HEP Cloud investigation of Google Compute Engine

2016 Summer internship final report

Flavio Giobergia

Supervisors
Gabriele Garzoglio
Steven Timm

Contents

Chapter 0

Introduction

This Summer internship has been spent exploring Google Compute Engine and its possible integration into HEP Cloud (a Scientific Computing Division project). Since no previous work had been done for the Google integration, the internship had to start from square one (although the previous Amazon Web Services experience helped setting expectations and milestones): this implies that the first part of the internship consisted in getting to know Google's offering and figuring out if and how that would suit HEP Cloud needs: while the reader may find it trivial, it actually required going through Google's documentation, understand potentials and limits of various options and make choices based on constraints and requirements. Because of that, that part has not been omitted and constitutes the first part (after the HEP Cloud introduction) of this report.

The nature of this document dictates that it should mainly focus on the individual work carried out throughout the internship. Apart from the introductory chapter, the rest of the report will only contain such work. This does not in any way mean that the Google investigation has been a one-man work: on the contrary, many people spent their time on the project. When relevant, this report will mention some of other people's work and give the credit due.

Chapter 1

HEP Cloud

1.1 The problem

The High Energy Physics program is evolving and that comes with an ever increasing appetite for computing. The High Luminosity LHC at CERN, the muon and the neutrino programs all require high precision and handle highly complex events: this requires substantial computing power. In the years to come, the computing needs are expected to grow by – at the very least – a factor of 10. This requires scaling to accommodate for that demand.

The demand for computing resources is not evenly distributed throughout the year: some periods may be more computationally intensive (when simulations, data processing and analysis happen), while others are relatively calm. The traditional approach to computing resources provisioning (i.e. buying enough machines based on the worst case) is a bad fit for this fluctuating scenario: the most intensive periods are well handled but, during the rest of the time, only a fraction of the available computing power may be actually used. This inflexible solution thus may result in a waste of money and resources.

Alternative solutions, such as that offered by Open Science Grid [1], target this very problem, exploiting unused resources through the opportunistic usage of idle cycles. As a federation of resources owned by scientific institutions, OSG has a limited capability to address the ever increasing need for bursts of resource capacity. The HEP Cloud Facility project aims at integrating commercial Cloud resources to support such bursts in computing demand, paying for resources only when the need arises.

1.2 The solution

The HEP Cloud project was launched in June 2015 by the Scientific Computing Division at Fermilab. It aims to provide a common interface to access a heterogeneous set of computing resources, including local clusters, grids, high-performance computers, community and commercial clouds. This last resource is the most suitable candidate to target the aforementioned problem. Rather than buying resources, cloud computing services offer a pay-as-you-go approach: clients rent virtual machines and pay by the hour, with none of the additional costs that would come with buying and maintaining the underlying hardware.

HEP Cloud already supports Amazon Web Services, one of the largest cloud computing services currently available [2]. In early 2016, HEP Cloud successfully provided an extra 58,000 AWS cores to the CMS experiment [3], four times the computing power available at Fermilab. In order to further expand HEP Cloud, support for other cloud services is required: the rest of this report will detail the work that has been carried out during this Summer internship for supporting Google Cloud Platform on HEP Cloud.

1.3 Google Cloud Platform

Google Cloud Platform is Google's response to AWS. It offers a cloud computing platform with a large number of services, including Google Compute Engine, the Infrastructure as a Service (IaaS) component.

While the decision of supporting from scratch a new service may seem counterintuitive, Fermilab decided it would be best not to rely on a single service provider: having a second option allows for comparisons of prices and performance, while also providing a redundant service in case things go awry with the first one. The steps taken during the internship to support GCP are:

- launching a Scientific Linux VM on Google servers
- configuring and tuning the VM based on HEP Cloud needs
- using Google API for a scripted interaction with the machines
- implementing a monitoring system to supervise the cloud's overall health
- implementing a billing monitor to keep track of the expenses charged by Google
- launching benchmark tests to assess the potential of Google's machines
- doing a case study on the Mu2e experiment to evaluate if and how it can benefit from GCP

Chapter 2

Investigating Google Compute Engine

2.1 Running Scientific Linux on GCE

HEP Cloud makes use of HTCondor, a specialized workload management system for compute-intensive job [4]. HTCondor collects jobs from users (in the form of submission files) and dispatches said jobs to a machine within a pool of computers available. This pool can be comprised of different kinds of machines and HEP Cloud leverages this feature to handle various resources.

HTCondor offers some support to Google Compute Engine: this allows launching virtual machines (that may then proceed to carry out the user's job) along with limited customization (only machine type, zone, image and metadata are configurable [5]). While this allows for some basic testing, more complex operations could not be performed through HTCondor.

After having expressed these concerns, the HTCondor team, in collaboration with Google, started working on expanding the support offered for GCE. The update was made available around the end of the internship: Steven Timm managed to use the known procedures (some from this Summer internship, some from AWS) to successfully launch new jobs using the existing infrastructure.

In order to launch a VM, an "image" is required. This image is a file containing a representation of a bootable hard disk. Different formats are available for representing images: Google currently accepts raw images (i.e. byte-by-byte copies of the actual hard disk contents). The raw image needs to be named `disk.raw`, placed in a tar.gz archive and uploaded to Google Storage. Then, either the browser console, the command-line interface or the API allow for the creation of a new Google image, which may be used at any given time for

dispatching a new VM.

Google already provides a series of vanilla images. While these images are perfect for early testing, HEP Cloud requires images with some software on them (e.g. HTCondor, GlideinWMS [6]) to successfully control dispatched instances. Moreover, users expect to find specific software (required for simulations, data analysis and other CPU intensive activities) already running on the machine: the perfect candidate is Scientific Linux [7], a distro designed with those needs in mind. Unfortunately, Scientific Linux is not among the list of images provided by Google.

