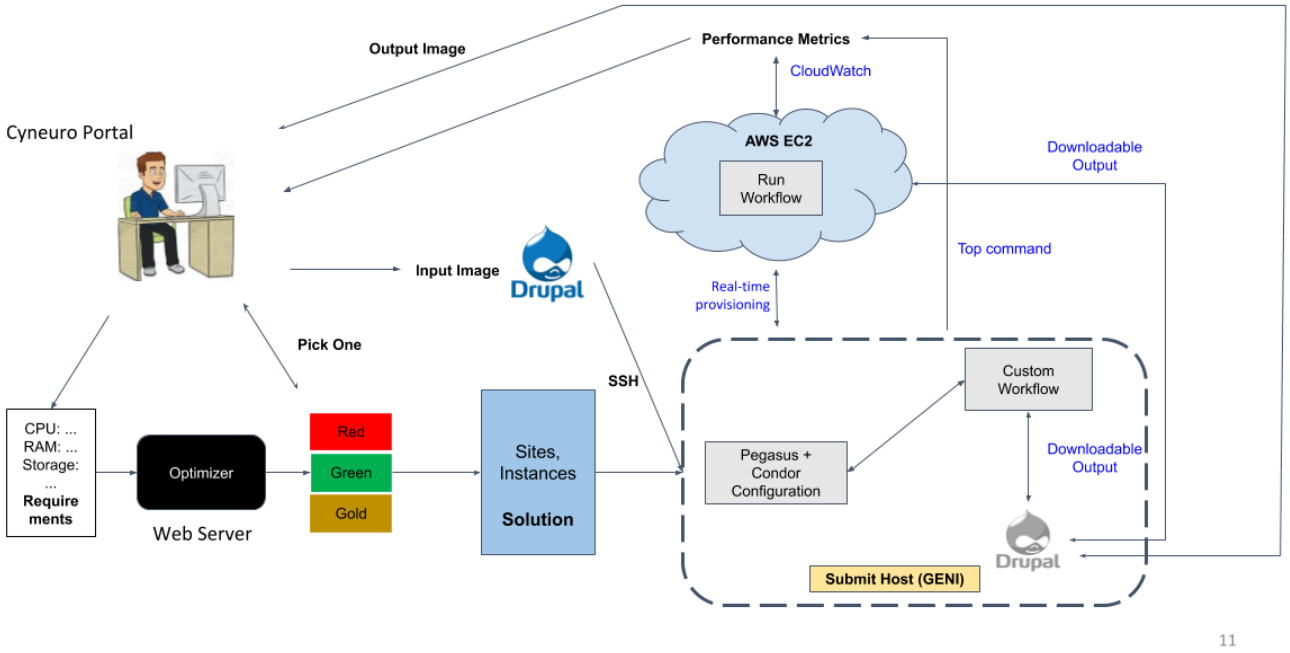
Cloud Custom Template for Non-Experts

CS 4530/7530 Cloud Computing Project Report

Xinzhou Liu, Brad Boutaugh, Ashish Pandey

Introduction : We are trying to create a middleware which will help non-expert cloud users in accessing cloud solutions easily as per their requirements. The user will be offered template solutions as per their specified requirement. The selected cloud template will be deployed real time and the custom user workflow will be initiated on the deployed resources. The user will have the facility to monitor status of his workflow and also monitor deployed cloud resources on central Cyneuro GUI portal or any other integrated custom GUI Portal. We are primarily focusing on researchers who have typically huge datasets for analysis. Our middleware aims at creating cloud solutions based on optimal mix of factors like cost, performance, and agility offered from the CSPs, thus saving them money and time.

Implemented Model :



Control and Data Flow

Components :

1) ***GUI :*** Cyneuro portal being being developed under chatbot project.
2) ***Optimizer :*** We have used CPLEX optimizer to create optimization model. The model is based on user defined constraints. The model's objective is to reduce the cost of the template solution for specific user requirement. We achieve multiple template solutions by varying the user defined resources constraints, under a threshold given by user. The threshold given by user is used to magnify constraints resources, and thus varying constraints fed into optimizer engine gives varying template solutions. These templates are divided into three group of solutions based on the threshold and user preference of certain resources.

Red Solutions: Strict user defined resource constraints are considered. It gives one cost effective optimal solution.

Green Solutions: **All user defined resources** are amplified in step sizes up to threshold limit. Each step gives an amplified resource constraint which gives a solution. The number of steps up to the threshold decides the number of solutions generated in this category.

Gold Solutions: **User preferred resources** are amplified in step sizes up to threshold limit. Each step gives an amplified resource constraint which gives a solution.

