Provisioning on EC2 with HTCondor

Condor Week 2016

John Hover <jhover@bnl.gov>
Jose Caballero Bejar <jcaballero@bnl.gov>
Carlos Gamboa<cgamboa@bnl.gov>
Michael O'Connor<moc@bnl.gov>

Purpose of this talk

Half project report, half tutorial.

How did ATLAS run at large scale (thousands of VMs, tens of thousands of cores) on AWS?

- Sufficient details in slides and references to do the same.
- NOTE: Several slides have details we won't delve into. Here for reference.

Can *we* run on AWS?

- Enough info to determine if it can be done efficiently/economically.
- Some general guidelines/strategies.

Context

US ATLAS facility wanted option of moving some work to AWS in order to free up dedicated data centers.

AWS initiated intensive demo project funded by US$200K grant in credits. Goal was for ATLAS to adapt fully to run in production on AWS; demonstrate the ability to do so economically.

Site visits. Weekly phone tag-ups.

Engineering team support, Spot team advice.

ATLAS/BNL involvement: data, networking, workload system, provisioning

Role of HTCondor

We (ATLAS) already have a provisioning utility (AutoPyFactory) that is a layer on top of Condor-G used to submit Grid pilot jobs to all ATLAS sites in the world.

Used heavily by ATLAS sites as local batch system (including Brookhaven Laboratory)

HTCondor becoming de-facto CERN standard batch system.

Made sense to use for provisioning VMs on EC2.

And obviously sensible to use as virtual cluster batch system to run pilot jobs.