This would normally imply having to create the new image from scratch with the desired OS and software. Luckily, the Amazon image used for HEP Cloud already provided most of the required features. Hyun Woo Kim changed some provider-specific settings (e.g. grub configuration) and, after installing some Google tools [8] the existing image was working on GCP, thus cutting down on the time required to have a working HEP Cloud-ready machine

2.2 "Basic plumbing"

After the first machine was up and running, a series of concerns arose. First, the machine needed to be accessible and this access needed to be limited and secured. Second, the machine needed to receive information on how it should behave (this information may yes be hardcoded in the image, but that would prevent on-the-fly changes and it would result in all the machines sharing the same configuration, which may not be the desired behaviour).

2.2.1 Metadata

The latter point is handled using metadata, a method offered by Google (and other cloud services as well) to submit key-value pairs to an instance upon creation (or at later times). These strings may encode anything from specific configuration files to scripts to be executed when particular events happen.

HTCondor supports metadata: these can be sent using either the `gce_metadata` or the `gce_metadata_file` commands (the difference between the two is in the source of the metadata – the second option reads the metadata from a file). Metadata may then be retrieved from within the virtual machine by sending HTTP requests to `metadata.google.internal` (using `/computeMetadata/v1/` as base path).

2.2.2 SSH

As for the first problem, SSH (Secure SHell) is probably the most common tool for accessing a machine remotely. Google makes no exception and uses SSH as the preferred tool for interacting with instances. SSH uses public-key cryptography to authenticate the remote computer and allow it to authenticate the user, if necessary. Users may generate a public/private pair of keys and provide the server with the public key, while keeping the private one for themselves. The server will then use the public key received to test whether the client requesting access actually owns the right private key. Whoever knows the private key, thus, may gain SSH access to the server.

Google already provides useful solutions for handling SSH access to the various instances. `gcloud` is a CLI tool that provides full support to Google Cloud services. This tool also takes care of generating SSH keys on behalf of the user (using tools like `ssh-keygen`), thus making the entire process completely transparent. While this may be good most of the times, having full control over the SSH keys used is required.

Because of that, Google offers options for semi-direct handling of SSH keys: specific metadata (`sshKeys` and `ssh-keys`) are used by Google to "inject" public keys in a given machine (public keys are stored in the `/home/user/.ssh/authorized_keys` file). Project-wide, as well as instance-bound, keys may be used (the only difference between the two being the scope of the keys).

2.2.3 Firewall rules

Only exposing the minimum required to the outside world is a good rule of thumb for security. Google enforces that rule by configuring firewalls so that all incoming traffic is blocked by default and then uses "allow" rules for ports 22 (SSH) and 3389 (RDP). This usually works fine, but some scenarios may require those ports to be filtered and other ports to be allowed instead. Doing that is extremely easy, as the Developers Console has options for creating and deleting "allow" rules that can either target specific IP ranges, or the entirety of the machines. Since the change of these rules normally happens slowly, the web interface has been deemed a good enough tool for this operation.

2.3 Programmatical handling of instances

Google offers a number of ways for interacting with instances. The most user-friendly one is the Developers Console available from any browser. This console provides easy-to-use interfaces for humans. A slightly less user-friendly tool is `gcloud`: it provides no graphical UI and may be used as a normal command-line program. It is great for retrieving information quickly, without having to go through the frills that come with the browser console. `gcloud` may be

also be used for bash scripting. The best way to programmatically work with instances (i.e. creating scripts that interact with machines) is using the API Google offers.

Google offers RESTful API (i.e. API that use HTTP and its features) for interfacing with its Cloud services. Furthermore, it provides libraries for some common languages (including Python). That, along with the provided documentation, is enough to start writing code that interacts with GCE instances. Being able to do that is of vital importance: as other sections of this report point out, the entire cloud needs to be monitored at all times and automating that process (through the use of Google API) is the way to go. While particularly easy to use, the API needs to be secure and that implies having an authentication and an authorization part: the latter is achieved using OAuth 2.0.

2.3.1 OAuth 2.0 and Google Accounts

OAuth 2.0 is an authorization framework that enables third-party applications to obtain limited access to a service on behalf of a user of the service that is being accessed [9]. As such, this method makes it so that third-party applications need not ask users for their username and password (which should always be private) to access the contents protected with said credentials.

The way OAuth 2.0 works is straightforward: the third-party application (or client) has the user authenticate on the server of interest. In return, the server provides a pair of tokens: an access token and a refresh token. These tokens are then used by the client to make requests to the server, which can in turn enforce any kind of policy desired by the user (e.g. limiting the scope of the access). Access tokens are short-lived: after expiration, the client can use the (long-lasting) refresh token to request further access tokens with no need for the user's permission. Figure 2.1 illustrates how the refreshing process happens, highlighting how the refresh token is only ever used with the authorization server, while the access token is sent to the resource server (although, in many cases, the two servers are one).

This method is used for the so called User Accounts, In other cases, a Service Account may also be available. Instead of receiving a pair of tokens, the service account only gets a private key, that can be used to request access tokens.

While at a first glance the two methods may seem similar, there are actually many differences between the two. The most important one is that user accounts are normally used when the client needs to access some of the user's information stored on the server and thus requires the owner's authorization. On the other hand, many applications do not need access to any user information and instead only need to use some of the server's services. In this case, it would not make sense to ask a user for their authorization: this is why service account were introduced.