We customize the cloud resource information to feed the optimizer in the format given below, we considered +60 instance types to be taken into consideration for optimization. More instances can be added in the below format to be taken into consideration by the optimizer. Optimizer does not differentiate between these json objects and thus gives a solutions which could be mix of multiple cloud providers. We show below a sample having instance of a local machine and a aws instance.


```
[
{
  "csp":"LOCAL",
  "OS":"LINUX",
  "name":"instageni.umkc_1",
  "vCPU":"1",
  "ram":"1.5",
  "price":"0.0155",
  "network":"5",
  "clock":"1.2",
  "pricing_ssd":"0.10",
  "pricing_hdd":"0.045"
},
{
  "csp":"AWS",
  "OS":"LINUX",
  "name":"a1.medium",
  "vCPU":"1",
  "ram":"2",
  "price":"0.0255",
  "network":"10",
  "clock":"2.3",
  "pricing_ssd":"0.10",
  "pricing_hdd":"0.045"
}]
```

Variables solved using optimizer are : instance_x_number,
where instance_x_number is the number of instances required of instance type indexed at x.

Cost Minimization Objective:
instance_1_price*__instance_1_number__ + instance_2_price*__instance_2_number__ + …… + instance_x_price*__instance_x_number__

Sample Constraints:

instance_1_vcpu*__instance_1_number__ + instance_2_vcpu*__instance_2_number__… >= user_vCPU* step_threshold
instance_1_memory*__instance_1_number__ + instance_2_memory*__instance_2_number__ … >= user_vCPU* step_threshold
instance_x_clock > = user_clock where x is instance index
instance_x_network > = user_network where x is instance index
instance_x_number >= 0 where x is instance index

additional constraints can be added similarly,

3) Testbed Configuration :

GENI node was used as the local submit host where all the jobs were managed and scheduled to run either locally or on the cloud server, i.e., AWS EC2. The workflow management system Pegasus and job scheduling system HTCondor were properly configured. In order to schedule jobs, all the worker machines must be visible in the condor pool, which is a set of machines that use HTCondor for resource management. In order to make the cloud resources available in the condor pool, we tried many command line tools such as *condor_annex* and *pyglidein.*

Condor_annex is a tool that provisions the compute resources on cloud servers so that the jobs can run remotely in the cloud. So far, condor_annex only supports AWS EC2. Setting up condor_annex required creating a key pair and opening a dedicated network port on the submit host to securely connect with AWS EC2. After configuration, the *condor_annex* command, with user-defined parameters, can launch a certain number of a specific type of EC2 instance, and then make the instances available in the condor pool for the job scheduler to choose from.

We wanted to add another cloud resource XSEDE to the condor pool, so we tried pyglidein. Pyglidein is a mechanism by which one or more remote machines temporarily join a local condor pool once this service is set up in both submit and client machines. Pyglidein is set up as a server in submit host and as a client in potential cloud resources. Pyglidin client adds the cloud resources into submit machine's resources pool and then the cloud resources can be used to run jobs scheduled in the submit node. We were able to configure Pyglidein on submit host and cloud resources but the configuration is failing and we have not been able to debug it yet. Additionally such setup would not be real time as configurations has to be done beforehand.

Last piece of configuration is for Pegasus, which runs on top of HTCondor. We modified the Pegasus tutorial workflow "pipeline" into a customized workflow by modifying the daxgen.py script. The script defines the abstract workflow, which will be converted into a concrete, executable workflow by Pegasus at the runtime. We also modified the site catalog to include both local and remote (AWS) execution sites, and the replica and transformation catalogs to expose the locations of input/output data and executables.

4) Custom Workflow :

The workflow we tested in our experiment is essentially a simple image processing pipeline that converts a color image into grayscale. It starts with building a virtual environment and installing necessary image processing tools, then runs the Python script that actually does the image processing and saves the output image, and, finally, tears down the environment and cleans up the intermediate files.

