# Early Experience Using Amazon Batch for Scientific Workflows

Kyle M.D. Sweeney
Department of Computer Science and Engineering
Notre Dame, IN
ksweene3@nd.edu

Douglas Thain
Department of Computer Science and Engineering
Notre Dame, IN
dthain@nd.edu

## ABSTRACT

Recent technological trends have pushed many products and technologies into the cloud, relying less on local computational services, and instead purchasing computation a la carte from cloud service providers. These providers focus more on delivering technologies which are service based rather than throughput based. With the advent of Amazon Batch, a new high throughput service, we wished to see how capable it was for running scientific workflows compared to existing cloud services. To that end, we developed a testing suite which created workflows focusing on increasing shared file sizes, increasing unique file sizes, and increasing number of tasks, and ran the workflows on Amazon Batch plus two other similar configurations for comparison: EC2 workers and Work Queue on EC2. We found that while there is a significant delay in sending jobs to Amazon Batch and running raw EC2 workers, there is little overhead in the actual running of the task, and similar performance to using Work Queue on EC2 when the workflow does not require large input files. Additionally, when performing real a workflow, Batch achieved a speedup over Work Queue workers on EC2 instances of 1.18x.[1]

**ACM Reference format:**
Kyle M.D. Sweeney and Douglas Thain. 2018. Early Experience Using Amazon Batch for Scientific Workflows. In *Proceedings of 9th Workshop on Scientific Cloud Computing, Tempe, AZ, USA, June 11, 2018 (ScienceCloud'18),* 8 pages.
https://doi.org/10.1145/3217880.3217885

## 1 INTRODUCTION

Modern science requires powerful computation to sift through data and help make discoveries, advancing human knowledge. Traditionally, scientists have used high performance computing (HPC) centers to enable their work, often using workflows to transform their work into high-throughput computing as a means of distributing the work among many computers. However these centers are expensive to acquire and maintain, creating an impetus for alternative sources of computation. The cloud is seen as a promising alternative, as scientists only need to pay for what they use.

Amazon offers many cloud products such as Amazon Simple Storage Service (S3) and Amazon Elastic Cloud Computing (EC2) enabling customers to link them together and produce powerful custom services. However, these platforms and services are usually aimed at creating long running cloud services, not high-throughput computation. Amazon released Amazon Batch last year, a cloud based batch system. We wished to see what the capabilities of Amazon Batch for scientific workflows is by connecting it to a workflow management system. Given our background with Makeflow [1], we created back-end modules linking Makeflow to Batch, and tested Batch's performance against using plain EC2 workers, and running Work Queue (WQ) [2] on EC2. To create these modules, we had to answer some key questions when running remote jobs: how are the jobs placed and ran, how are files delivered from the jobs, and how are the jobs retrieved when finished. Amazon Batch works by provisioning a set of EC2 workers which fulfills user specified requirements. The EC2 workers then accept jobs from different queues, running them inside of containers. We wanted to test this batch system against simply provisioning a correctly sized EC2 instance for a given job, and another system which also implements a similar workers-on-pre-provisioned-workers system. Thus we chose to use Work Queue: a pilot job system which enables users to create their own cloud batch system out of a variety of different kinds of computing resources. For example, if a researcher has access to three different HPC sites, by starting workers on those sites, the user can run any number of jobs and workflows on those sites via Makeflow, which coordinates between the given resources.

Taking inspiration from Zhang et al's work on "Capability of Large Scale Computers" [3], we identified five metrics by which we would test our three systems: total workflow runtime, the histographic runtime, the job dispatch latency, the queue delay of the systems, and the job retrieval latencies. To isolate these features, we created a tool which allows us to generate customized workflows: customizing the input sizes, how long jobs run, the number of jobs, and having an output file or not. By customizing these workflows, we could then see in which areas the systems were stronger and weaker. We also ran a real workflow to see how Batch and Work Queue compared in practice.

We found that in our customized workflows, Work Queue had the smallest overhead when running jobs, and the fastest dispatch and retrieval rates, as well as being the quickest when running jobs, since it sends data the fastest to each job. The EC2 workers performed the worst in most fields, delivering data the slowest, and having higher execution times for the jobs themselves, although its queue delay was the lowest, thus it was able to dispatch jobs better than Batch. Batch performed between these two, having fairly consistent execution times and mostly low queue overhead. It was not able to deliver data as fast as Work Queue could, however. When running the real workflow, we found that Batch executed it

[1]This work was supported in part by National Science Foundation grant OAC-1642409


*ScienceCloud'18, June 11, 2018, Tempe, AZ, USA*

ACM ISBN 978-1-4503-5863-7/18/06…$15.00
https://doi.org/10.1145/3217880.3217885

faster than Work Queue since Batch transferred less data overall, running about 1.18x times faster. Thus, we conclude that Amazon Batch is a capable, cloud-level workflow system.

## 2 BACKGROUND

To test Amazon Batch, we used Makeflow, which is a workflow system designed to allow users to use already familiar concepts: workflows and make files, and combine them into an easy to use package. Make is a classic interface for describing many different jobs and how they come together, which is why it is a natural fit as the interface language to describe a workflow. To that end, Makeflow interfaces with several different job processing systems to execute the jobs given to it. It can run them locally, or interface with Condor [4] or other systems, abstracting away the exact protocols for those systems so the user can focus on their jobs.