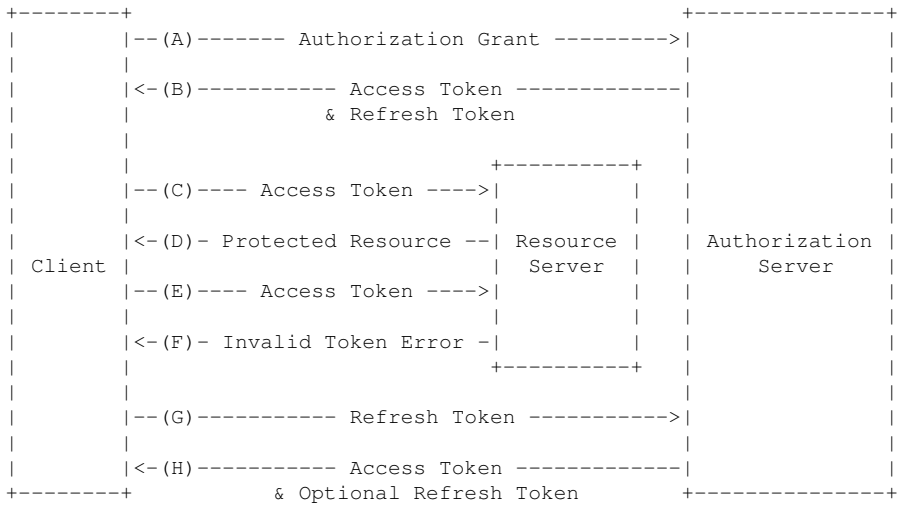

Figure 2.1: Refreshing an Expired Access Token (from RFC 6749)

The HEP Cloud scenario does not require any interaction with a user's data (as it works as a central resource manager): the service account option is therefore ideal. All code written therefore uses this type of account. HTCondor used to only support user accounts: the new update supports service accounts as well (as a temporary workaround, a user account was being used as a makeshift service account).

2.4 Monitoring

Monitoring the instances' status is of paramount importance, as it provides information about the health of the cloud at a glance, highlights any problem or unexpected behaviour and shows useful statistics on the usage of the various machines. While countless metrics may be measured (Google offers more than 900 [10]), only a couple have been deemed useful for this initial stage (along with those already collected through HTCondor): the number of running instances and the CPU utilization for such instances. This information should be aggregated by project, geographical zone and machine type.

2.4.1 The existing platform

Fermilab (through Fifemon [11]) is already doing extensive monitoring activities on the existing infrastructures: the monitoring code will therefore need to interface with it. Fifemon uses Graphite for storing numeric time-series data [12] and Grafana for querying and visualizing time series and metrics [13].

The way Graphite works is incredibly straightforward. A server listens on a given port and it accepts 3-tuples following the format `<metric path> <metric value> <metric timestamp>` (data may also be encoded using the pickle protocol [14]: this is actually what Fifemon uses). The `metric path` field allows for dot-separated hierarchies (e.g. `foo.bar.baz.qux`) and proves useful when grouping data.

In order to send data to Graphite, Fifemon already provides some classes for implementing new probes.[15]. The `Graphite` class handles the interactions with the Graphite server and exposes the `send_dict` method, whose sole purpose is that of sending data (stored as a dictionary) to Graphite. The other useful class is `Probe`. This class exposes the method `run` that loops indefinitely and collects the data (through the `post` method, that should be overridden by derived classes) to be sent.

2.4.2 The instance counting probe

The first probe that was developed serves as an instance counter. It queries Google's servers to get a list of running machines. The provided API allows for the retrieval of all instances for a given project and a given geographical zone. Since no selector on the machine type is offered, that part is handled by the probe itself: it scans the list of machines returned, filters by running instances and separates by machine type. The final metric path uses the following structure: `namespace.project.zone.machine_type.count`, where `namespace` is a uniquely identified path (e.g. `hepcloud.test.gce`); the rest is self-explanatory. The core parts of the probe can be found in Listing 2.

2.4.3 The CPU utilization probe

Knowing the average CPU utilization helps knowing how loaded the cloud is at any given time. Retrieving the CPU utilization is a little trickier than counting the number of instances, for reasons that will soon be explained.

Stackdriver is the monitoring API offered by Google. It provides easy access to a number of different metrics collected on the various instances. There are two categories of metrics:

- Some metrics can be collected without access to the instance itself. These are usually less accurate (as they are based on the information provided by the external process that is running the virtual machine), but use no extra resources (CPU, memory, data transfer), since – as mentioned – it is not the instance's duty to report the collected data; and that data would be collected and stored anyway.

– Other metrics can only be collected from within the instance. This requires special software (called "agent") to be installed and running on every VM. This provides information that may not be accessible from the outside (as the VM is seen as a black box that uses resources) and those measurements that could be taken from the outside are usually less reliable (e.g. when measuring the amount of memory used, an inside agent would know exactly how much memory every process is using, while, from the outside, the only data that can be collected is the amount of memory in use by the process running the VM, which includes some overhead memory used by the process and not available to the VM). The big drawback of this solution is that every agent needs to report the collected information, with the waste of resources that comes with it.

The former category is the best one when the number of machines scales up: it lowers the overall consumption of resources and the averaging of inaccurate measurements produces a value that is somewhat accurate. The only problem with using the external process' information is that it may sometimes register a CPU utilization beyond 100%. This is likely due to the way that percentages are computed: $\frac{CPU_{In\,use}}{CPU_{Max\,expected}}$[1]. $CPU_{Max\,expected}$ can only be an estimate of how much CPU the external process is expected to use, based on the type of machine that is being run. However, the external process may end up using – for a short burst – more than the expected amount of CPU, thus resulting in an apparent CPU usage over 100%. If known, this problem does not cause any particular annoyance and it is likely that such behaviours pass unnoticed after averaging out.