The pipeline code shown below takes in 3 parameters: a Python script, an input and an output image files.

```
#!/bin/bash
set -e
virtualenv env2.7
source env2.7/bin/activate
python -m pip install --upgrade pip
pip install Pillow

python "$@" # rgb2grey.py imgIn imgOut

deactivate
rm -rf env2.7
```

The image processing script rgb2grey.py used in the pipeline is shown below:

```
import sys
from PIL import Image
# The name of the DAX file is the first argument
if len(sys.argv) != 3:
        sys.stderr.write("Usage: %s rgb2grey, need 1 image file and 1 output file\n" %
(sys.argv[0]))
        sys.exit(1)
imgIn = sys.argv[1]
imgOut = sys.argv[2]

img = Image.open(imgIn).convert('LA')
img.save(imgOut)
```

The pipeline was added as an executable in the abstract workflow generator script (daxgen.py):

```
# daxgen.py
# Add executables to the DAX-level replica catalog
img_process = Executable(namespace="dax", name="pipeline.sh", installed=False,
version="4.0", os="linux", arch="x86_64")
img_process.metadata("size", "2048")
img_process.addPFN(PFN("file://{}/code/pipeline.sh".format(wd),"local"))
dax.addExecutable(img_process)
```

We designed this workflow because it is highly customizable and easy to scale in the future. Large Image processing can be computationally and economically expensive, so optimized compute resource allocation and

management can significantly reduce the cost and improve the efficiency. With the Pegasus/HTCondor system, the workflow can be run on either local or remote or mixed resources.

5) *Performance monitoring:*

The performance metrics on the submit host (local machine) includes peak CPU and memory usages. All user needs to do is run the metrics.sh. And here is how it works. To measure the performance, we first fetched the outputs of the *top* command and saved it to *top.log*, as shown below.

```
top -u $USER -b -n 1 | sed -n '8,$p' > top.log
python metrics.py
```

In the metrics.py, the content of top.log was read to calculate the current total CPU and memory usages by summing the usages of each command.

```
cpu = 0
mem = 0
try:
      with open('./top.log','r') as file:
      for line in file:
            data = line.strip().split()
            cpu += float(data[8])
            mem += float(data[9])
except FileNotFoundError:
      print 'top.log not found'
```

Then, the current usages were compared to the max usages recorded in the *max_metrics.log* to update max usages.

```
exists = os.path.isfile('./max_metrics.log')
if exists:
      with open('./max_metrics.log','r') as file:
      lines = file.readlines()
      max_cpu = float(lines[0].strip().split()[1])
      max_mem = float(lines[1].strip().split()[1])
max_cpu = max(max_cpu, cpu)
max_mem = max(max_mem, mem)
```

Finally, the peak performance metrics was saved in the max_metrics.log and printed out to the user. For example,

```
max_cpu: 33.0%
max_mem: 40.3%
```

6) **Integrating Shell Scripts :**

The run.sh executable was ssh-called from the web server with 2 parameters. The shell script contains the following steps:

a) Configure the personal HTCondor environment

b) Read the parameters. The $1 parameter represents the execution SITES extracted from the user-selected template. There are 3 options: local, aws, and mix (contains both local and aws resources). The $2 parameter, an array, represents the EC2 INSTANCES on which the resources are provisioned.

The following code show how to launch all the instances in the array. Each item in the array is a instance-type:instance-count pair. Condor_annex is used on every type of instance in the array except the local.

```
INSTANCES=$2
IFS=',' read -r -a array <<< "$INSTANCES"
```

```
for index in "${!array[@]}"
do
    element=${array[index]}
    IFS=':' read -r -a kv <<< "$element"
    echo "type:${kv[0]}, count:${kv[1]}"
    instance=${kv[0]}
    count=${kv[1]}
    if [ $instance != 'local' ]
    then
        # start ec2 instances
        yes | condor_annex -count $count -annex-name "Annex$index" -aws-on-
demand-instance-type $instance
    fi
done
```

c) Before running the workflow, we need to clean up previous workflow record. Code is shown below:

```
# go to workflow folder
cd ~/pipeline_mix/
# clean up previous workflow files
echo "remove previous workflows..."
#pegasus-remove submit/xl6v8/pegasus/pipeline/run*
rm -rf submit/xl6v8/pegasus/pipeline/run*
```

d) Run the workflow with the .dax file and execution sites

```
./plan_dax.sh pipeline.dax $SITES
```

Conclusion:

- Created an optimizer engine, being exposed via web services which can be integrated to any custom GUI.
- Able to configure workflow management softwares which enables user to access workflows easily.
- Auto and real time time deployment of cloud solution based on user defined criteria.
- Created custom workflow for image analysis.
- Monitoring service for workflow and deployed cloud resources.
- **User can define their resource requirement, select workflow input, deploy cloud template as per their choice, monitor workflow status and monitor cloud resources performance metrics from central Cyneuro GUI portal.**

Possible future work/ Discussion:

- Enhancement of optimizer engine.
- More resource intensive, relevant and complex workflows to be created for testbed creation.
- Enabling more cloud services such as XSEDE, OSG, Local MU resource which can done using pyglidein.