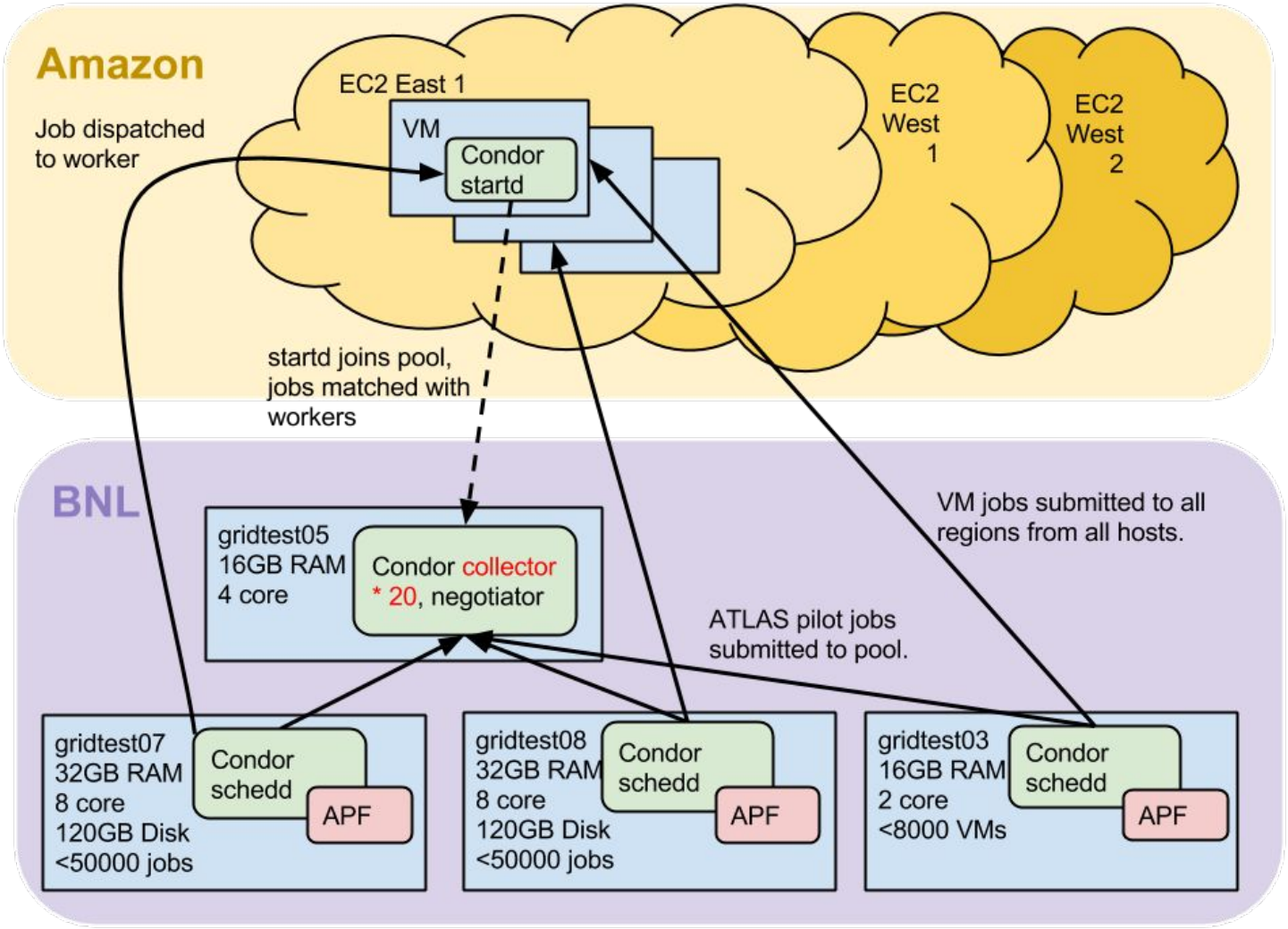
Overall HTCondor Amazon Architecture

Hybrid cloud.

> Static components set up on dedicated computers at Brookhaven Lab. Three substantial hosts with good network and public IPs.
>
> Only execute (startd) hosts are run on VMs in EC2; connecting back to pool at BNL.

A lot of these guidelines generally apply to any very large pool, with some additional WAN-related items (since workers may be all over the US, or the world).

Driving principle is to maximize the ability to provisiong *capacity* as needed.


Amazon
EC2 East 1
VM
Condor startd
EC2 West 1
EC2 West 2
Job dispatched to worker
startd joins pool, jobs matched with workers
BNL
gridtest05
16GB RAM
4 core
Condor collector * 20, negotiator
VM jobs submitted to all regions from all hosts.
ATLAS pilot jobs submitted to pool.
gridtest07
32GB RAM
8 core
120GB Disk
<50000 jobs
Condor schedd
APF
gridtest08
32GB RAM
8 core
120GB Disk
<50000 jobs
Condor schedd
APF
gridtest03
16GB RAM
2 core
<8000 VMs
Condor schedd
APF

OS-level tweaks {collector, negotiator, schedd}

In general, we're assuming that nodes have plenty of CPUs and RAM sufficient to run a compute-intensive service. Remove all the OS system and networking limits you can...

```
/etc/security/limits.conf
*    -    nofile          1000000
*    -    nproc         unlimited
*    -    memlock       unlimited
*    -    locks         unlimited
*    -    core          unlimited

/etc/sysctl.conf
fs.file-max = 1000000
```

```
sysctl -w net.core.rmem_max=8388608
sysctl -w net.core.wmem_max=8388608
sysctl -w net.core.rmem_default=65536
sysctl -w net.core.wmem_default=65536
sysctl -w net.ipv4.tcp_rmem='4096 87380 8388608'
sysctl -w net.ipv4.tcp_wmem='4096 65536 8388608'
sysctl -w net.ipv4.tcp_mem='8388608 8388608
8388608'
sysctl -w net.ipv4.route.flush=1
```

