatic fashion as the workflows are run, reporting dynamic job status and progress.

For a quick and easy example of leveraging this GUI interface, click on the Example tab in the left-hand sidebar. Then, a workflow-example will appear underneath Workflows. Click on the workflow-example and run the workflow by clicking the green Run button in the top-right. As the workflow runs, the user is able to click on the Graph button to view the graph interface (see Figure 18) and back to the Table button for the table interface (see Figure 17), as desired, to view the workflow's progression.

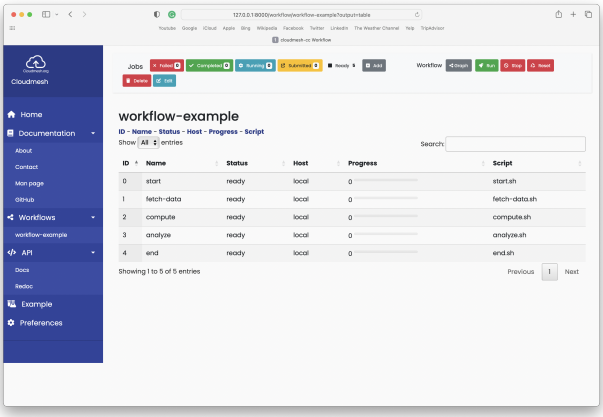

Figure 17. Cloudmesh cc workflow table view.

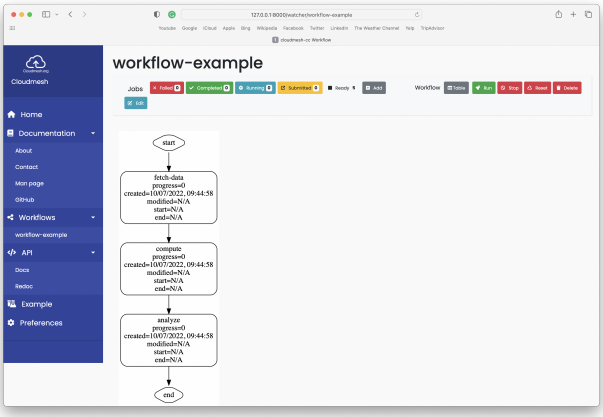

Figure 18. Cloudmesh cc workflow graph view.

6.5 Run the workflow

The workflow can be run easily via the GUI. We added a special set of buttons to the workflow table and graph display to simplify running of the workflow. Certainly, the workflow can also be activated while calling the appropriate REST call, either through Python, the OpenAPI docs page, or, for example, a curl call. Figures 19-21 showcase various methods to run the example workflow, which is called *workflow-example*.

```
$ curl -X 'GET' 'http://127.0.0.1:8000/
                 workflow/run/workflow-example?show=True'
       -H 'accept: application/json'
```

Figure 19. Running the example workflow with curl.

```
from cloudmesh.cc.workflowrest import RESTWorkflow
rest = RESTWorkflow()
result = rest.run_workflow('workflow-example')
```

Figure 20. Running the example workflow with cloudmesh RESTWorkflow API.

```
 import requests
url = f'http://127.0.0.1:8000/workflow/run/workflow-example?show=True'
 r = requests.get(url)
 print(r)
```

Figure 21. Running the example workflow with requests API.

7 Other Cloudmesh Features

Cloudmesh comes with a sophisticated package management system, allowing the integration of packages on demand targeting various providers and capabilities including a built-in command shell (not just only a command line tool). Cloudmesh was first developed as a hybrid cloud API, command line and command shell framework. It provided interfaces to AWS, Azure, Google, and OpenStack clouds for virtual machine[1] and data file services. It is characterized by defining default templates for virtual machine management on these clouds. Hence, it was possible to switch between clouds with only a few commands and stage virtual machines on them, such as with the commands demonstrated in Figure 22.

```
$ cms vm start --cloud aws
$ cms vm start --cloud azure
```

Figure 22. Simple VM management for hybrid clouds

.