Listing 1 is a JSON response to a request for CPU utilization by project. `timeSeries` is an array of objects, each one identifying a metric for a specific instance. The `points` array contains a list of measurements for the specific metric-instance pair: `startTime` and `endTime` are the same because the metric kind is `GAUGE`. Other metric kinds (e.g. `DELTA`) have different start and end times. `value.doubleValue` is the actual reading: in this case, 0.10% of the CPU was in use, as the machine was idling (to be more accurate, the type of request sent required the information of the last 5 minutes to be aggregated; the 0.10% value represents the average CPU utilization over the previous 5-minutes period).

The previously mentioned requirement stating that all data collected needs to be grouped by project, zone and machine type cannot be enforced easily in this scenario. The Stackdriver API is not providing any information regarding the machine type (although the machine type is one and one only, for any given instance), but it returns information about the geographical zone (`resource.labels.zone`): this is a design choice made by the author of the API. For the time being, the probe uses a placeholder for the machine type ("`n1-standard-1`"). Other solutions for retrieving the actual machine type

[1] This is a guess of what actually happens, as Google does not provide information on any of the formulas used.

are $O(n)$ in either the number of API requests or in memory: since this would not scale well, such solutions have been discarded.


```
{
  "timeSeries": [
    {
      "metricKind": "GAUGE",
      "metric": {
        "labels": {
          "instance_name": "cpu-test-instance"
        },
        "type": "compute.googleapis.com\/instance\/cpu\/utilization"
      },
      "points": [
        {
          "interval": {
            "endTime": "2016-09-18T15:57:20.946345Z",
            "startTime": "2016-09-18T15:57:20.946345Z"
          },
          "value": {
            "doubleValue": 0.10114381041274
          }
        }
      ],
      "resource": {
        "labels": {
          "instance_id": "8997168970812556963",
          "project_id": "fermilab-poc",
          "zone": "us-central1-b"
        },
        "type": "gce_instance"
      },
      "valueType": "DOUBLE"
    }
  ]
}
```


Listing 1: Stackdriver response to CPU utilization request

A request for adding the machine type information to the response has been opened and hopefully this key parameter will be provided in a future version of the API. If that piece of information becomes available, the instances counting probe would be rendered useless, as the CPU utilization probe could be used to also collect the number of instances. Listing 3 contains most of the code for the CPU utilization probe.

Figure 2.2 shows what the GCE part of the dashboard currently looks like on

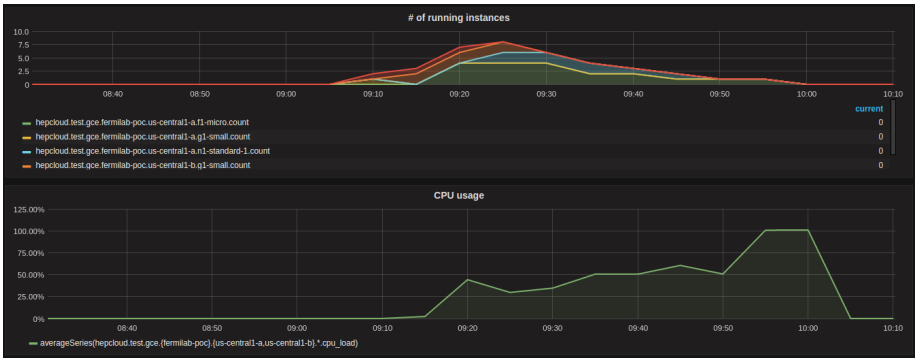

Figure 2.2: Google Compute Engine dashboard on Grafana

Grafana.

2.5 Billing

There are many reasons that could lead to a bitter surprise in the billing: anomalous behaviours, unexpected expenditures and malicious users are just a few examples. The sooner these expenses are noticed, the sooner measures can be taken to limit the damage: this is why monitoring the invoices is so important.

Google provides a daily invoice as a CSV file in Google Storage. Getting an update every 24 hours is not ideal (Amazon provides the same information 4 times as often) and the possibility of getting more frequent updates is currently being explored.

Gabriele Garzoglio wrote the bulk of the code on top of the existing billing probe for AWS, with smaller contributions made by the author. The code acts as follows:

- the CSV files containing the invoices are fetched from Google Storage
- the files are filtered so as to only consider the items within the specified time period
- each item's total cost is computed by summing the item's expenses throughout the time period
- the result of the previous point is aggregated into broader categories
- the final result is sent to Graphite and displayed using Grafana

Due to the sensible nature of the data handled by this probe, the repository containing the source code is being kept private: in accordance to that decision, none of that code will be included in this report.

2.6 Benchmarking

Benchmarking is used to get a quantitative measure of how well a machine performs certain tasks. Different benchmark suites perform different operations, testing the machine for different purposes: testing a high performance computer and a DBMS with the same test – when the two work differently – would not make sense.

The benchmark tests used on Google Cloud machines are *hepspec06* and *ttbar_cms*. These tests are the same as the ones previously used on Amazon machines: this is the perfect way to assess how Google machines compare to AWS.

Google offers numerous default configurations (called "machine types") characterized by different number of cores and amount of main memory available. Moreover, custom configurations may be used – if none of the default ones are to the user's taste.

For benchmarking, the `n1-standard` family has been used. The family includes machines with 1, 2, 4, 8, 16 and 32 cores and 3.75 GB of memory per core (e.g. 4 cores $\rightarrow$ 15 GB). All the machine types but the 32 cores one have been tested using both suites. Table 2.1 lists the results for GCE, while Table 2.2 shows the ones for AWS (available as the result of the work of 2015 Summer student Davide Grassano [16]) .