HTCondor pool config and scaling {collector, negotiator}

- Multi-process collector
  - Collector process is single-threaded, so suffers from some scalability issues.
  - Instead, setup to use multiple collector processes:

```
COLLECTOR_SOCKET_BUFSIZE = 20480000
COLLECTOR_HOST=$(CONDOR_HOST):29650
COLLECTOR1 = $(COLLECTOR)
COLLECTOR2 = $(COLLECTOR)
COLLECTOR1_ARGS = -f -p 29660
COLLECTOR2_ARGS = -f -p 29661
COLLECTOR_NAME=$(FULL_HOSTNAME)
COLLECTOR1_ENVIRONMENT = "_CONDOR_COLLECTOR_LOG=$(LOG)/Collector1Log …"
COLLECTOR2_ENVIRONMENT = "_CONDOR_COLLECTOR_LOG=$(LOG)/Collector2Log …"
DAEMON_LIST = $(DAEMON_LIST) COLLECTOR1 COLLECTOR2
```

See: https://htcondor-wiki.cs.wisc.edu/index.cgi/wiki?p=HowToConfigCollectors

HTCondor pool config and scaling {collector, negotiator}

- Depth first filling of pool. To expose idle nodes ASAP.

```
NEGOTIATOR_POST_JOB_RANK =  (RemoteOwner =?= UNDEFINED) * \
                            (0 - cpus)
```

- **NEGOTIATOR_INTERVAL = 300**
  NEGOTIATOR_INFORM_STARTD = False

HTCondor pool config and scaling {schedd}

GRIDMANAGER_MAX_PENDING_REQUESTS = 100
GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE_EC2 = 30000
GRIDMANAGER_JOB_PROBE_INTERVAL = 600
GRIDMANAGER_JOB_PROBE_INTERVAL_EC2 = 600

Basically bump up any polling and max_X values by 50% to 100%.

HTCondor pool config and scaling {startd}

Use partitionable slots: Maximizes flexibility of EC2 image.

```
SLOT_TYPE_1 = 100%
NUM_SLOTS = 1
NUM_SLOTS_TYPE_1 = 1
SLOT_TYPE_1_PARTITIONABLE = True
SlotWeight = Cpus
```

Use slot users to separate jobs. Reduces config on VM. (No shared filesystem anyway).

```
SLOT1_USER = slot1
SLOT2_USER = slot2
DEDICATED_EXECUTE_ACCOUNT_REGEXP = slot.+
STARTER_ALLOW_RUNAS_OWNER = False
EXECUTE = /home/condor/execute
```

HTCondor pool config and scaling {startd} 2

Use the Condor Connection Broker (CCB)

This deals with the issue of workers behind a NAT or firewall. Automatically triggers the creation of the CCB service on the collector.

```
CCB_ADDRESS = $(COLLECTOR_HOST)
```

Allow user on schedd (if s/he has the pool password) to administer (e.g. shut down) the machine:

```
SEC_ENABLE_MATCH_PASSWORD_AUTHENTICATION  = True
ALLOW_WRITE = $(ALLOW_WRITE), submit-side@matchsession/*
ALLOW_ADMINISTRATOR = condor_pool@*/*
```

This capability is critical so we can actively scale the cluster up and down.

HTCondor pool config and scaling {all}

- Password authentication
  - Avoid GSI overhead
  - Can’t use hostname-based auth safely over WAN.
  - Password baked into image (make AMI private!)
- Match session re-auth
  - **`SEC_ENABLE_MATCH_PASSWORD_AUTHENTICATION = True`**
  - Prevents renegotation of authentication on each connection.
- Shared port
  - Allows service to multiplex communication through a single port.

```
USE_SHARED_PORT = TRUE
DAEMON_LIST =  $(DAEMON_LIST) SHARED_PORT
```

EC2 VM Config/Issues

EC2 attributes (e.g. AvailabilityZone) pulled from metadata service and published to startd classad:

```
EC2PublicIP
EC2PublicDNS
EC2InstanceID
EC2InstanceType
EC2AMIID
EC2AvailabilityZone
```

We use Puppet w/ Facter, but custom init script also usable. See: http://svn.usatlas.bnl.gov/svn/griddev/provisioning-templates/files/condor/condorconfig.init
EC2AvailabilityZone used in job requirements in multi-region pool.