In addition, we have developed a package called GAS that addresses the creation of analytics REST services from

[1]cloudmesh also provided support for clouds that are no longer supported such as Eucalyptus and Open Cirrus. Academic clouds such as Chameleon Cloud were also supported.

Python functions. This package was developed to address the problem that integrating deployment frameworks in the age of cloud computing is often out of reach for domain experts. GAS is a simple framework allowing even non-experts to deploy and host services in the cloud. To avoid vendor lock-in, it supports multiple vendors through the use of cloudmesh vm management [16].

8 Workflow Applications

We have applied the workflow system on a number of applications. All workflows for these applications are available in our GitHub and can be adapted easily. This includes the Cloudmask and MNIST application workflows which we describe next in more details.

8.1 MLCommons Cloudmask Workflow

Cloudmask is a program that develops a model to classify sections of satellite images as either containing clouds or clear sky by using machine learning. This is beneficial for temperature measurement and meteorology. Information regarding Cloudmask can be found on its GitHub page [22]. One of our goals is to run Cloudmask for benchmarking. As benchmarking Cloudmask requires several phases and scripts, including a mixture of shell scripts and Python scripts, leveraging Cloudmesh-cc provides a much easier runtime instead of manually issuing many commands at a terminal. We have created a sample workflow the runs a coordinated workflow across a number of hybrid resources. This includes an HPC computer at the University of Virginia called Rivanna, as well as two desktop computers. This workflow can easily be adapted to include other machines. In this particular workflow, we execute the benchmarks on a number of different CUDA cards (see Figure 23. For Rivanna, the code also utilizes our cloudmesh-vpn component that provides the ability to connect to the UVA VPN from Python, then fetches the data and executes the various benchmarks once the data is available.

The workflow will take approximately 24 hours to run if resources are available. The workflow iterates through the five GPUs available on Rivanna, including V100, P100, A100, K80, and RTX2080, and runs the program three times on each GPU. Each run trains the model with 10, 30, and 50 epochs for benchmarking. Upon completing a run, the logs and benchmarks are written into a results folder. In the appendix, we showcase how to run a portion of this workflow while utilizing only Rivanna (see Appendix A.2).

8.2 MNIST Workflow

MNIST is a well-known program to detect handwritten digits. It provides value for our work because it is well understood and is used in many educational efforts. Also, we created a workflow that integrates the UVA Rivanna HPC (the workflow can easily be adapted to other machines). The nice feature about this application is that it can be configured to run very quickly while still using various GPUs and benchmark their runtimes for running several MNIST Python programs. These programs include machine learning processing, convolutional neural network, long short-term memory, recurrent neural network, and others. The programs can be found on GitHub [19].

As for the workflow, we adapted it not only to run one algorithm but multiple in an iteration across the GPUs (similar to 23).

On a successful run, the output will be receiving runtimes similar to:

Table 4. MNIST Performance as obtained by cloudmesh-cc on various graphics cards using workflow scheduling

Name	Time
a100	106.046
v100	138.087
rtx2080	138.048
k80	171.057
p100	202.055

9 Conclusion

We have designed and implemented a Hybrid Reusable Computational Analytics Workflow Management with the help of the cloudmesh component framework. The component added focuses on the management of workflows for computational analytics tasks and jobs. The tasks can be executed on remote resources via ssh and even access queuing systems such as Slurm. In addition, we can integrate the current computer on which the workflow is running. This can include operating systems such as Linux, macOS, Windows, and even Windows Subsystem for Linux. Through cloudmesh, access to a command line and a command shell is provided. A simple API and a REST interface are provided. The framework also has an elementary Web browser interface that allows visualizing the execution of the workflow. It is important to know that the workflow can be started on remote resources and is running completely independently from the client tool once a task is started. This allows a “stateless” model that can synchronize with the remotely started jobs on demand. Hence, the framework is self-recovering in case of network interruptions or power failure. Due to our experiences with real (and many) infrastructure failures at the authors’ locations, the availability of such a workflow-guided system was beneficial. Furthermore, the developed code is rather small and, in contrast to other systems, is less complex. Hence, it is suitable for educational aspects as it is used for master’s and undergraduate level research projects. The project has also been practically utilized while generating benchmarks