Machine type	# cores	Speed (GHz)	Price ($/h)
n1-standard-1	1	2.30	0.038
n1-standard-2	2	2.30	0.076
n1-standard-4	4	2.30	0.152
n1-standard-8	8	2.30	0.304
n1-standard-16	16	2.30	0.608

Machine type	ttbar/s per core	ttbar/s	ttbar/s per $/h
n1-standard-1	0.03048	0.03048	0.8020
n1-standard-2	0.02024	0.04048	0.5326
n1-standard-4	0.01973	0.07892	0.5192
n1-standard-8	0.02075	0.16598	0.5460
n1-standard-16	0.01947	0.31157	0.5125

Machine type	HS06 per core	HS06	HS06 per $/h
n1-standard-1	23.67	23.67	623
n1-standard-2	16.92	33.85	445
n1-standard-4	18.50	74.01	487
n1-standard-8	13.95	111.63	367
n1-standard-16	12.56	200.97	331

Table 2.1: GCE benchmarking results (ttbar and hepspec06)

Machine type	# cores	Speed (GHz)	Price ($/h)
m3.xlarge	4	2.50	0.266
m3.2xlage	8	2.50	0.532
m4.xlarge	4	2.40	0.252
m4.2xlage	8	2.40	0.504
m4.4xlarge	16	2.40	1.008
c3.xlarge	4	2.80	0.210
c3.2xlage	8	2.80	0.420
c3.4xlarge	16	2.80	0.840
c4.xlarge	4	2.90	0.220
c4.2xlage	8	2.90	0.441
c4.4xlarge	16	2.90	0.882
r3.xlarge	4	2.50	0.350
r3.2xlarge	8	2.50	0.700
r3.4xlarge	16	2.50	1.400
cc2.8xlarge	32	2.60	1.090

Machine type	ttbar/s per core	ttbar/s	ttbar/s per $/h
m3.xlarge	0.0139	0.0557	0.209
m3.2xlage	0.0139	0.111	0.208
m4.xlarge	0.0201	0.0806	0.320
m4.2xlage	0.0191	0.153	0.304
m4.4xlarge	0.0198	0.317	0.315
c3.xlarge	0.0153	0.0611	0.291
c3.2xlage	0.0153	0.122	0.291
c3.4xlarge	0.0149	0.239	0.284
c4.xlarge	0.0228	0.091	0.415
c4.2xlage	0.0226	0.181	0.410
c4.4xlarge	0.0205	0.327	0.371
r3.xlarge	0.0151	0.060	0.172
r3.2xlarge	0.0150	0.120	0.171
r3.4xlarge	0.0146	0.233	0.166
cc2.8xlarge	0.0141	0.450	0.413

Machine type	HS06 per core	HS06	HS06 per $/h
m3.xlarge	14.3	57.1	215
m3.2xlage	12.2	97.6	184
m4.xlarge	16.1	64.5	256
m4.2xlage	15.1	121	240
m4.4xlarge	13.5	217	215
c3.xlarge	14.9	59.4	283
c3.2xlage	14.7	118	281
c3.4xlarge	13.2	212	252
c4.xlarge	17.5	69.9	318
c4.2xlage	16.5	132	300
c4.4xlarge	14.8	237	268
r3.xlarge	15.5	62	177
r3.2xlarge	14.2	114	162
r3.4xlarge	12.7	203	145
cc2.8xlarge	11.2	359	329

Table 2.2: AWS benchmarking results (ttbar and hepspec06) [16]

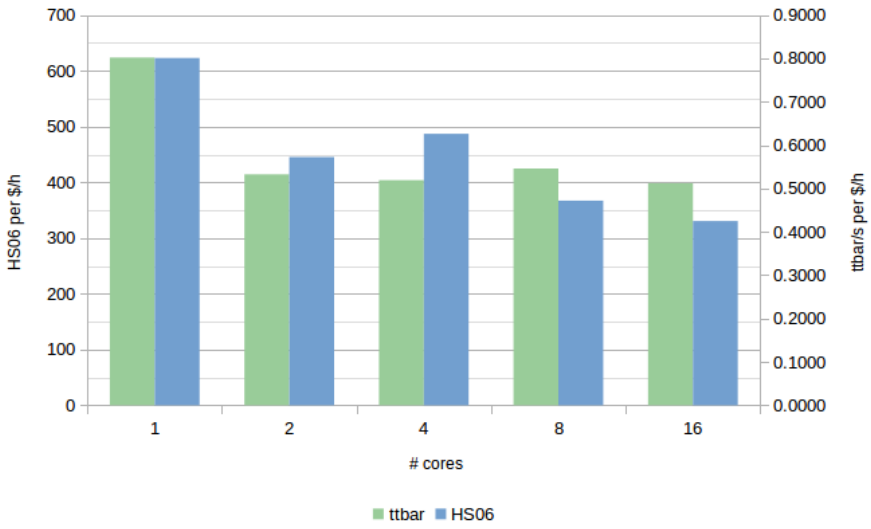

Figure 2.3: Value for money by machine type

While some of Amazon machines get better scores in absolute terms, Google's machines seem to be an overall better deal, because of the lower hourly cost. It is worth keeping into account that Amazon's results are out-of-date: many of the instances listed in Table 2.2 belong to a previous generation [17]. Benchmarking tests should be run on Amazon's new machines to get an up-to-date result: this, though, is out of this internship's scope and would result in a never-ending chase for more recent results.

