IOP Conf. Series: Journal of Physics: Conf. Series **898** (2017) 052041 doi:10.1088/1742-6596/898/5/052041

Virtual machine provisioning, code management, and data movement design for the Fermilab HEPCloud Facility

S Timm[1], G Cooper, S Fuess[1], G Garzoglio[1], B Holzman[1], R Kennedy[1], D Grassano[1], A Tiradani[1], R Krishnamurthy[2], S Vinayagam[2], I Raicu[2], H Wu[2], S Ren[2] and S-Y Noh[3]

[1] Scientific Computing Division, Fermilab, Batavia, IL 60563, USA
[2]Computer Science Dept., Illinois Institute of Technology, Chicago, IL 60616 USA
[3]Korea Institute of Science and Technology Information, Daejeon, Korea
Email: timm@fnal.gov

Abstract. The Fermilab HEPCloud Facility Project has as its goal to extend the current Fermilab facility interface to provide transparent access to disparate resources including commercial and community clouds, grid federations, and HPC centers. This facility enables experiments to perform the full spectrum of computing tasks, including data-intensive simulation and reconstruction. We have evaluated the use of the commercial cloud to provide elasticity to respond to peaks of demand without overprovisioning local resources. Full scale data-intensive workflows have been successfully completed on Amazon Web Services for two High Energy Physics Experiments, CMS and NOvA, at the scale of 58000 simultaneous cores. This paper describes the significant improvements that were made to the virtual machine provisioning system, code caching system, and data movement system to accomplish this work. The virtual image provisioning and contextualization service was extended to multiple AWS regions, and to support experiment-specific data configurations. A prototype Decision Engine was written to determine the optimal availability zone and instance type to run on, minimizing cost and job interruptions. We have deployed a scalable on-demand caching service to deliver code and database information to jobs running on the commercial cloud. It uses the frontier-squid server and CERN VM File System (CVMFS) clients on EC2 instances and utilizes various services provided by AWS to build the infrastructure (stack). We discuss the architecture and load testing benchmarks on the squid servers. We also describe various approaches that were evaluated to transport experimental data to and from the cloud, and the optimal solutions that were used for the bulk of the data transport. Finally, we summarize lessons learned from this scale test, and our future plans to expand and improve the Fermilab HEP Cloud Facility.

1. Introduction to the HEPCloud Project

The Fermilab HEPCloud Facility enables High Energy Physics experiments to perform the full spectrum of computing tasks, including data intensive computing and reconstruction, using commercial clouds as an extension of the Fermilab facility. The first year of the project successfully demonstrated data-intensive computing for two experiments. The use case for the Compact Muon Solenoid experiment at CERN (CMS) generated 800TB of output over the course of a month of running 58000 compute cores on Amazon Web Services (AWS)[1]. The use case of the NOvA experiment at Fermilab anticipated 2-3 TB both of input and output in an estimated 2 million hours of computing[1]. In addition to the significant transfer of inbound and outbound data that is being processed in these high-throughput computing tasks, both applications also need to contact remote

databases across the wide-area network. They also have very large code bases that need to be transferred to the remote cloud. The code base and database servers are already accessible via web protocols; all that remains is to deploy a scalable web caching solution in the cloud.

In preparation for the first phase of the HEPCloud project we scale-tested the auxiliary caching services that are used for code movement and database query caching, to be sure they could handle the expected load. We also carefully benchmarked the compute speed, the available network transfer bandwidth, and the throughput to the cloud-based storage system. This work was necessary to accurately plan the budget for these projects and estimate their total duration. We describe the architecture of the scalable services, the methodology used to stress-test them, and the benchmark results. We will then describe the benchmarking work that was done on the Amazon Web Service compute instances, network bandwidth and S3 storage service components.

2. Scalable Services Definition and Description

The Amazon Elastic Compute Cloud (EC2) provides virtual machines on demand that are known as instances, and have software contained in Amazon Machine Images (AMI). Block storage can be attached to these instances either as local ephemeral disks or from the Elastic Block Storage (EBS). The Simple Storage Service (S3) is used for object storage of data. Amazon Web Services [10] provides load balancing and orchestration services that allow users to organize raw AWS instances into an organized and scalable web service, in this case a distributed web caching service. Amazon Web Services are deployed in units known as regions. In the summer of 2015 there were three regions in the United States, us-east-1, us-west-1, and us-west-2 corresponding to Northern Virginia, Northern California, and Oregon respectively. Each region is further divided into several availability zones. For example, us-west-2 has availability zones us-west-2a, us-west-2b, and us-west-2c, each of which is distributed across multiple data centers. The goal is to deploy a scalable caching service in each availability zone.