Miscellaneous VM Configuration Notes

- Local ephemeral disk(s) turned into volume, formatted, and mounted at runtime as /home. All apps configured to assume /home is “large”.
- LHCB Fast-benchmark integrated with Condor startd benchmarking, value published in classad.
- CVMFS used for application software.
- CVMFS, HTCondor, and admin SSH authorized keys setup by Puppet
- No runtime software install/downloads.
- Amazon Machine Images (AMIs) built and configured using an Imagefactory/Puppet/Hiera -based system. See for more info:

https://docs.google.com/presentation/d/1s5x2KNDIrzlZRsKhhqvJCnN23zyUgSz6YYb2ZYLpW2A/edit?usp=sharing

AWS Configuration and Scaling Notes

- Instance types
  - All instance types have different mixes of ephemeral disk sizes, CPU count, memory, and network connectivity.
  - Block device mapping varies from type to type!
- EC2 Regions
  - Not all regions have the same instance types available!
- HVM vs. PV
  - Newer types only support HVM (Hardware virtualization). Older types only PV (Paravirtual)
  - Requires separate VM creation process. (Now supported by Imagefactory)
- Resource limit increases via request/tickets.
  - Many places where protections can prevent acquiring capacity at scale.
  - E.g. max number of EBS disks, Each instance uses one implicitly for the root partition.

AWS Configuration and Scaling

Instance type, region choice: Maximize variety for capacity and cost.

Spot bidding: Bid what resources are really worth to you.

For ATLAS $.03 cpu/hr was reasonable max. ($.25/hr for 8-core types, $.50/hr for 16-, etc.)
ATLAS minimum profile (2GB RAM, 20GB disk/core)

Type	Virt	vCPU	Mem	Storage	Bid/hr	Comments
m2.4xlarge	PV	8	68	2 x 840	$.25	
m3.2xlarge	HVM	8	30	2 x 80	$.25	
r3.2xlarge	HVM	8	61	1 x 160	$.25	
r3.4xlarge	HVM	16	122	1 x 320	$.50	
r4.8xlarge	HVM	32	244	2 x 320	$.80	
i2.4xlarge	HVM	16	122	4 x 800	$.50	
i2.8xlarge	HVM	32	244	8 x 800	$.80	
cr1.8xlarge	HVM	32	244	2 x 120	$.80	Not available in us-west-1
d2.4/8xlarge	HVM	16			$.50	Problem booting. Omitted.