Of particular interest is the way different Google machines compare. Figure 2.3 sums up the various machines' value for money. The single core option ranks significantly better than the other ones. Finding the precise reason for this behaviour would require further investigation – as well as access to GCE's internal details. This behaviour, though, could be expected since the amount of work done usually increases sublinearly with the number of cores (many reasons lead to this result: scarce parallelism in the code, policies enforced by the OS and processor architecture). A similar behaviour had already been observed in FermiCloud, which uses the same OS and the same hypervisor (KVM).

Chapter 3

Mu2e case study

3.1 Cosmic Ray Veto

The Mu2e experiment [18] has been chosen as alpha user for the Google Compute Engine integration. In particular, Google's machine will be working on the simulations required for verifying the updated design of the Cosmic Ray Veto system. Cosmic rays can interact with the detector, possibly leading to a fake conversion signal: in order to achieve the experiment's designed sensitivity, the CRV needs to account for such events accurately.

3.2 The workload

The simulation requires the execution of 500,000 to 2 million jobs. Each job is expected to take about 4 hours to complete (based on previous experience with Open Science Grid) and produces a 100÷200 MB output. Each job requires 1 core, 2 GB of memory and 9 GB of storage.

Two different scenarios have been outlined for the retrieval of the jobs' outputs:

- The first option requires all outputs to be transferred immediately from Google to Fermilab. This option does not require to temporarily store data anywhere else, but implies having the machine "waste" time sending data to Fermilab, using a connection that can be assumed to be slow (or, at least, slower than the intra-Google one).
- The second option stores all outputs to Google Storage (resulting in a quicker data transfer). All the data is then pulled from Google Storage periodically. This introduces a cost for using Google Storage, but

eliminates the cost of idle VMs waiting on output transfer completion, competing for bandwidth.

Since significant measurements of the links speeds are not available, an approximation of sorts has been used. Since only 5 jobs per day are expected to be executed by every core (and each job takes approximately 4 hours to run), the price estimate accounts for an extra 4 hours every day ($24 - 5 \cdot 4$). This period is orders of magnitude greater than the time it would take to transfer the outputs to either destination (Google or Fermilab) assuming link speeds in the tens or hundreds of Mbps. This period may therefore be used to account for all the supporting operations that need to be carried out (download the parts needed, upload the output and so on). This cushion time makes it so that no tangible difference exists between the two aforementioned scenarios (apart from the Google Storage usage for the latter case). During the actual execution, the supporting operations are likely to last much less than the expected 4 hours: each VM can move on to the next job as soon as it is done with the previous one, shrinking the total time and the total cost estimated in this study.

3.3 The results

Table 3.1 contains the final estimate of the cost for this simulation. Appendix B contains a spreadsheet covering the main costs that come with running the described jobs (those costs sum up to the final result shown in Table 3.1). While the spreadsheet in the appendix only shows static data, the actual file allows tweaking various parameters to adapt to different situations.

Interestingly, the total cost does not change with the kind of machine used. This is because of the underlying assumption that the work done scales linearly with the number of cores (and the price). The benchmarks show that this is not exactly the case: a more complex model could account for this.

Second, the total number of cores provided does not influence the total cost, but only the total duration of the simulation (as the spreadsheet shows, the total number of cores assumed to be used is 150,000: this is approximately the

	500,000 jobs		2,000,000 jobs	
Machine type	1st scenario	2nd scenario	1st scenario	2nd scenario
custom	88,107	88,172	352,430	352,820
n1-standard-1	98,259	98,324	393,038	393,428
n1-standard-2	98,259	98,324	393,038	393,428
n1-standard-4	98,259	98,324	393,038	393,428
n1-standard-8	98,259	98,324	393,038	393,428
n1-standard-16	98,259	98,324	393,038	393,428

Table 3.1: Total cost estimate (all results are in USD)

number of cores expected for this simulation). Increasing the number of cores, though, may not be the best of strategies: if that happens, the storage and the link between Fermilab and Google could become bottlenecks, with all the implications of the case.

3.4 Discounts

The prices mentioned in the spreadsheet are the ones applied in most cases. Google – though – offers discounted prices in some situations. Keeping these into account may result in a substantially lower invoice.

3.4.1 Sustained use

If an instance is run for a significant portion of a billing month, Google will refund part of the money spent. The discount applied may get as high as 30% if the usage level for the month is 100%. This kind of offer is particularly suited for heavy users. Fermilab could be labelled as such, if a large portion of the computations were to migrate to the cloud.

3.4.2 Preemptible instances

An even higher discount comes with preemptible instances. Google's machines run at all times, independently of the effective load. Because of that, it is convenient for Google to sell the unused computing power at a fraction of the price: machines using those resources are called preemptible. Preemptible instances cost roughly 28% of their "normal" counterparts, but have a major drawback: while still executing a workload, the VM may be preempted i.e. shut down to get a hold of its resources for customers paying the full price. Machines receive a "warning" signal 30 seconds before being shut down. The user may specify – upon creation of the instance – a script to be executed in such a situation. If possible, the script should create a checkpoint of the current progress done and prepare to shut down gracefully.

Google makes some guarantee on the minimum time a preemptible instance will stay up: if the VM is shut down before 10 minutes, no charge will incur. Google also enforces an upper bound on the execution time a preemptible instance will get: if the virtual machine has been running for more than 24 hours straight, it will be preempted – no matter what.

Chapter 4

Conclusions

The HEP Cloud facility is a portal to an ecosystem of diverse computing resources, commercial or academic, including the Grid, High Performance Computers, and Clouds. It provides a complete solution to users for managing scientific workflows. The Facility routes workloads to local or remote resources based on workflow requirements, cost, and efficiency of accessing various resources.