To interface with all of these systems, Makeflow has a series of modules which adapts the actual interface of the workflow drivers to the idealized execution model of Makeflow. To accomplish this, it makes two assumptions. The first is that the underlying system can operate as a sandbox execution model. This means that every worker used by Makeflow is independent of all the other workers, and receives their input files separately. The second assumption is that each underlying system can fulfill the duties specified by Makeflow's Submit, Wait, and Kill methods.

In the sandbox execution model, Makeflow assumes that each task can be independently executed with the specified parameters, without affecting the other tasks. This means fulfilling the Cores, Memory, and Disk space requirements, as well as copying in the input files, and being able to then retrieve the specified output files. Many of the adapter modules accomplish this by pretending each job runs totally independently, and cache the input files whenever two jobs are on the same machine, and use the same inputs. This helps speed up execution by not needing multiple file copies.

Makeflow's three primary module functions, Submit, Wait, and Kill deal with handling jobs. Submit gives a job to the module, specifying the inputs and outputs, and returns a JobId. The module must figure out which workers are free to accept a job, or to put them inside of a queue for executing later. Once all jobs are submitted, Wait is called until all are finished. Wait returns a non-positive integer, or a valid JobId. If a valid JobId is returned, then Makeflow checks that output the files also exist. If not, then if retries were enabled, the job will either simply fail, or be re-submitted. Once all JobIds have been returned, Makeflow cleans up and quits, as all tasks have been done. Kill is called when the user kills Makeflow.

Work Queue is an example of a tool which implements Makeflow's sandbox model. Users can start Work Queue workers on any computer, specifying how much of the computer's resources it can use, and which project to accept jobs from. A master computer runs Makeflow, and handles sending jobs to the workers. The workers obtain jobs from the master, and returns the outputs from the jobs. The workers will attempt to use all of the resources which the user has specified as available. The principal caveat is that all workers must be able to contact the master to obtain the files, thus users need an outward facing computer to run Makeflow on.

Amazon's EC2 is a machine instance service, giving users the ability to purchase virtual servers meeting their specific needs and desires, such as how many cores, how much memory, attach disk space, and customize the operating system. Amazon S3 is a cloud storage service, giving users a convenient location to place their files for applications, scaling up with their needs. Finally, Amazon Batch is a new service designed at enabling batch workloads. Users can create computation environments, customizing things such as the VPC, security groups, number of cores, etc. Users then create jobs to run, and queues to place those jobs into. Jobs can be given dependencies, priority, and other management features. The compute environments are EC2 instances, either maintained by Amazon, or created and maintained by the user.

## 3 ARCHITECTURE

To evaluate different approaches of using Amazon services, we created three different systems which integrate into Makeflow: Amazon EC2, Work Queue running on EC2, and Amazon Batch. When integrating these different approaches into Makeflow, there were three key issues that need to be addressed. The first is how to actually run the jobs. Jobs have specific requirements in terms of CPU, memory, and disk usage. Placing these jobs and being able to start and stop them is a non-trivial task. The second is delivering files to and from jobs. Jobs often have input files and then create output files that are either end products or the inputs for other jobs. Thus each system needs a way of sending files to each job. Finally, Makeflow needs a way of monitoring jobs as they run, determining if they are done, and if they are, retrieving any files they produced. By having an ability to do all three of these main ideas, modules can then easily interact with the Submit, Wait, and Kill methods specified by Makeflow. The EC2 workers and WQ approach were already established approaches in Makeflow as the batch integration was being developed. We decided to look at how these systems implemented their integration and took inspiration from them due to their similarity with Batch's design.

**EC2 + Batch** Amazon Batch is a service which works by first requesting some basic configuration details from the user, such as a minimum, desired, and maximum number of cpus. It then provides queues which are associated with environments, and finally users submit jobs to those queues, specifying which environments are acceptable. In this way, Batch keeps several EC2 instances around and then builds on top of these pre-existing sites and a job-specified container to then actually run the job. The environment thus is able to persist for many different workflows or batches of jobs, amortizing the expensive operation of launching EC2 instances over the lifetime of the workflows and batches.

The main idea behind our integration of Amazon Batch into Makeflow, is to leverage this persistence, having each job in a container, and using S3 as a remote bucket, to create a sandbox-like environment which fits our Makeflow model. To do this, the module requires a configuration script from the user providing the execution environment, the job queue, VPC, etc, along with the S3 bucket for cloud data storage. Then, the user provides this script to Makeflow, along with an Amazon Elastic Container Service image. Makeflow uses the S3 bucket as cloud storage, and the environment to run jobs. The user can use this script as many times as they wish with Makeflow, before deleting their cloud resources. To help users,

```
CORES=10
MEMORY=10000
DISK=20000

output1.dat: program input1.dat
        program -i input1.dat -o output1.dat

output2.dat: program input2.dat
        program -i input2.dat -o output2.dat

output3.dat: program input3.dat
        program -i input3.dat -o output3.dat

output_final.dat: program output1.dat output2.dat output3.dat
        program -i output1.dat,output2.dat,output3.dat -o output_final.dat
```

