

# Building IaaS infrastructures on the AWS Cloud

Saul Pierotti

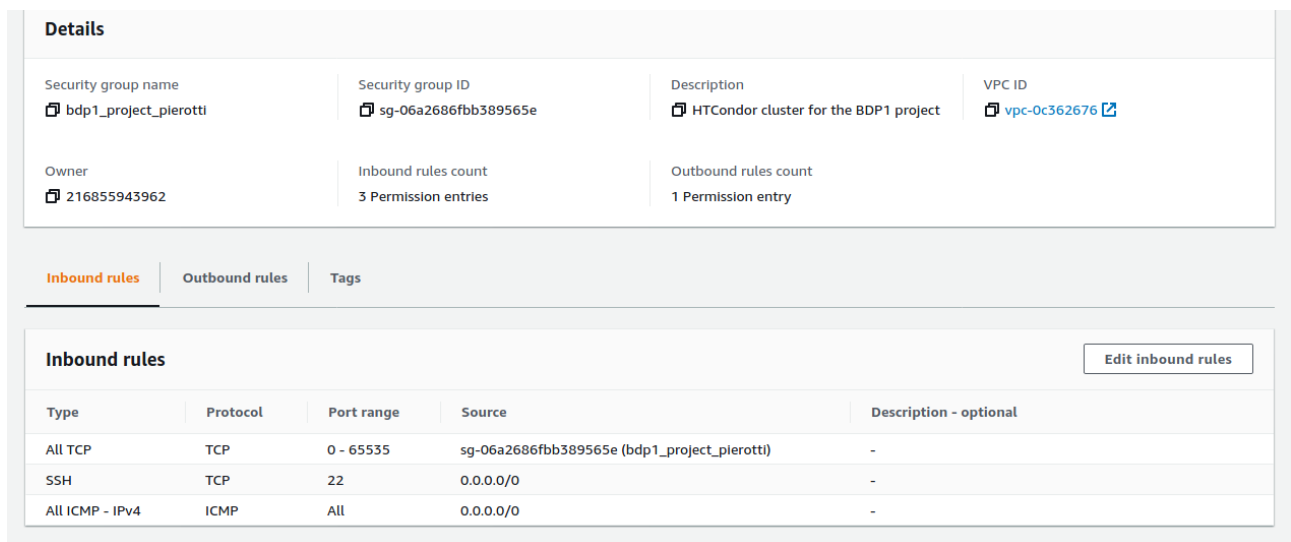
August 31, 2020

## 1 Creation of an HTCondor Cluster for the Alignment of NGS Reads

The demonstrative IaaS infrastructure described in this project consists of an HTCondor cluster of three nodes. One of them acts as Master Node and the remaining two nodes as Worker Nodes. The infrastructure is easily expandable by replicating the Worker Node instances. The Master Node was not used also as a Worker Node since the performance benefits and cost savings would be marginal, at the cost of potentially overloading the Master Node. Finally, I implemented a shared storage space using the distributed file system NFS (Sun Microsystems, 1984) directly attached to the Master Node but available to all the Worker Nodes.

### 1.1 Initialization of the Instances on the AWS Cloud

I used the cloud service provider Amazon Web Services (AWS, <https://aws.amazon.com/>) for this project. Worker Nodes and the Master Node were both built on similar machines. For the Master Node, I chose the **t2.medium** instance type with a 50 Gb SSD as root storage. For the Worker Nodes, I chose the **t2.large** instance type with a 50 Gb SSD as root storage. The operating system adopted for both machine types was Ubuntu Server 18.04.4 LTS. The Master Node and the Worker Nodes were all instantiated in the same availability zone (**us-east-1a**) so that they would be able to communicate through private IPv4 addresses. The security group for the instances was configured as follows:



The screenshot displays the AWS Management Console for a Security Group. The 'Details' tab is active, showing the following information:

- Security group name:** bdp1\_project\_pierotti
- Security group ID:** sg-06a2686fbb389565e
- Description:** HTCondor cluster for the BDP1 project
- VPC ID:** vpc-0c362676
- Owner:** 216855943962
- Inbound rules count:** 3 Permission entries
- Outbound rules count:** 1 Permission entry

Below the details, the 'Inbound rules' tab is selected, showing a table of rules:

Type	Protocol	Port range	Source	Description - optional
All TCP	TCP	0 - 65535	sg-06a2686fbb389565e (bdp1_project_pierotti)	-
SSH	TCP	22	0.0.0.0/0	-
All ICMP - IPv4	ICMP	All	0.0.0.0/0	-

An 'Edit inbound rules' button is visible in the top right corner of the inbound rules section.

All the TCP ports were opened to the other members of the same security group since HTCondor daemons use a dynamically assigned port. I opened the ICMP port for accepting incoming **ping** requests for testing purposes. Since all the TCP ports were open, there was no need for setting up additional ports for NFS.