This Summer internship has been spent investigating the Google Compute Engine option for HEP Cloud: this entailed getting a solid understanding of the way GCE works and the way it can be integrated; followed by the implementation of tools that will be used to monitor and get useful information out of Google's cloud. Then, GCE has been compared to AWS using different metrics. Finally, Mu2e has been considered as a potential alpha user and an estimate based on the experiment's needs and Google's offering has been provided.

Google Compute Engine proved to be a viable alternative to Amazon Web Services. The recent HTCondor update allows for full control of GCE instances and introduces service accounts support. While some problems with Google API have come up (lack of machine type in Stackdriver responses and low update frequency for billing), none of them is so severe as to prevent the integration and can be easily overcome (although future updates may result in an improved service). Given the benchmark results and Google discounts, it would appear that Mu2e could benefit from an integration with GCE.

References

[1] Open Science Grid project, https://www.opensciencegrid.org/about/

[2] f http://fortune.com/2015/05/19/amazon-tops-in-cloud/, May 2015

[3] Jeff Bar, *Experiment that Discovered the Higgs Boson Uses AWS to Probe Nature*, https://aws.amazon.com/blogs/aws/experiment-that-discovered-the-higgs-boson-uses-aws-to-probe-nature/, March 2016

[4] *What is HTCondor?*, https://research.cs.wisc.edu/htcondor/description.html

[5] *The GCE Grid Type*, HTCondor documentation http://research.cs.wisc.edu/htcondor/manual/v8.4/5_3Grid_Universe.html#SECTION00637000000000000000

[6] GlideinWMS, http://glideinwms.fnal.gov/

[7] Scientific Linux, https://www.scientificlinux.org/

[8] GCP, Compute Image Packages, https://github.com/GoogleCloudPlatform/compute-image-packages

[9] IETF, *The OAuth 2.0 Authorization Framework*, https://tools.ietf.org/html/rfc6749, October 2012

[10] Stackdriver Monitoring, *Metrics List*, https://cloud.google.com/monitoring/api/metrics

[11] Fifemon, https://fifemon.fnal.gov/

[12] Graphite, *Graphite Overview*, http://graphite.readthedocs.io/en/latest/overview.html

[13] Grafana, http://grafana.org/

[14] The Python Standard Library, *pickle – Python object serialization*, https://docs.python.org/2/library/pickle.html

[15] Fifemon base classes, https://github.com/fifemon/probes/tree/master/bin/fifemon

[16] Davide Grassano, *Benchmarking of public and local cloud resources*, http://eddata.fnal.gov/lasso/summerstudents/papers/2015/Davide-Grassano.pdf

[17] Amazon EC2 Instance Types, https://aws.amazon.com/ec2/instance-types/

[18] Mu2e experiment, Fermilab https://mu2e.fnal.gov/

Appendix A

Python excerpts

Listing 2: GCE Instances Number Monitor

```
def setup(json_path):
    os.environ['GOOGLE_APPLICATION_CREDENTIALS'] = json_path

    try:
        credentials = GoogleCredentials.get_application_default()
        return discovery.build('compute', 'v1', credentials=credentials)
    except Exception as e:
        logger.error('Cannot talk to GCE: %s',e)
        return None

def get_gce_instances(client, project, zone):
    counts = defaultdict(int)
    pageToken = None

    if client != None:

        # while True .... if .... break -> do..while behavior
        while True:
            result = client.instances().list(
                                        project=project,
                                        zone=zone,
                                        pageToken=pageToken).execute()
            pageToken = result.pop('nextPageToken', None)

            if 'items' not in result:
                logger.info('No instances found in %s' % zone)
                return counts
            for instance in result['items']:
                if instance['status'] == 'RUNNING':
```

```
                    machine_type = instance['machineType'].split('/').pop()

                    metric_name = '{machine_type}.count'.format(
                                machine_type=fifemon.graphite.sanitize_key(machine_type))
                    counts[metric_name] += 1

            if pageToken == None:
                break

        return counts
    else:
        return []

class GceProbe(fifemon.Probe):
    def __init__(self, *args, **kwargs):
        self.zones = kwargs.pop('zones', ['us-central1-a'])
        self.projects = kwargs.pop('projects', [None])
        self.client = setup(kwargs.pop('credentials', 'credentials.json'))

        super(GceProbe, self).__init__(*args, **kwargs)

    def post(self):
        if self.client == None:
            return

        for project in self.projects:
            for zone in self.zones:
                data = get_gce_instances(self.client, project, zone)
                logging.info('queried zone {0} for project {1}'.format(zone,project))
                if len(data) == 0:
                    continue
                if self.use_graphite:
                    try:
                        self.graphite.send_dict('{namespace}.{project}.{zone}'.format(
                                namespace=self.namespace,
                                project=project,
                                zone=zone),
                                data, send_data=(not self.test))
                    except Exception as e:
                        logging.error("error sending data to graphite: %s"%e)
                if self.use_influxdb:
                    logging.warning('InfluxDB is currently not supported')

# [...]
```