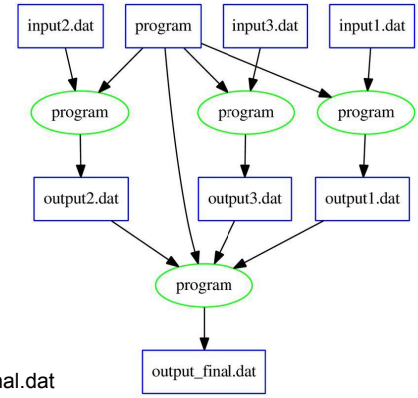



**Figure 1: Makeflow File and Resulting DAG**

*In this figure, we have an example Makeflow script which lists how many Cores are needed for each job, amount of RAM, and how much Disk space will be used. We then see that each output is mapped to some inputs, and the command to run which relates all of these together. The last command will not be ran until the other three are created, since in its input parameters, it lists the outputs of the others, just as one does in make.*

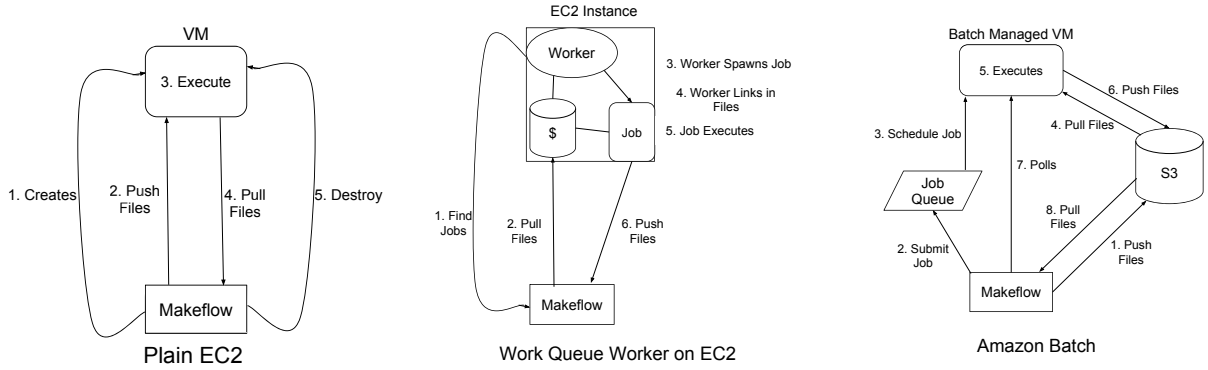



**Figure 2: An Overview of the Three Architectures**

we created a bash script which creates this configuration file, and a bash script for cleaning up cloud resources.

For each job, Makeflow uploads the necessary input files to S3, keeping track of which files have been sent, and doesn't re-upload. It then creates a shell script file for the Amazon batch job to run. The shell script instructs the job to pull down the input files from S3, run the job Makeflow submitted, then push the resulting files back to S3. Makeflow then uploads this command file, and creates a job definition. Once the job definition is created, the job is submitted, and Makeflow waits for jobs to complete. When polling for completed jobs, Makeflow loops through its internal list of submitted jobs, and sees if one has succeeded or failed. If any have succeeded, then the resulting files are pulled down from S3, and the job is labeled success, allowing Makeflow to submit a new job. If the job has failed, Makeflow attempts to re-run the job, if user specified when starting Makeflow. When Amazon Batch runs a job, it goes through the queue to see if any are waiting. If some are, it uses their priority to determine who goes first. In our module, we use the same priority for all tasks. After selecting a job, it then sees if there is any space in the compute environment to run. If there is, it is ran on the compute environment. For the Compute Environment, Amazon Batch only asks that users request the minimum, desired, and maximum number of CPUs to run for its hardware requirements. Everything else, e.g. scaling up and scaling down the back-end machines due to workload, is done by Amazon.

**EC2 + SSH** For this method, we create a brand new instance for every job, send it files, execute, and then end the instance when the job finishes. This method can be thought of as an EC2+ssh method. The reason we wanted to examine this method when comparing it to EC2, is to compare the persistent-EC2 instances of Batch which might not always have a perfect packing size for the jobs which require running, to having customized vms, specifically sized for each job. In our implementation, before being able to take advantage of EC2 instances running jobs, the user must provide a configuration file to Makeflow. This file contains information for setting up an Amazon EC2 instance. We created a bash script which

automatically sets up these services and creates the configuration file.

When Makeflow decides to send a job, it will first try to start an instance. If the user has specified a limited number of external workers at a time, it will first check to see if there is room left, or wait until there is. The module then reads the machine requirements: number of cores, disk space, and memory, and attempts to best match that to an Amazon machine type. Once it has selected that, it starts to spin up the instance. Once Amazon reports that the instance has finished initializing, Makeflow then pushes over all necessary input files including the binary via SCP into the instance. When all of the input files have been transferred over, Makeflow establishes an SSH connection and runs the job. After the job has been completed, either succeeding or failing, Makeflow will then use SCP to pull any output files back from the instance machines, and then kill the instance machine. Once the instance has been deleted, Makeflow is free to submit another job.

**EC2 + Work Queue** As an improvement, instead of launching an instance for every job, we take advantage of an extant technology, Work Queue, and have it manage accepting and running jobs from Makeflow. Both this method and Amazon Batch rely on creating persistent instances which can be reused for many different jobs, thus amortizing their startup and teardown costs over not only a single workflow, but as many as are desired. The main difference is comparing WorkQueue which is tightly integrated into Makeflow and specifically designed to take advantage of many different kinds of computational resources and leverage them to run a workflow, vs combining together different Amazon technologies to accomplish the similar goal of a custom cloud batch system.