The Amazon Auto Scaling service provides the functionality to increase or decrease the number of identical instances in a group of instances. It can be configured to scale up or down in response to a variety of conditions, including values measured by the CloudWatch monitoring service. These include operating system metrics such as CPU load, incoming or outgoing network bandwidth, and disk activity.

Elastic Load Balancing provides a unified scalable frontend to the collection of servers. It distributes load across servers in configurable ways. It also does health checks on the servers before adding them to the service, and removes them from the service if they are not healthy.

The compute servers which make use of the web caching service discover it by means of the Amazon Route 53 Domain Name System (DNS) web service. This service can dynamically attach an external or internal DNS name to a variety of AWS services, including the Elastic Load Balancer, EC2 instances, or S3 buckets. It can also be used in a fail-over mode to route users to services outside of AWS.

AWS CloudFormation is used to orchestrate and manage a collection of related AWS resources, provisioning and updating them in an orderly and predictable fashion. The resources and the relationships between them are declared in a template of JSON format, which can also have parameters input at launch time.

We use the AWS services above to serve web content to two open-source services. The CERN VM File System (CVMFS) for code distribution [3], and the Frontier distributed database caching system [11]. The CVMFS system is a read-only file system. Files are fetched from a distributed set of servers and cached locally on the worker node in a POSIX file system which is mounted via FUSE in user space. Only the parts of the software needed to actually run the workflow are downloaded to the local node. CVMFS functions best with a squid[2] proxy server or servers located on site. Many High Energy Physics collaborations use CVMFS to distribute their code.

The Frontier distributed database caching system provides a scalable RESTful http-based protocol to access a variety of databases. It can be configured to do multiple tiers of caching. Frontier relies on the squid proxy server to cache the data at local sites around the world. It is ideal for distributing data in applications that read mostly the same data at the same time. The databases in question contain

information about experimental conditions and configuration. The Frontier project distributes a special squid server package which automatically starts one squid server per core on a multiple core machine with settings optimized for caching Frontier data.

3. Architecture and Implementation

One scalable web caching service is instantiated in each Availability Zone. We use CloudFormation to deploy a software stack that includes an Autoscaling Group of squid servers, CloudWatch metrics to measure the aggregate network traffic flow, an Elastic Load Balancer to access the service, and a Route 53 DNS address to contact the Elastic Load Balancer. Both the Elastic Load Balancer and the squid servers behind it are accessible only to the other machines in our AWS Virtual Private Cloud and are not open to the public Internet. The system begins with a single squid server. More squid servers are created automatically when the outbound network bandwidth of the squid server is higher than 18MB/minute for 5 minutes. Similarly, when squid servers are idle for some time they are automatically scaled down. The code caching load peaks during periods when large numbers of jobs are starting.

The compute jobs run in separate EC2 compute instances which are launched as needed. These instances have a script that runs at boot time which detects which availability zone it has been launched in and then modifies the CVMFS client configuration files and Frontier configuration files to point to the web caching server in the corresponding availability zone. For example, in availability zone us-west-2a the script pointed to the internal DNS address elb2.us-west-2a.elb.fnaldata.org, which was resolvable only inside AWS. For Frontier database caching we found that it was also beneficial to have a local squid server running on every compute instance as well. For a compute instance with N cores this decreased the Frontier traffic through the load balancer by a factor of N.

4. Web Caching Server Capacity Measurements

We needed to demonstrate that a scalable caching solution could meet the bandwidth and reliability requirements of our application. The typical CMS workflow causes 1-1.5 GB of code and 0.5GB of database information to be cached per job. Squid servers are limited by network bandwidth so for these initial server tests we compared two Amazon instance types: m3.xlarge which has 4 CPU cores and an average network bandwidth of 1Gbit/sec, and m3.large which has 2 CPU cores and average network bandwidth of 700Mbit/sec. We also created client machines by installing the CVMFS client RPM. For these tests the clients were of instance type m3.medium. We ended up running squid servers on c3.xlarge instances due to their relatively better network bandwidth which is variable but averages 2-3Gbit/s.