Listing 3: GCE CPU Utilization Monitor

```
def setup(json_path):
    os.environ['GOOGLE_APPLICATION_CREDENTIALS'] = json_path #

    try:
        credentials = GoogleCredentials.get_application_default()
        return discovery.build('monitoring', 'v3', credentials=credentials)
    except Exception as e:
        logger.error('Cannot talk to GCE: %s',e)
        return None

def mapToTime(x):
    return datetime.datetime.strptime(x['interval']['endTime'], '%Y-%m-%dT%H:%M:%S.%fZ')

def get_gce_cpu_usage(client, proj, startTime, endTime, filter):
    project = 'projects/{0}'.format(proj)
    pageToken = None
    obj = {'count': defaultdict(int), 'load': defaultdict(float)}

    while True:
        result = client.projects().timeSeries().list(name=project,
                                            interval_endTime=endTime,
                                            interval_startTime=startTime,
                                            filter=filter,
                                            pageToken=pageToken,
                                            aggregation_alignmentPeriod='300s',
                                            aggregation_perSeriesAligner='ALIGN_MEAN').execute()
        pageToken = result.pop ('nextPageToken', None)
        if 'timeSeries' not in result:
            break
        for instance in result['timeSeries']:
            zone = instance['resource']['labels']['zone']
            # eventually (when/if Stackdriver implements it),
            # this will look like instance['resource']['labels']['machineType']

            machineType = 'n1-standard-1'
            avgMeasurement = float(instance['points'][0]['value']['doubleValue'])
            metric = '{zone}.{machine_type}'.format(zone=zone, machine_type=machineType)

            obj['count'][metric] += 1
            obj['load'][metric] += avgMeasurement

        if pageToken == None:
            break

    cpu_loads = defaultdict(float)
    for key in obj['load']:
        cpu_loads['{0}.cpu_load'.format(key)]=obj['load'][key]/max(1,obj['count'][key])
```

```
    return cpu_loads

class GceProbe(fifemon.Probe):
    def __init__(self, *args, **kwargs):
        self.zones = kwargs.pop('zones', ['us-central1-a'])
        self.projects = kwargs.pop('projects', [None])
        self.client = setup(kwargs.pop('credentials', 'credentials.json'))

        super(GceProbe, self).__init__(*args, **kwargs)

    def post(self):
        if self.client == None:
            return

        now=datetime.datetime.utcnow()
        endTime = now.isoformat() + 'Z'
        startTime = (now - datetime.timedelta(minutes=5)).isoformat() + 'Z'
        filter = 'metric.type = "compute.googleapis.com/instance/cpu/utilization"'

        for project in self.projects:
            data = get_gce_cpu_usage(self.client, project, startTime, endTime, filter)
            logging.info('queried project {0}'.format(project))
            if len(data) == 0:
                continue
            if self.use_graphite:
                try:
                    self.graphite.send_dict('{namespace}.{project}'.format(
                            namespace=self.namespace,
                            project=project),
                            data, send_data=(not self.test))
                except Exception as e:
                    logging.error("error sending data to graphite: %s"%e)
            if self.use_influxdb:
                logging.warning('InfluxDB is currently not supported')
# [...]
```

Appendix B

Mu2e Spreadsheet

# cores provided	**150000**	cores	
jobs/day/core	**5**		
# of jobs	**500,000**	**2,000,000**	**5,000,000**
output_size/job	**0.146484375**	GB	
base_code_size	**10**	GB	
# transfers GS→FNAL / day	**1**		
Hard disk size	**9**	GB	
Hard disk cost/month	0.36		
Jobs/day	750000		

Google Compute Engine pricing		
Service	Price	Unit
Custom CPU	0.02663	$/core/hour
Custom memory	0.00357	$/GB/hour
Network egress	0.08	$/GB
Standard hard disk	0.04	$/GB/month

VM catalogue

VM Specifications	# cores	RAM (GB)	Price/hour
custom	**1**	**2**	*0.03377*
n1-standard-1	1	3.75	0.038
n1-standard-2	2	7.5	0.076
n1-standard-4	4	15	0.152
n1-standard-8	8	30	0.304
n1-standard-16	16	60	0.608

 # VMs req'd

VM Type	# VMs req'd
custom	150000
n1-standard-1	150000
n1-standard-2	75000
n1-standard-4	37500
n1-standard-8	18750
n1-standard-16	9375

 # days for all jobs

# jobs	# days req'd
500000	*0.666666667*
2000000	*2.666666667*
5000000	*6.666666667*

Total CPU + RAM cost

VM Type	500000	2000000	5000000
custom	81048	324192	810480
n1-standard-1	91200	364800	912000
n1-standard-2	91200	364800	912000
n1-standard-4	91200	364800	912000
n1-standard-8	91200	364800	912000
n1-standard-16	91200	364800	912000

Storage costs

VM type	500,000	2,000,000	5,000,000
custom	1200	4800	12000
n1-standard-1	1200	4800	12000
n1-standard-2	1200	4800	12000
n1-standard-4	1200	4800	12000
n1-standard-8	1200	4800	12000
n1-standard-16	1200	4800	12000

1st scenario
Network costs

# jobs	Traffic (GB)	Cost
500000	73242.1875	5859.375
2000000	292968.75	23437.5
5000000	732421.875	58593.75

2nd scenario
Network costs

# jobs	Traffic (GB)	Cost
500000	73242.1875	5859.375
2000000	292968.75	23437.5
5000000	732421.875	58593.75

Storage costs

# jobs	Storage (GB)	Cost
500000	73242.1875	65.1041666667
2000000	109863.2813	390.625
5000000	109863.2813	976.5625

Total cost by machine type and # of jobs	500,000 jobs		2,000,000 jobs		5,000,000 jobs	
	1st scenario	2nd scenario	1st scenario	2nd scenario	1st scenario	2nd scenario
custom	88,107	88,172	352,430	352,820	881,074	882,050
n1-standard-1	98,259	98,324	393,038	393,428	982,594	983,570
n1-standard-2	98,259	98,324	393,038	393,428	982,594	983,570
n1-standard-4	98,259	98,324	393,038	393,428	982,594	983,570
n1-standard-8	98,259	98,324	393,038	393,428	982,594	983,570
n1-standard-16	98,259	98,324	393,038	393,428	982,594	983,570

All prices are in USD