To use this method, Makeflow requires that the user sets up the workers they wish to use, and install Work Queue on it. Then the user can run Makeflow, giving a project name. When the workers are pointed to it, they accept a job from Makeflow, receive the input files, and create sandboxes for each job. The files are linked into the sandbox, and the job runs. Once the job finishes, the worker will send the output files back to the master, and keep the input files. These input files are cached for later use, if future jobs require the same, eliminating sending every input file for every job. But the worker does need access to the master computer to work. Once Makeflow completes, the workers shutdown.

## 4 EVALUATING WORKFLOW SYSTEMS

To properly analyze the capability of a system to handle a workflow, and how well it performs, we identified five features of a workflow system to measure: First, the total Workflow Runtime, second, the Distribution of Job Runtimes, third, the job dispatch latency, fourth, the queue delay, and fifth, the job retrieval latency.

**Workflow Turnaround Time**: The workflow runtime relies on many factors, but it can give us a very general overview of how well the systems perform different kinds of jobs. For example, just because one system is very good at long computation times, does not mean it can handle workflows that have large input files. By scaling workflows of differing types, we can measure the overall effectiveness of a system for those workflows.

**Distribution of Job Runtimes**: Because each system is using cloud resources, it is hard to know exactly how far away from the hardware the job runs. A Histogram can reveal the per-core overhead of each setup. When starting an EC2 instance, the distance between the provided server and the actual hardware is unknown to the user. Amazon lists it as a virtual computing environment, thus there could be overhead when running a job[5]. Running Work Queue on an EC2 instance is somewhere between the Batch and Pure EC2 system. When running as EC2 instance, the job is ran via a remote SSH command, where as WorkQueue calls the command directly. Finally, Amazon Batch runs at least two layers away from the hardware. The first layer is the actual server itself. While Amazon makes promises about the resources available for the server, they could be either a container or a vm resting on top of actual hardware. From there, to help customize job execution, each job must specify a Docker image, generally from Amazon's container service. Then, each job is ran inside of it.

**Job Dispatch Latency**: While Makeflow is responsible for scheduling jobs, each of the three different systems handles the submission of jobs in a very different manner. When using the EC2 module, it will create a new EC2 instance, upload all files on it, and run the job. WQ's approach is to run workers on the pre-established machines, and retrieve a job from the master. Makeflow is the master; when it needs to submit a job, it broadcasts to the module that a job is ready for work. Amazon Batch has a two step process. First, a job definition is created; it bundles together the necessary information for the batch system to determine when and where a job can be ran, e.g number of processors, amount of memory, and environment variables, along with an array representing the parsed commandline string of a job to run. After the job definition is created, a job can be submitted, calling on the job definition to run. Amazon batch will then determine which compute environments can handle running a container for this job, and if space is free, run the job.

*Upload of Files*: When working with Amazon EC2, Makeflow transfers all of the necessary input files from the local host, and sends them to the remote EC2 instance. Because a job has its own unique EC2 instance, each file needs to be sent to each instance. Thus each job has its own unique copy of the data. There's no need to keep track of what files have been sent before, as all files need to be sent. However this approach has the immediate downside of creating a lot of WAN traffic, from re-sending the same files.

When using Work Queue running on EC2 workers, Makeflow and Work Queue coordinate sending files. Workers maintain their own cache of files on their machine, linking in input files for a job after setting up the job's sandbox. This allows Makeflow to only send files once to each worker. Files are only re-transmitted over the WAN to each running instance. Once all the instances with workers have a copy, the files never need to be re-sent again. For jobs which share input files, this can drastically reduce the amount of data that needs to be sent over the WAN.

For Amazon Batch, our approach uses an S3 bucket as a cloud-level cache for all files. When Makeflow attempts to send an input file to the S3 bucket for a job, it checks to see if the file has already been sent. Included in the input files is the command file. This command file is a shell script which will pull down the necessary files for the job, run the job, then upload the newly created files. Amazon Batch's approach only ever sends input files from the Makeflow-master once. However, an additional file is created for every job: the command file. But this file is small. For most workflows, this

small file will pale in comparison to most input files. In the cloud, it does produce more stress on the internal LAN of the network since every job is pulling those files down from S3 into the local container, and not caching once on the machine. Additionally, an intermediate file, one which is produced by a rule and consumed by another, will be moved twice: once to the master, and once to S3.

**Job Retrieval Latency**: Each system has a different method of reporting back when a job is done, and how quickly this can be accomplished. In both of Amazon Batch and EC2 workers cases, the master needs to poll and retrieve a finished status from the jobs, where as WQ has each job report back when it is done. Thus, the speed at which jobs are retrieved and new jobs pushed are linked.

**Queue Delay**: It is Makeflow's responsibility to order jobs: it won't schedule jobs which depend on others before the first are finished. When working with EC2 instances, we define the Queue delay as the time between Makeflow calling setting up the instance, pushing the files to it, and then actually running the test. Amazon EC2+WQ relies on Makeflow for job scheduling. Work Queue simply asks which jobs are available for work, accepts them, and returns the files when done, and doesn't schedule them. The queue delay here is thus a measurement of the time between Makeflow dispatching the job and Work Queue worker claiming the job, receiving the files, and starting the task. While Amazon Batch is capable of determining job order via dependencies, our system does not employ this usage. Submitting a job is simply telling the batch system to attempt to schedule a job for execution. When Amazon Batch has a selection of jobs to run, it will try and match the specified CPU and RAM requirements to some managed instance backing up the compute environment, then runs the job in a Docker container. The difference between Amazon recognizing the job as being created and when the container running the job is started is our definition for the queue delay of Amazon Batch. Note, our measurement doesn't include the uploading of files to S3, or the pulling of files from S3 to the container. These are accounted for in the total runtime, however.