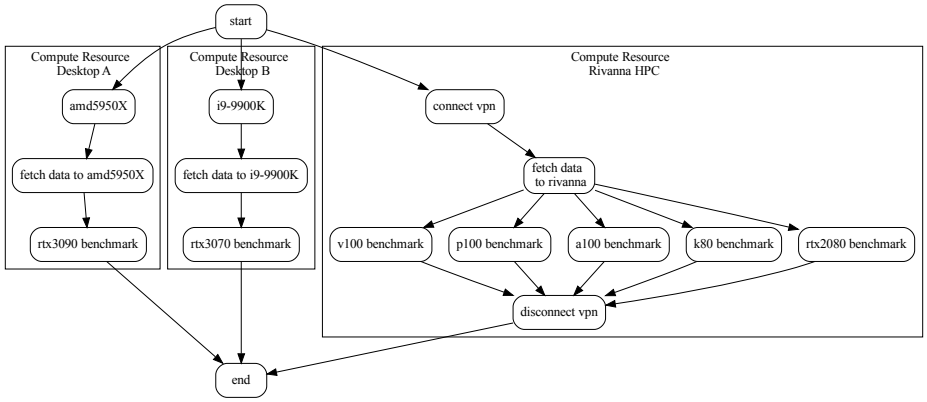

Figure 23. Workflow for Cloudmask

for the MLCommons Science Working Group showcasing real-world applicability beyond a student research project.

Acknowledgements

Continued work was in part funded by the NSF CyberTraining: CIC: CyberTraining for Students and Technologies from Generation Z with the award numbers 1829704 and 2200409 and NIST 60NANB21D151T. We would like to thank the students that participated in the REU for their help and evaluation in a pre-alpha version of the code. We are excited that this effort contributed significantly to their increased understanding of Python and how to develop in a team using the Python ecosystem.

References

[1] airflow 2022. Apache Airflow. https://airflow.apache.org [Online; accessed 14. Oct. 2022].

[2] Amazon Web Services. 2022. Encoding workflow dependencies in AWS Batch | Amazon Web Services. https://aws.amazon.com/blogs/hpc/encoding-workflow-dependencies-in-aws-batch [Online; accessed 14. Oct. 2022].

[3] Amazon Web Services. 2022. Serverless Workflow Orchestration AWS Step Functions. https://aws.amazon.com/step-functions [Online; accessed 14. Oct. 2022].

[4] Amazon Web Services. 2022. What is a Workflow? Cloud Computing Workflows Introduction - AWS. https://aws.amazon.com/what-is/workflow [Online; accessed 14. Oct. 2022].

[5] Amazon Web Services. 2022. What is Amazon Simple Workflow Service? - Amazon Simple Workflow Service. https://docs.aws.amazon.com/amazonswf/latest/developerguide/swf-welcome.html [Online; accessed 14. Oct. 2022].

[6] Kaizar Amin, Mihael Hategan, Gregor von Laszewski, estor J. NZaluzec, Shawn Hampton, and Albert Rossi. 2004. GridAnt: A Client-Controllable Grid Workflow System. In *37th Hawai'i International Conference on System Science*, Vol. 7. IEEE Computer Society, Los Alamitos, CA, USA, Big Island, HW. https://doi.org/10.1109/HICSS.2004.1265491 The original paper is: von Laszewski, Gregor and Kaizar Amin and Shawn Hampton and Sandeep Nijsure. Technical report and Argonne National Laboratory and 31 July 2002. https://laszewski.github.io/papers/vonLaszewski-gridant.pdf.

[7] Argonne National Laboratory. 2022. Parsl: Parallel Scripting in Python. https://parsl-project.org [Online; accessed 14. Oct. 2022].

[8] Google Cloud. 2022. Quickstart: Create a workflow using the gcloud CLI | Workflows | Google Cloud. https://cloud.google.com/workflows/docs/create-workflow-gcloud [Online; accessed 14. Oct. 2022].