We simulated the load through two different scripts. One called largequery made repeated requests (2500 in parallel) for the same 10MB file, for a total of 2.5TB of total throughput per client. Smallquery fetched a very small file a very large number of times (312500) and was designed to test the total number of requests that the load balancer could serve.

Figure 1 shows the network throughput of the three frontier-squid servers that were activated in the course of the largequery test. The three squid servers turn on one at a time as the high load continues. The full throughput of the system, including the clients, load balancers, and servers, is limited only by the maximum network throughput that the clients and servers can generate. The Elastic Load Balancer is found not to be a network traffic bottleneck. This is important because all network traffic to and from the squid servers does go through the load balancer.

Figure 2 shows the number of the requests per minute that were coming into the load balancer due to the smallquery script. It shows that the load balancer can easily handle up to 500,000 requests per minute without having any dropped requests. We observe that during periods of high load the elastic load balancer DNS entry starts to contain more IP addresses in the list of IP addresses that it returns.

We have successfully demonstrated a sustained network bandwidth that is greater than the anticipated bandwidth that will be required for database caching and code caching in our largest use case. We have demonstrated that the load balancing structure does not adversely affect network bandwidth and that it results in no dropped requests.

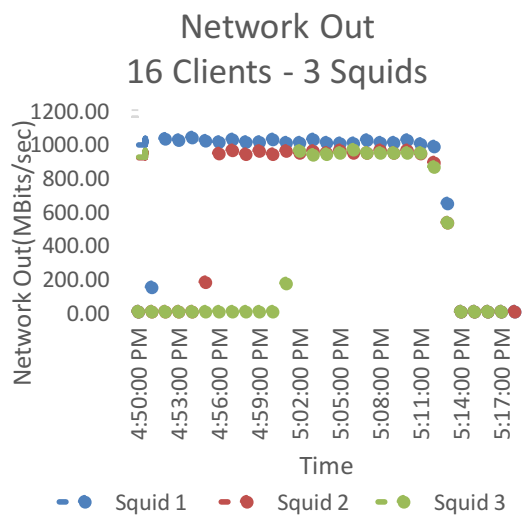


Figure 1: Network throughput of Squid Server

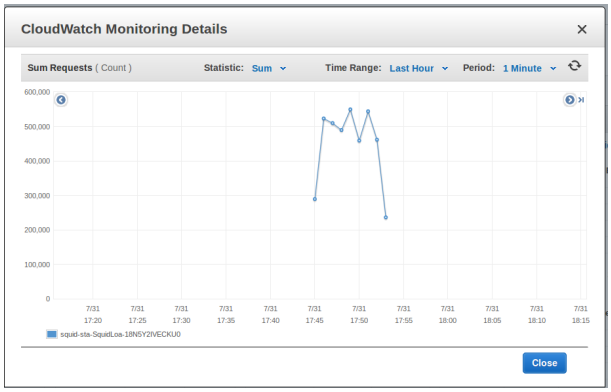


Figure 2: Load Balancer Requests per Minute

5. CPU and Bandwidth Benchmarking Description

We studied CPU and network bandwidth of AWS instances and local cloud resources, with the purpose of using them for a full scale CMS job. The benchmarks used were GENSIM, HS06, and some custom-made bandwidth benchmarks. The typical CMS job workflow includes generation of physics events (GEN), simulating how they decay in the detector (SIM), digitization of the detector response (DIGI) and reconstruction into tracks (RECO). The first two phases, GEN and SIM are combined into the benchmark. We do generation and simulation of 150 ttbar events. This benchmark is an accurate predictor of how long the real jobs will run, and is a good way to determine that a CMS job will run smoothly without failing. The results are given as total ttbar/s and ttbar/s per core. By running the benchmark multiple times on the same machines, it was determined that the results were very consistent, with maximum standard deviation obtained of 2%.

The HS06 (HEP SPEC 06) is a subset of the SPEC CPU2006 benchmarks that are written in C++. This is a similar mix to the instructions in High Energy Physics code. Its purpose is to stress the CPU and compiler of the system for both integer and floating point calculations. Since it is a generic benchmark, the obtained results will be comparable to a much wider set of machines. The results are given by the HS06 value, which is obtained by calculating the geometric mean of the inverted ratios between the running time for each benchmark in the package and the respective associated weight. Before calculating the geometric mean, the ratios are averaged over 3 runs of the benchmarks.