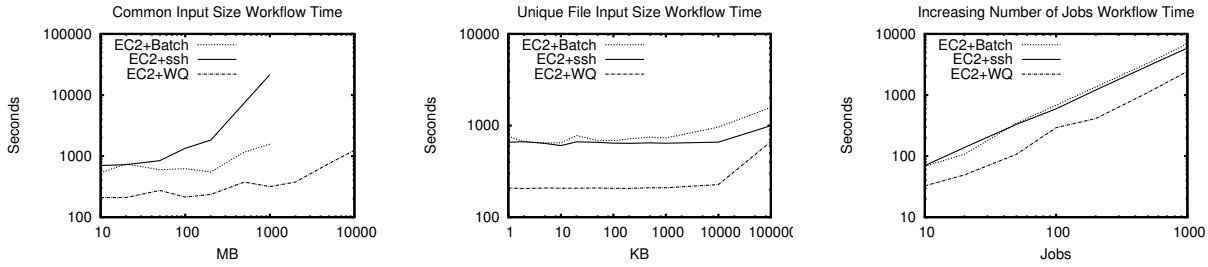
## 5 EVALUATION

To properly test our system, we created a benchmark generator which creates custom Makeflows. We can choose how many jobs are in a Makeflow, how large the unique file for each job is, how long each job will busy wait, and how large the common input file given to every job is. This allows users to test their own workflow systems, to see how it will perform for their kinds of workflows.

For each of our tests, we tried to standardize the underlying systems for each module. When running Work Queue on top of Amazon EC2 instances, we used a total of 10 cores across two T series instances. Both of the instances then ran Work Queue worker, listing all of their cores, memory, and available disk space as available for use. When running Amazon batch, we allowed Amazon to decide which compute systems would be optimal, requesting only 10 cores for the minimum, maximum, and desired cases. When using an entire instance to run a task, for our plain EC2+ssh model, we limited the number of workers to 10 as was our account limit. In each case, we informed Makeflow that the job only needed 1 core.

We ran five increasingly larger kinds of workflows. The first series of tests determined how the different systems behave as the common input for each job grows larger, from 10MB-10GB. The tasks themselves open the common input file, dump 4MB of random data from /dev/urandom to /dev/null, and busy wait for 20 seconds, replicating processing some data, or performing some calculation. For each file size, we ran 100 jobs. The second series determined how the systems behave as the size of the unique input files grows larger, from 1KB-100MB. The tasks themselves would open the input, dump the entire content of the file to /dev/null, and busy wait for 20 seconds. For every file size, we ran 100 jobs. The third series determined how the systems behaved when increasing the number of jobs to run, from 10 to 1000. Each job would dump 4MB of data from /dev/urandom, and busy wait for 20 seconds. The fourth test was to determine the raw dispatch and retrieval rate of the systems. For this test, the jobs did not perform any I/O, nor did they busy wait. The final test was to see how the systems performed running a real workflow. We ran the BWA-GATK [6] workflow, with private data. For the first four tests, our Amazon Batch image was a small image built on Python2.7:latest image from Docker-hub, and installed AWS CLI on it. For the last one, we used an image based on amazonlinux:latest, and installed unzip, AWS CLI, java, and perl.

**Workflow Runtime** In figure 3, we see that each system performs differently for each type of workflow. When the workflow is primarily composed of a single large common input file, a straight instance per job method became exponentially worse and worse as the size of the input file increased. This is expected, as the file needs to be transferred over WAN for every job. Work Queue running on pre-fetched instances had the best performance, due to the caching ability of the workers. Amazon Batch performed between the two, due to how data is transferred to the workers. First, the data is sent to S3, and then downloaded to each worker. This makes it much cheaper for the user, as they minimize the amount of data that needs to be transferred over the WAN. Amazon Batch did suffer from failures with bigger input sizes, with some tasks failing due to a known issue with not enough blocks for the ThinPool. Future work will address this, however, we believe that the data gathered is a good indicator for performance as compared to the other systems. Similarly, we did not run EC2 test on the large sizes, as an obvious trend is in the graph. For EC2+ssh test we also skipped data point 500, due to loosing that run log but we had the 1000MB data point, which showed the trend. Each system had similar performance for growing unique input files for each job. Again Work Queue was best, most likely due to the lower overhead cost of running tasks. Where the other systems need to set up their environments, Work Queue is already running, allowing for faster run-times. When the number of jobs grew, the runtime for each also grew, with Batch and EC2 systems growing at a much larger rate than Work Queue, likely due to the overhead of both Systems. When running the data-scaling jobs, the specifications were the same across all jobs, with a minimum of 1000 designated as the diskspace, unless the amount of data needed to be increased, e.g for common data, when the input was 10GB, the diskspace was designated as 12GB. For the job-scaling run, while the number of cores needed was always specified as 1, the amount of memory or disk was left out for the 10-jobs run, and sometimes specified as 1000 or 1. Ultimately, since the jobs themselves are tiny and do not create files, this should have no effect on the ultimate runtime.