[9] Fábio Dario. 2021. *Enterprise Workflows with Azure Logic Apps and Azure Functions.* gft-engineering. https://medium.com/gft-engineering/enterprise-workflows-with-azure-logic-apps-and-azure-functions-303971a15f10

[10] ecfan. 2022. Workflow Definition Language schema reference - Azure Logic Apps. https://learn.microsoft.com/en-us/azure/logic-apps/logic-apps-workflow-definition-language [Online; accessed 14. Oct. 2022].

[11] GitHub. 2022. Cancel a workflow run - Workflow runs. https://docs.github.com/en/rest/actions/workflow-runs#cancel-a-workflow-run [Online; accessed 14. Oct. 2022].

[12] graphviz 2021. Graphviz. https://graphviz.org [Online; accessed 21. Oct. 2022].

[13] IBM. 2021. Business Automation Workflow REST APIs programming. https://www.ibm.com/docs/en/baw/19.x?topic=apis-business-automation-workflow-rest [Online; accessed 14. Oct. 2022].

[14] IBM. 2022. IBM Documentation. https://www.ibm.com/docs/en/spectrum-lsf [Online; accessed 14. Oct. 2022].

[15] Keplet. 2020. The Kepler Project — Kepler. https://kepler-project.org/index.html [Online; accessed 14. Oct. 2022].

[16] Gregor von Laszewski, Anthony Orlowski, Richard H. Otten, Reilly Markowitz, Sunny Gandhi, Adam Chai, Geoffrey C. Fox, and Wo L. Chang. 2021. Using Cloudmesh GAS for Speedy Generation of Hybrid Multi-Cloud Auto Generated AI Services. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. 144–155. https://doi.org/10.1109/COMPSAC51774.2021.00032

[17] André Merzky, Mark Santcroos, Matteo Turilli, and Shantenu Jha. 2015. RADICAL-Pilot: Scalable Execution of Heterogeneous and Dynamic Workloads on Supercomputers. *CoRR* abs/1512.08194 (2015). arXiv:1512.08194 http://arxiv.org/abs/1512.08194

[18] Microsoft. 2022. Batch service workflow and resources - Azure Batch. https://learn.microsoft.com/en-us/azure/batch/batch-service-workflow-features [Online; accessed 14. Oct. 2022].

[19] mnist-programs 2022. UVA C4GC REU 2022 MNIST Programs. GitHub. https://github.com/cybertraining-dsc/reu2022/tree/main/code/deeplearning/mnist [Online; accessed 24. Oct. 2022].

[20] OASIS. 2022. Web Services Resource Framework (WSRF) TC XML Schema Documentation. Web Page. https://schemas.liquid-

technologies.com/OASIS/WSRF/1.2/

[21] Robert Outlaw. 2016. Workflows - REST API (Azure Logic Apps). https://learn.microsoft.com/en-us/rest/api/logic/workflows [Online; accessed 14. Oct. 2022].

[22] Juri Papay. 2022. CloudMask benchmark notes. GitHub. https://github.com/laszewsk/mlcommons/tree/main/benchmarks/cloudmask/#readme [Online; accessed 24. Oct. 2022].

[23] Pegasus 2022. Pegasus WMS's documentation. https://pegasus.isi.edu/documentation/index.html [Online; accessed 14. Oct. 2022].

[24] SLURM 2022. Slurm Workload Manager - Documentation. https://slurm.schedmd.com/documentation.html [Online; accessed 14. Oct. 2022].

[25] snakemake 2022. Snakemake Documentation. https://snakemake.readthedocs.io/en/stable [Online; accessed 14. Oct. 2022].

[26] Gregor von Laszewski. 1996. *A Parallel Data Assimilation System and Its Implications on a Metacomputing Environment.* Ph. D. Dissertation. Syracuse University, USA. https://laszewski.github.io/papers/laszewskithesis.pdf AAI9737816.

[27] Gregor von Laszewski. 1999. A Loosely Coupled Metacomputer: Cooperating Job Submissions Across Multiple Supercomputing Sites. *Concurrency: Practice and Experience* 11, 15 (December 1999), 933–948. https://doi.org/10.1002/(SICI)1096-9128(19991225)11:15<933::AID-CPE461>3.0.CO;2-J The initial version of this paper was available in 1996.