The bandwidth tests have been carried out through the usage of custom made scripts that employ the same transfer protocols and storage systems that will be adopted during the execution of a CMS job. Amazon S3 storage is one of the possible solutions for storing intermediary files that need to be written by the first phase of the job and read by the second phase. In order to test it, the high level ‘aws s3 cp’ command from the AWS CLI was used to simultaneously transfer 1, 10 and 100 1GB files, from up to 25 VMs at the same time. By doing this test we hoped to determine whether we would see any outright failures of fetches from S3.

In order to store the final results of the job, two storage options were considered. The globus-url-copy and xrdcp commands were adopted respectively to transfer to 2 different servers, dCache[9] and EOS[8]. We used fndca1 which is the public dCache server at Fermilab. Due to the high latency from Amazon to Fermilab, the file transfers had to be carried out by using multiple parallel streams, the best number of which was determined through a study of the parallelism parameter used by both commands. The globus-url-copy also allows to set the number of simultaneous TCP connections to

use at the same time. With the aim of simulating the data transfer of a CMS job, 1, 5, 10 and 20 1GB files were transferred simultaneously to the storage, from up to 25 VMs at the same time.

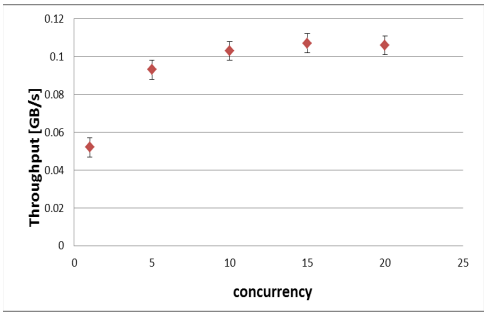


Figure 3: Study of the effect of the concurrency parameter on total throughput

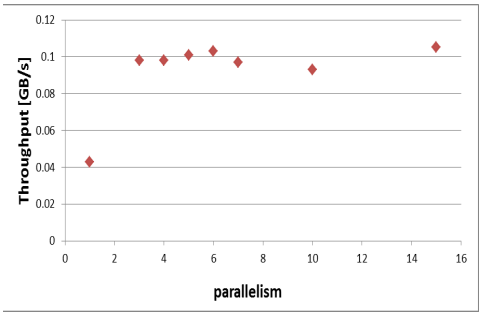
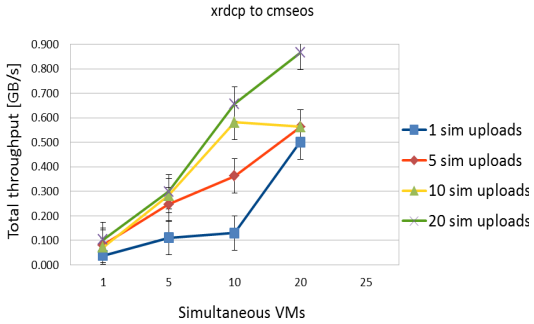
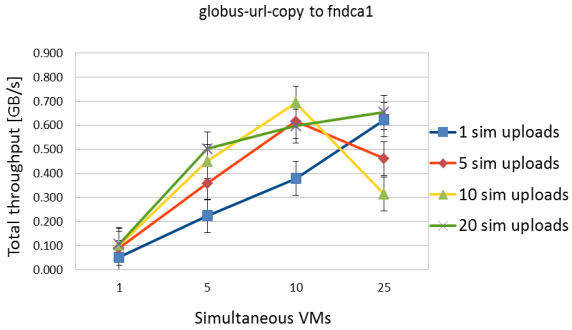


Figure 4: Study of the effect of the parallelism parameter on total throughput





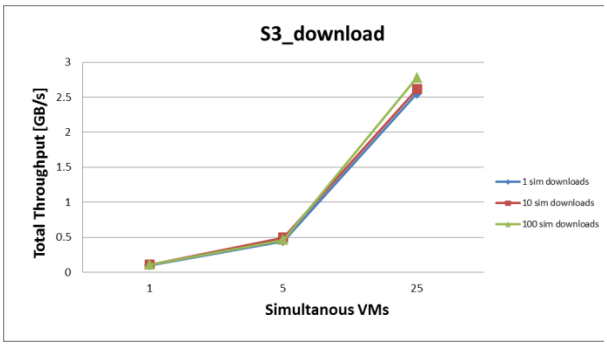


Figure 5 (a) Total throughput of xrdcp to cmseos server (b) total throughput of globus-url-copy to dCache server (c) total bandwidth to AWS S3

Table 1: Final Results from the GENSIM and HS06 benchmarks for AWS Instances.