*n=100 jobs, t=20s, c=0MB* *n=100 jobs, t=20s, u=0KB* *t=20s, c=0MB, u=0KB*

**Figure 3: Total Workflow Comparions between Different Tasks and Systems**


*n=1000 jobs, t=20s, c=0MB, u=0KB*

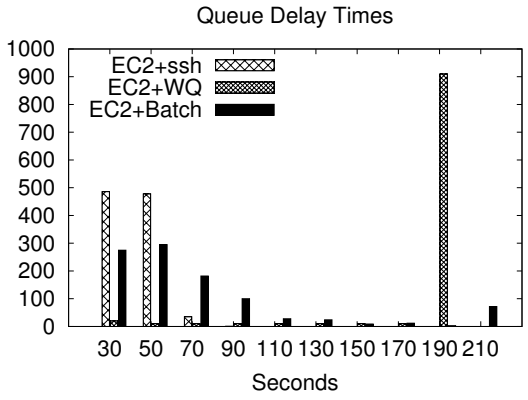
**Figure 4: Histographic Runtimes of Different Systems**
*Here, the times have been binned into intervals, with all extrema in the graph binned into the first and last bins, respectively.*

**Histographic Runtime of Jobs** To examine the overhead of the system, we took the data from the 1000 job, busy wait test, and a re-run of the EC2+WQ run of this job size. Each job performed 4MB of I/O and busy waited for 20 seconds, up to the resolution of the *time*() function in the C standard library. The primary differences between the Batch, EC2, and EC2-WQ tasks is the "queue" size that Makeflow understands. For Batch, it was set to 100 by default, for EC2 it was set to 10 to limit the number of workers to 10, and for EC2+WQ, it was set to 100 as well, while the default is 1000.

Each system doesn't add much overhead to the jobs being ran. Each job should run for roughly 20 seconds, and as seen in figure 4, both Batch and Work Queue have similar run times. The runtime of a single job in Amazon Batch is defined as the difference between when Amazon reports the job container started and ended. The most extreme data point is a single job which ran for between 26 and 27 seconds, but the vast majority were between 21 and 24 seconds. For Work Queue, it's defined as the time when a job is between the running and waiting retrieval states. All of the jobs ran between 19 and 21 seconds, as expected. For EC2, which has the widest spread of jobs, it's defined as the time between the job is started and when the module polls that the job is done. This helps explain why there is a wide spread of reported execution times in the histogram. There were only two erroneous jobs in the job which ran for less than 19 seconds, and two jobs which failed but when re-ran, ran correctly and are captured in the histogram.

| System | Dispatch Rate | Retrieval Rate |
|---|---|---|
| Batch | 0.276 tasks/second | 0.813 tasks/second |
| EC2 | 0.528 tasks/second | 0.260 tasks/second |
| Work Queue | 36.452 tasks/second | 36.370 tasks/second |

**Figure 5: Job Dispatch and Return Rate of the Three Systems**




*n=1000 jobs, t=20s, c=0MB, u=0KB*

**Figure 6: Queue Delay Histograms**
*Here, the times have been binned into intervals, with all extrema in the graph binned into the first and last bins, respectively.*

**Job Dispatch/Retrieval Latency and General Queue Delay** One of the most important characteristics of a system is the raw ability to dispatch and retrieve jobs from our Makeflow to the underlying batch system. To measure this, we created a workflow that delivered a binary, but did no work. Then we examined how long it took Makeflow to send off all of the jobs in its queue, and once there were no more jobs to dispatch, how long it took to recover the remaining jobs. By default, Work Queue has a queue of 1000, Batch has a queue of 100, and EC2 had a queue of 10, limiting the number of instances to only 10.

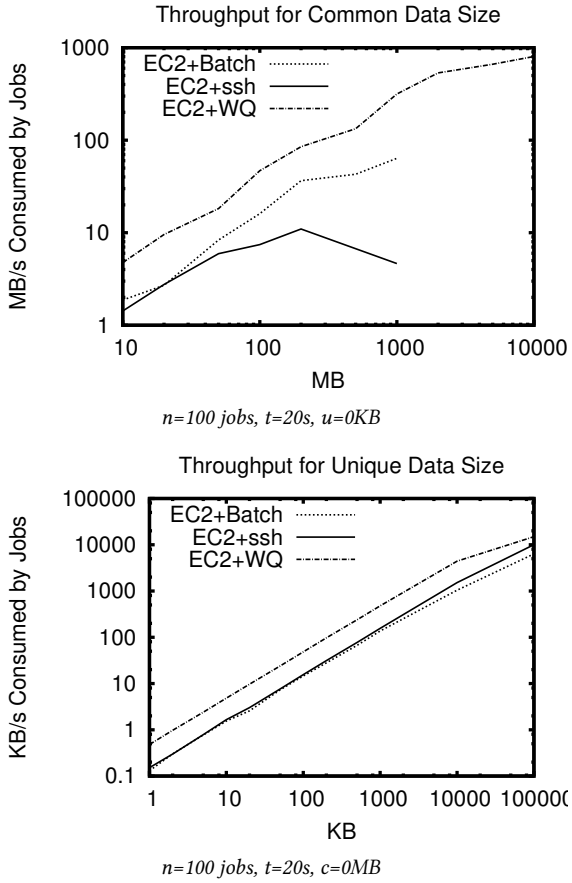



**Figure 7: Ability to Deliver Data To Each Task**

As seen in the table in figure 5, Work Queue is able to dispatch and retrieve completed jobs the fastest, due to simply transferring responsibility to a job already in memory on the same computer. EC2 has good performance as well, since the module would start calling a series of AWS API calls, creating the machine directly. It did end up recovering more slowly, due the intricacies of going through the full motion of setting up an instances. Finally Batch had decent performance, despite the several tasks it needs to perform in going out to submit the jobs and then poll for jobs to finish.