Data & Workload Adaptations: PanDA

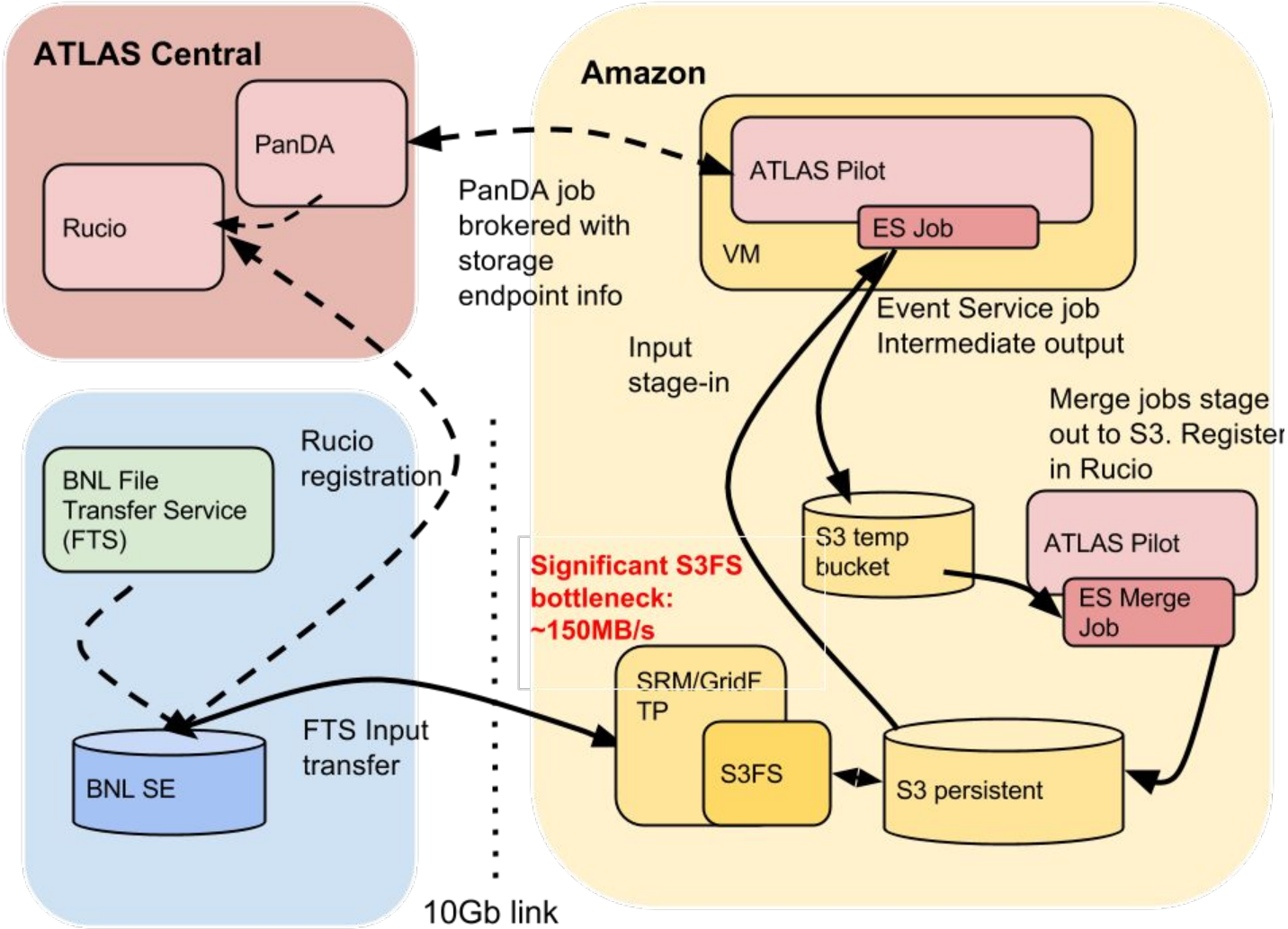
- PanDA (Production and Distributed Analysis) Scaling/enhancements.
  - Event Service
    - One job does work in small pieces (10-15 minutes) and saves intermediate output to S3. VM termination results in little lost work. Next job picks up where last stopped.
    - At the end, a short merge job pulls all pieces from S3, merges them, and stages out final result. Intermediate pieces then deleted, tracking discarded.
  - ATLAS DDM (Dynamic Data Management)
    - Make S3 a first-class endpoint type. Aggressively delete unneeded data.

Data & Workload Adaptations: Networking

- Normally AWS charges for exporting data from AWS (data *egress*)
- ESNet (the DOE research network provider) peered with Amazon at 3 locations.

Location	Bandwidth	EC2 Region
Reston Virginia	10GB (soon 40GB)	us-east-1
Seattle Washington	100GB	us-west-2
Sunnyvale California	100GB	us-west-1

- ATLAS recieved waiver as long as data costs were less than 15% of total compute bill.



Separate setup for each EC2 region. Each now a Tier 2.

Addl. test results in extra slides. Not long-term solution.

Data, Workload: General Guidelines

If you can make your jobs less than 30 minutes (20 minutes ideally) you don’t need checkpointing.

Best practice would be to put all relevant input data into S3 in advance of run. You don’t want to waste VM compute time copying large files in/out from outside AWS (even though ingress is free).