Instance	Cores	Core type	GHz	$/hr	ttbar/s/ Core	ttbar/s total	(ttbar/s)/ ($/hr)	HS06/ core	HS06 Total	HS06/ ($/hr)
m3.xlarge	4	E5-2670	2.50	0.266	0.0139	0.056	0.209	14.3	57	215
m3.2xlarge	8	E5-2670	2.50	0.532	0.0139	0.111	0.208	12.2	98	184
m4.xlarge	4	E5-2676	2.40	0.252	0.0201	0.081	0.320	16.1	65	256
m4.2xlarge	8	E5-2676	2.40	0.504	0.0191	0.153	0.304	15.1	121	240
m4.4xlarge	16	E5-2676	2.40	1.008	0.0198	0.317	0.315	13.5	217	215
c3.xlarge	4	E5-2680	2.80	0.210	0.0153	0.061	0.291	14.9	59	283
c3.2xlarge	8	E5-2680	2.80	0.420	0.0153	0.122	0.291	14.7	118	281
c3.4xlarge	16	E5-2680	2.80	0.840	0.0149	0.239	0.284	13.2	212	252
c4.xlarge	4	E5-2666	2.90	0.220	0.0228	0.091	0.415	17.5	70	318
c4.2xlarge	8	E5-2666	2.90	0.441	0.0226	0.181	0.410	16.5	132	300
c4.4xlarge	16	E5-2666	2.90	0.882	0.0205	0.327	0.371	14.8	237	268
r3.xlarge	4	E5-2670	2.50	0.350	0.0151	0.060	0.172	15.5	62	177
r3.2xlarge	8	E5-2670	2.50	0.700	0.0150	0.120	0.171	14.2	114	162
r3.4xlarge	16	E5-2670	2.50	1.400	0.0146	0.233	0.166	12.7	203	145
cc2.8xlarge	32	E5-2670	2.60	1.090	0.0141	0.450	0.413	11.2	359	329

Table 2: Final results from the GENSIM and HS06 benchmarks for local machines

N_CORE	CORE TYPE	GHz	ttbar/s-core	ttbar/s total	HS06/core	HS06 total
8	Intel XEON X5355	2.66	0.0179	0.143	8.3	67
8	AMD Opteron 2389	2.90	0.0217	0.174	12.0	96
16	AMD Opteron 6134	2.30	0.0173	0.277	9.9	159
32	AMD Opteron 6128	2.00	0.0149	0.477	8.6	277
32	AMD Opteron 6134	2.30	0.0162	0.520	9.5	302
64	AMD Opteron 6376	2.30	0.0136	0.873	10.0	640
64	AMD Opteron 6376	2.30	0.0136	0.868	9.5	607
1	E5-2660V2	2.20	0.0193	0.019	17.9	18
1	E5-2660V2	2.20	0.0192	0.019	17.8	18
8	E5-2660V2	2.20	0.0182	0.145	14.3	115
1	E5640	2.60	0.0229	0.023	18.4	18
8	E5640	2.60	0.0217	0.174	15.2	122
1	Intel XEON X5355	2.66	0.0173	0.017	13.2	13
4	Intel XEON X5355	2.66	0.0170	0.068	11.4	46

6. Results of Benchmarking

The results for the GENSIM and HS06 benchmarks are reported in Table 1 and Table 2. The cost model adopted in this analysis is based on the on-demand pricing of AWS instances, which is indicative of the '0.25 of the on-demand' algorithm that is being considered for the spot market. From the cost effectiveness alone, the best machines that have been observed would be those from the c4 and m4 series, but the c4 and m4 instances do not have local ephemeral disk so we must attach Amazon Elastic Block Storage (EBS) at extra cost. Thus the c3 instances, particularly the c3.2xlarge, are the optimal instances with enough memory, disk, and bandwidth to run a CMS job in a cost-effective manner. In order to compare local real and virtual machines with the AWS ones, the same benchmarks have been run on our physical hardware and on virtual machines in our local private cloud, FermiCloud. The results, presented in Table 2, show that the performance measurements of local and public cloud machines are similar. In production, our actual mix of AWS instances averaged 0.0158 ttbar/s per core, while the makeup of our cluster at Fermilab averaged 0.0163 ttbar/s per core [7].