For the internal queuing rate, we took the same data as from the histogram set. Amazon batch had the largest spread in its queuing rate, as when jobs are ran is determined by Amazon. Most of the jobs ran within 30 to 90 seconds, with a large number starting in less than 30 seconds. EC2 had the shortest queue times of all three systems, likely due to the limiting of the queue size due to the number of allowed workers. The primary delay came from setting up the instance and polling to see if it is ready for execution, then sending the worker and execute files. Work Queue had a rather consistent delay of 190 seconds, the longest out of all of the systems. This is due to the fact that as a job is returned, the task is placed immediately at the back of the queue, with 100 jobs in front of it.

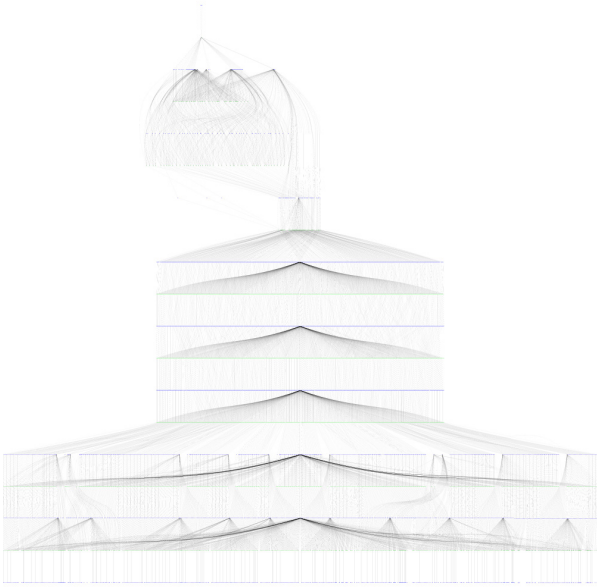


**Figure 8: Visualization of Similar BWA-GATK Workflow**

*Upload of Files* One of the most expensive tasks for each system is delivering files to each job. Because each job needs to have data sent to it, we can look at how many Bytes per second are consumed by each job over the runtime of the job. Because there were 100 jobs, we measure this by taking 100, multiplying by the size of the data file required by each job, and divided by the total runtime of the workflow.

For increasing Common data size, we see that Work Queue was the fastest in being able to deliver data to the individual jobs, since once the file was sent, it was cached on the workers. Batch also had good performance, as data only had to be transferred over WAN once, letting Amazon's internal LAN handle delivery to each task. Finally, EC2 had the worst performance, as data was only ever delivered over WAN. This trend is seen again for the unique data sizes, where Work Queue is fastest, followed by both Batch and EC2. Work Queue is likely fastest due to runtimes being faster, since the data is sent before the job is considered "started" as compared to the other two system. Both Batch and EC2 had very similar performance, with EC2 starting to be just a hair faster than Batch, probably due to files not needing to go through S3 before getting to their tasks.

**Real-Data Workflow Test** For this test, we ran a workflow which parallelizes the Burroughs-Wheeler Alignment and Genome Analysis Toolkit (BWA-GATK). We used private data as the input data with 10 splits, and ran it in two configurations. In the first configuration, we used Batch requesting 10 cores as desired, min, and max number of cores, and allowing Amazon to handle selecting the types of EC2 instances to use, after we set up our VPC and the other necessary steps required for creating a compute environment. In the EC2+WQ configuration, we set up a C4.large and a C4.2xlarge for a total of 10 cores. We also set universal rules for Makeflow setting Cores to 1, Memory to 3000 and Disk to 4000.

| System | Total Runtime | Data Transferred | Jobs/min |
|---|---|---|---|
| Batch | 3h 0m 26s | 48.26 GB | 2.94 |
| EC2+WQ | 3h 33m 13s | 64.34 GB | 2.49 |

**Figure 9: Batch vs WQ BWA-GATK Performance**
*Batch moved less data and ran in less time, achieving higher throughput of Jobs/min, compared to EC2+WQ, with 1.18x speed up*

As we can see in the table in figure 9, by using Batch, we outperform Work Queue. This is likely due to the fact that Batch sent less data, thanks to using S3 as a remote cache for the cloud, and not sending files more than once. Because the output of the Batch workflow is capture by Amazon Cloud Logs and isn't sent back to the Makeflow master, we redirected output of the WQ configuration to */dev/null* in order to make the two equivalent. Our Makeflow toolset comes with a monitor which can read the Makeflow logs, which is where the runtime and jobs/min figures come from. To measure amount of data transferred, for Batch our system states which files are sent or retrieved from S3, and our Makeflow file also specifies how large the files used are. For those files not specified (such as binaries, or Jar files, or a packaged version of the JRE) we simply used stat to find their size and added up the sizes. For WQ we used the figures reported by Work Queue in the Work Queue log of how much data was sent and received and added up the figures.

## 6 CONCLUSION

In our paper, we focused on comparing Amazon Batch to running WorkQueue on Amazon resources. Previous to Batch, using Amazon resources tended to need a layer such as WorkQueue to bring those resources together. When examining the performance of Batch, in our BWA-GATK workflow, it transfered less data, ran in a shorter runtime, and accomplished more jobs/min than our WQ configuration. It also had a fairly consistently low queue delay, between 30-90s in total. When taken together, we find that Batch is a viable cloud-level runtime for running scientific applications. The two major improvements we suggest would be to allow users to specify files per job, and make that a part of the job submission, as well as allowing jobs to grow in their disk usage.