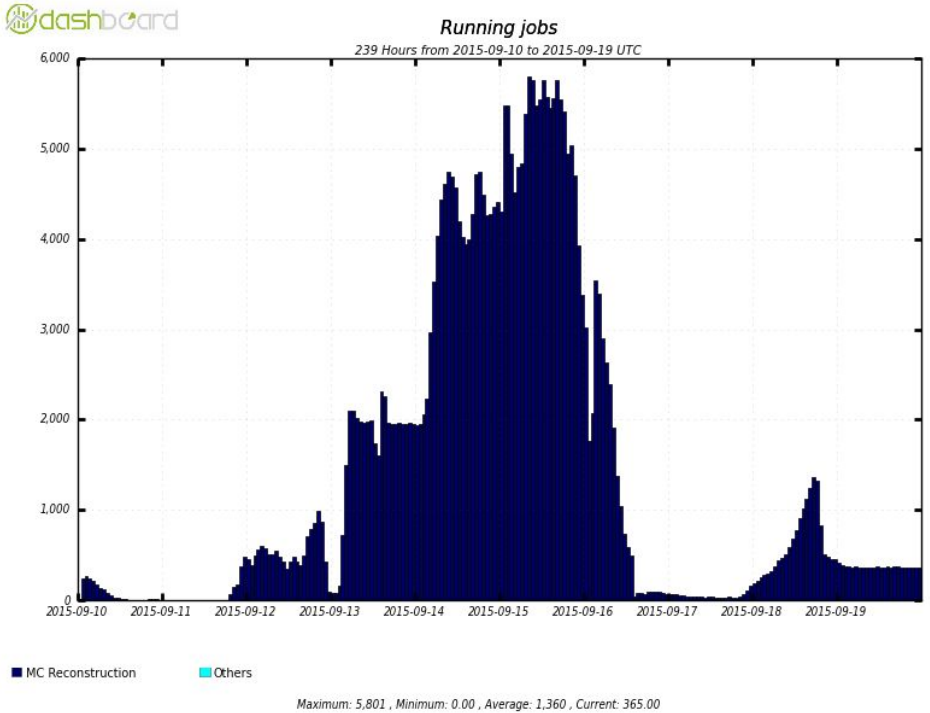
Determine in advance your total data egress volume. Is $ acceptable?

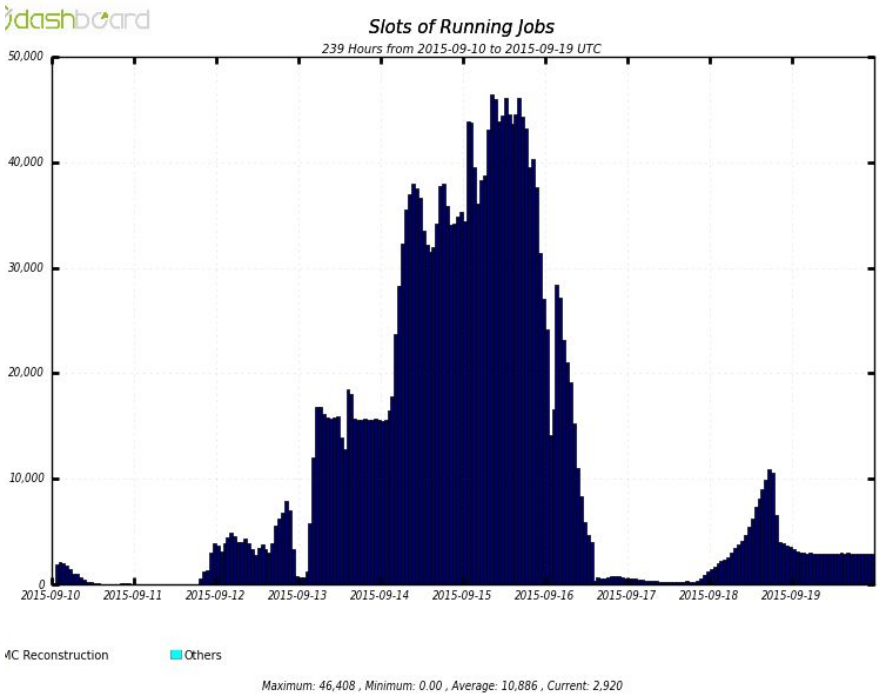
Plan to delete unneeded data sooner rather than later. Do it programmatically!