[28] Gregor von Laszewski. 2019. Cloudmesh Manual. GitHub. https://cloudmesh.github.io/cloudmesh-manual/

[29] Gregor von Laszewski and J.P. Fleischer. 2022. Hybrid Multi-Cloud Analytics Services Framework. GitHub. https://github.com/cloudmesh/cloudmesh-cc [Online; accessed 14. Oct. 2022].

[30] Gregor von Laszewski, Jarek Gawor, Carlos J. Peña, and Ian Foster. 2002. InfoGram: A Peer-to-Peer Information and Job Submission Service. In *Proceedings of the 11th Symposium on High Performance Distributed Computing (HPDC '02)*. IEEE Computer Society and Washington and DC and USA, Edinbrough and U.K., 333–342. https://laszewski.github.io/papers/vonLaszewski-infogram.pdf http://dl.acm.org/citation.cfm?id=822086.823347.

[31] Gregor von Laszewski, Mihael Hategan, and Deepti Kodeboyina. 2007. *Java CoG Kit Workflow.* Springer London, London, 340–356. https://doi.org/10.1007/978-1-84628-757-2_21

[32] Yong Zhao, Hategan, M., Clifford, B., Foster, I., von Laszewski, G., Nefedova, V., Raicu, I., Stef-Praun, T., Wilde, and M. 2007. Swift: Fast, Reliable and Loosely Coupled Parallel Computation. In *2007 IEEE Congress on Services*. 199–206. https://doi.org/10.1109/SERVICES.2007.63

A Appendix

A.1 Installation

To leverage cloudmesh-cc, use the cloudmesh-installer to install the Cloudmesh suite of repositories. Optionally, to utilize the graph visualization you must install also *graphviz* must be installed. On Windows, Git Bash is required, in addition. The overall installation is very simple and is supported on a variety of operating systems. We leverage the cloudmesh-installer to locally install the cloudmesh suite of repositories by executing the following commands:

```
$ mkdir ~/cm
$ cd ~/cm
$ pip install cloudmesh-installer -U
$ cloudmesh-installer get cc
```

To install graphviz you can use the following commands on the appropriate operating system:

Windows. Git Bash and Graphviz must be installed. The user can use an instance of Chocolatey that is run as an administrator for convenience:

```
$ choco install git.install
        --params "/GitAndUnixToolsOnPath
                  /Editor:Nano
                  /PseudoConsoleSupport
                  /NoAutoCrlf" -y
$ choco install graphviz -y
```

macOS. Graphviz must be installed. The user can use Homebrew for convenience:

```
$ brew install graphviz
```

Linux. Graphviz must be installed. The user can use apt for convenience:

```
$ sudo apt install graphviz -y
```

To test the workflow program, prepare a `cm` directory in your home directory by executing the following commands:

```
$ cd ~/cm/cloudmesh-cc
$ pytest -v -x --capture=no tests
```

A variety of separate tests are available that test individual capabilities.

A.2 Running Cloudmask Workflow

To run the Cloudmask workflow, run the following commands:

```
$ cd ~/cm
$ git clone https://github.com/laszewsk/mlcommons.git
$ cd mlcommons
$ pytest -v -x --capture=no
      benchmarks/cloudmask/target/rivanna/run_cloudmask_workflow.py
```

A.3 Contributing

All contributions are done under the Apache License. The code is maintained as an open-source project on GitHub while using the typical GitHub code management tools such as:

- Code Repository
- Issue Management
- Pull Request Management
- Automatic verification with GitHub Actions

The main branch is the release branch and is supposed to be functional at all times. Hence, contributions are first done in other branches, and once agreeing that they need to be integrated into the code, they are merged into main. All new code must be documented and have sufficient automated tests. Before creating a pull request, it is important that the tests within the test directory are passing. The repository already contains several pytests that can be leveraged to conduct routine testing of the code, including its SSH remote functionality, REST capability, and Python interface, among others.