## 7 RELATED WORK

D. Yuan et al in their work also perform an analysis of cloud systems, specifically exploring the cost of storing data files on the cloud, and attempt to minimize the cost of storing the intermediate files, similar to our architecture of sending the results from Amazon Batch jobs to S3 before downloading them [7]. Similarly, Suraj Pandey et al in their paper analyze using cloud systems for workflows and demonstrate a system to minimize the cost of using cloud systems, attempting to find the characteristics of the cloud system as well [8]. Y. Yang et al in their work attempt to schedule jobs in the cloud which minimizes the execution costs and meets deadlines by performing a simulation to understand cloud performance [9]. Similar to our work, analyzing the performance of using EC2 instances from Amazon, G. Wang et al in their work analyzed the performance virtualization has on computing performance specifically in EC2 [10]. Comparisons between Grid and Cloud computing, similar to the difference between using Amazon services and an HPC center, was performed by Ian Foster et al in their work [11]. Similarly, Mehrotra et al also analyze the specialized computer cluster from Amazon EC2 vs NASA's Pleiades cluster [12]. Additionally, G. Juve et al also analyzed the performance of EC2 vs HPC for scientific workflows, specifying three performance benchmarks [13]. G. Juve et al also discussed data sharing between jobs on EC2 and discussed different techniques to improve performance and sharing [14]. S. Akioka and Y. Muraoka also perform HPC benchmarks using EC2 workers in their work, as well as estimating the price for running HPC applications on EC2 [15]. One additional consideration of using Amazon Services is the possibility of using Spot Instances, that is spare resources usually set at a different price than normally requesting instances from EC2. While not always guaranteed to be available, these instances can be cheaper for users, and is an option for Amazon Batch. Work done by A. Ben-Yehuda et al can help users better save money while performing their tasks [16].

## REFERENCES


[1] M. Albrecht, P. Donnelly, P. Bui, and D. Thain, "Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids," in *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, ser. SWEET '12. New York, NY, USA: ACM, 2012, pp. 1:1–1:13. [Online]. Available: http://doi.acm.org/10.1145/2443416.2443417

[2] P. Bui, D. Rajan, B. Abdul-Wahid, J. Izaguirre, and D. Thain, "Work Queue + Python: A Framework For Scalable Scientific Ensemble Applications," in *Workshop on Python for High Performance and Scientific Computing (PyHPC) at the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (Supercomputing)* , 2011.

[3] Z. Zhang, D. S. Katz, M. Wilde, J. M. Wozniak, and I. T. Foster, "Mtc envelope: Defining the capability of large scale computers in the context of parallel scripting applications," in *HPDC'13*, 2013.

[4] D. Thain, T. Tannenbaum, and M. Livny, "Distributed Computing in Practice: The Condor Experience," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[5] Amazon, "What is amazon ec2?" http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html.

[6] (2017) Bwa-gatk workflow example. [Online]. Available: https://github.com/cooperative-computing-lab/makeflow-examples/tree/master/bwa-gatk

[7] D. Yuan, Y. Yang, X. Liu, and J. Chen, "A cost-effective strategy for intermediate data storage in scientific cloud workflow systems," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, April 2010, pp. 1–12.

[8] S. Pandey, L. Wu, S. M. Guru, and R. Buyya, "A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments," in *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, April 2010, pp. 400–407.

[9] Y. Yang, K. Liu, J. Chen, X. Liu, D. Yuan, and H. Jin, "An algorithm in swindew-c for scheduling transaction-intensive cost-constrained cloud workflows," in *2008 IEEE Fourth International Conference on eScience*, Dec 2008, pp. 374–375.

[10] G. Wang and T. S. E. Ng, "The impact of virtualization on network performance of amazon ec2 data center," in *2010 Proceedings IEEE INFOCOM*, March 2010, pp. 1–9.

[11] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," in *2008 Grid Computing Environments Workshop*, Nov 2008, pp. 1–10.

[12] P. Mehrotra, J. Djomehri, S. Heistand, R. Hood, H. Jin, A. Lazanoff, S. Saini, and R. Biswas, "Performance evaluation of amazon ec2 for nasa hpc applications," in *Proceedings of the 3rd Workshop on Scientific Cloud Computing*, ser. ScienceCloud '12. New York, NY, USA: ACM, 2012, pp. 41–50. [Online]. Available: http://doi.acm.org/10.1145/2287036.2287045

[13] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, and P. Maechling, "Scientific workflow applications on amazon ec2," in *2009 5th IEEE International Conference on E-Science Workshops*, Dec 2009, pp. 59–66.

[14] ——, "Data sharing options for scientific workflows on amazon ec2," in *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2010, pp. 1–9.

[15] S. Akioka and Y. Muraoka, "Hpc benchmarks on amazon ec2," in *2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops*, April 2010, pp. 1029–1034.

[16] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir, "Deconstructing amazon ec2 spot instance pricing," *ACM Trans. Econ. Comput.*, vol. 1, no. 3, pp. 16:1–16:20, Sep. 2013. [Online]. Available: http://doi.acm.org/10.1145/2509413.2509416