September 2015 ATLAS EC2 Run

- ~45k cores
- US East region only (networking not ready on west coast)
- 10GB dedicated bandwidth between AWS and BNL
- Input data automatically pre-subscribed (copied to S3).
- Output pulled from S3 asynchronously after the run.
- ~6000 jobs (8-core multicore)
- ~4000 simultaneous VMs: mix of 8-,16-, and 32-core types. (8 >> 16 > 32)
- Ran ~5 days. 437,000 jobs completed, 402,000
- 3.2 million CPUhrs
- Compute cost approx $57K, Data+storage around $500.

Jobs, cores, Sept 10 - Sept 20, 2015





Cost Details (September 2015)

Total compute bill AWS:	$57,000
Total storage element cost:	$7,700 (to be eliminated…)
Total data transfer/storage cost:	~$800
CPU hours billed by AWS:	3.22 million CPU/hrs
CPU hours seen and used by ATLAS:	2.53 million CPU/hrs (to be improved!)
Cost cpu/hr per AWS bill:	$.017
Cost cpu/hr per ATLAS "goodput"	$.022
Cost RACF cpu/hr per Tony Wong's* cost analysis:	$.04

Ignores machine performance, but common jobs were at most ~50% faster at RACF. *https://indico.cern.ch/event/247864/contributions/1570317/attachments/426677/592243/Operating_Dedicated_Data_Centers__Is_It_Cost-Effective.pdf

March 2016 100k-core run

We attempted a 3-region 100k- core (~6000 VM) run in March.

Encountered EC2 <RequestLimitExceeded> error due to hitting EC2 API denial-of-service protections. These apply to the API endpoints and apply to all users. Same problem also hit by CMS/Fermilab in February at same scale.

Once the problem was understood, HTCondor team rapidly changed how Condor-G responded to seeing the error (i.e. by backing off). New code has been tested at the 10,000 VM level with success.

Many thanks to Todd Miller et. al. for rapid response!

Next Steps for ATLAS; Relevant for anyone

Condor_annex

AWS Spot capacity tool

m4.* types (EBS only, no ephemeral)

Cleaner security (mechanism to pass in secrets that only root on the VM should see).

Complete a 100k-core, 3-EC2-region, ~7000 VM demonstration run.

Convert our provisioning utility over to the HTCondor Python library. (Thanks to Brian Bockelman!)

Questions?

References/Links

This talk:
https://docs.google.com/presentation/d/15ppLKZ6mtbflPUdV7XveG6gRv233WwTT8zVRztSV8r0/edit?usp=sharing

Similar OSG AWS Talk:

https://docs.google.com/presentation/d/1MJVFMZiOLjdvDdjZeJSg3ZNaQbe2lkFeVZwXBP0Vfg8/edit?usp=sharing

RACF/ATLAS Central Condor configuration file examples
http://svn.usatlas.bnl.gov/svn/griddev/provisioning-templates/condor/

HTCondor Scaling/EC2 Recipes
https://htcondor-wiki.cs.wisc.edu/index.cgi/wiki?p=HowToManageLargeCondorPools
https://htcondor-wiki.cs.wisc.edu/index.cgi/wiki?

Autopyfactory project

https://github.com/PanDAWMS/autopyfactory

https://github.com/PanDAWMS/autopyfactory-tools

Provisioning Toolkit Links:

https://docs.google.com/presentation/d/1s5x2KNDIrzlZRsKhhqvJCnN23zyUgSz6YYb2ZYLpW2A/edit?usp=sharing

http://svn.usatlas.bnl.gov/svn/griddev/provisioning-config/

http://svn.usatlas.bnl.gov/svn/griddev/provisioning-toolkit/

http://svn.usatlas.bnl.gov/svn/griddev/provisioning-templates/

Other Links:

https://twiki.cern.ch/twiki/bin/view/PanDA/EventServer

https://github.com/redhat-imaging/imagefactory