We first analyzed the optimal parallelism and concurrency parameters for dCache and EOS transfers, in order to obtain the maximum efficiency for the minimum required number of inbound connections. From the analysis of the data reported in Figures 3 and 4, it was concluded that the best solution was to set parallelism at 4 and concurrency at 5. More concurrent simultaneous transfers would cause some of the upload requests to time out due to limited number of listeners at the Fermilab end.

Using the globus-url-copy command toward the fndca1 dCache server, and the xrdcp command toward the CMS EOS server, the upload bandwidth throughput from c3.2xlarge instances was analyzed. The results reported in Figures 5 (a) and (b) show that we were able to reach a maximum bandwidth of 0.7 GB/s (5.6Gbit/s) to fndca1 dCache with the globus-url-copy and 0.9 GB/s (7Gbit/s) with the xrdcp to CMS EOS. Both of these are large storage services with multiple machines to receive the data on our end.

We next measured the bandwidth throughput from amazon c3.2xlarge instances to Amazon S3. The results of the bandwidth analysis for reading from S3 are reported in Figure 5(c), from which it was concluded that no matter how much we would stress Amazon S3 within the capabilities of our AWS account, we would always get all the requested bandwidth, with the only limit being the network throughput of the c3.2xlarge instance. We observed no failures. S3 was used in our production AWS runs to read the "pile up" which is collections of other background interactions which are blended into the event we are simulating. We stored one copy of the pile-up in each AWS US-based region.

7. Summary

Through this process of CPU benchmarking, we identified several Amazon instance types that were suitable for our experimental use cases. In practice we ended up using all of the instance types that we benchmarked for maximum availability. The production software that we ran showed very similar performance characteristics to the benchmarks that we had done before the run[7].

We also demonstrated a total network throughput capacity from Amazon virtual machines to the Fermilab storage systems that was 2.5 times larger than the rate of data that was actually generated by the jobs in the CMS use case. Given the actual network connectivity between Fermilab and Amazon via the ESNet research network (100Gbit/s to some regions) we expect that eventually we can do even better and believe that we may currently be limited by the number of simultaneous files our servers can receive at once. The combination of sufficient caching service, network throughput, storage bandwidth, and compute instances show that we are ready to analyze data in bulk on public clouds.

References

[1] Timm S, *et al* Virtual Facility at Fermilab: Infrastructure and Services Expand to Public Clouds. In *The International Symposium on Grids and Clouds (ISGC),* volume 2015, 2015.
[2] Squid - HTTP proxy server http://www.squid-cache.org, 2015

[3] Blomer J *et al*, Status and Future Perspectives of CernVM-FS J. Phys.: Conf. Ser. 396052013, doi:10.1088/1742-6596/396/5/052013
[4] Wu H, Ren S, Timm S, Garzoglio G, Noh S, Experimental Study of Bidding Strategies For Scientific Workflows using Spot Instances. Submitted to MTAGS workshop Nov. 2015.
[5] Bernabeu G *et al*, Cloud Services for the Fermilab Scientific Stakeholders. CHEP workshop 2015 *J. Phys.: Conf. Ser. 664 022039*.
[6] Boyd J *et al* 2016 Advances in Grid Computing for the FabrIc for Frontier Experiments Project at Fermilab, Proceedings of CHEP2016.
[7] Bauerdick L *et al*, 2016 HEPCloud, a new paradigm for HEP facilities: CMS Amazon Web Services Investigation, FERMILAB-PUB-16-170-CD.
[8] Peters AJ *et al*, 2015 EOS as the present and future solution for data storage at CERN, J. Phys.: Conf. Ser. 664 (2015) 042042 doi:10.1088/1742-6596/664/4/042042
[9] Fuhrman P *et al,* 2000 A distributed rate-adapting buffer cache for mass storage systems, CHEP 2000 Proceedings, Computer Physics Communications 140(1).
[10] Amazon https://aws.amazon.com/ec2/ 2015.
[11] Blumenfeld B, Dykstra D, Lueking L, Wicklund E (2008) CMS conditions data access using FroNTier. J Phys: Conf Ser 119:072007

Acknowledgments

Fermilab is operated by the Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the United States Department of Energy. This work is also supported by KISTI and Fermilab under the joint Cooperative Research and Development Agreement CRADA-FRA 2015-001 / KISTI-C15005.