### 1.2 Configuration of the Master Node

The PS1 prompt of the Master Node was changed so to make the node easily identifiable from the command line.

```
ubuntu@bdp1-master-node:~$ echo $PS1
\[ \e ]0; \u@ \h: \w \a \] ${debian_chroot:+($debian_chroot)} \[ \033[01;32m \]
\] \u@bdp1-master-node \[ \033[00m \]: \[ \033[01;34m \] \w \[ \033[00m \] \]$
```

HTCondor (Thain D., Tannenbaum T., and Livny M., 2005) was then installed with the following commands:

```
sudo su
wget -qO - https://research.cs.wisc.edu/htcondor/ubuntu/HTCondor-Release.gpg.key | apt-key add - #
import the gpg key of HTCondor
echo "deb http://research.cs.wisc.edu/htcondor/ubuntu/8.8/bionic bionic contrib" >> /etc/apt/sources.
list # add the repository
echo "deb-src http://research.cs.wisc.edu/htcondor/ubuntu/8.8/bionic bionic contrib" >> /etc/apt/
sources.list
apt update
apt install htcondor
systemctl start condor # start and enable the condor service
systemctl enable condor
```

The correct proceeding of the installation and the start of the condor service where checked with the following commands:

```
ubuntu@bdp1-master-node:~$ sudo systemctl status condor
● condor.service - Condor Distributed High-Throughput-Computing
   Loaded: loaded (/lib/systemd/system/condor.service; enabled; vendor preset: enabled)
   Active: active (running) since Tue 2020-06-16 10:31:25 UTC; 1min 16s ago
     Main PID: 15225 (condor_master)
    Status: "All daemons are responding"
      Tasks: 3 (limit: 32767)
     CGroup: /system.slice/condor.service
             └─15225 /usr/sbin/condor_master -f
               └─15266 condor_procd -A /var/run/condor/procd_pipe -L /var/log/condor/ProcLog -R 1000000 -S 60 -C 111
                 └─15268 condor_shared_port -f

Jun 16 10:31:25 ip-172-31-8-109 systemd[1]: Started Condor Distributed High-Throughput-Computing.
ubuntu@bdp1-master-node:~$ ps ax | grep condor
15225 ?        Ss      0:00 /usr/sbin/condor_master -f
15266 ?        S       0:00 condor_procd -A /var/run/condor/procd_pipe -L /var/log/condor/ProcLog -R 1000000 -S 60 -C 111
15268 ?        Ss      0:00 condor_shared_port -f
15286 pts/0    S+      0:00 grep --color=auto condor
ubuntu@bdp1-master-node:~$
```

The following lines where appended at the end of the main HTCondor configuration file, located at /etc/condor/condor\_config:

```
# Master Node IP
CONDOR_HOST = <Master_Node_private_IP>

# Master Node config
DAEMON_LIST = COLLECTOR, MASTER, NEGOTIATOR, SCHEDD

HOSTALLOW_READ = *
HOSTALLOW_WRITE = *
HOSTALLOW_ADMINISTRATOR = *
```

Finally, the condor service was restarted with the following command:

```
sudo systemctl restartcondor
```

I set up the NFS server on the Master Node. A new 100 Gb standard magnetic volume was created from the AWS interface and attached to the machine. A primary partiton was initialized on the volume using `fdisk` and an Ext4 file system was created onto it using `mkfs.ext4`. The `/etc/fstab` file was modified so that the machine would mount the volume automatically at boot under the newly created directory `/data`. The following line was appended to `/etc/fstab`:

```
</new_volume/partiton>          /data    ext4    defaults          0 0
```

The following commands were then issued to install the appropriate packages:

```
sudo apt install nfs-kernel-server
```

The following line was appended to the NFS configuration file `/etc/exports`:

```
/data 172.31.0.0/16(rw,sync,no_wdelay)
```

Finally, I set the owner and group of the shared folder to `nobody:nogroup` and I edited the permissions of the folder so to grant unlimited access to it:

```
sudo chown nobody:nogroup /data
sudo chmod 777 /data
```

The `/data` folder was exposed to all the Worker Nodes on the address range `172.31.0.0/16`. This configuration does not pose a significant security risk since all the machines belong to the same Virtual Private Cloud (VPC). Only machines instantiated on the same VPC could access the exposed volume. Moreover, this configuration grants immediate access to the volume to additional Worker Nodes instantiated in the same VPC. I created a mock file on the `/data` folder so to be able to recognize the volume when mounted.

```
touch /data/this_is_a_shared_NFS_volume
```

### 1.3 Configuration of the Worker Nodes

I instantiated on AWS a new `t2.large` machine with Ubuntu 18.04 LTS. I changed the `PS1` prompt so to make the node easily identifiable from the command line.

```
ubuntu@bpd1-worker-node:~$ echo $PS1
\[ \e]0;\u@\h: \w\a\]${debian_chroot:+($debian_chroot)}\[ \033[01;32m\]\u@bpd1-worker-node\[ \033[00m\]:\[ \033[01;34m\]\w\[ \033[00m\]\$
```

I installed HTCondor on this system with the same procedure used for the Master Node. Only the `/etc/condor/condor_config` file was configured differently, by appending the following lines to it:

```
# Master Node IP
CONDOR_HOST = <Master_Node_private_IP>

# Worker Node config
DAEMON_LIST = MASTER, STARTD

HOSTALLOW_READ = *
HOSTALLOW_WRITE = *
HOSTALLOW_ADMINISTRATOR = *
```

I granted to the Worker Node access to the shared NFS volume. The following command was issued to install the required packages:

```
sudo apt install nfs-common
```

A new directory was then created at `/data` using the `mkdir` command. The `/etc/fstab` file was edited by appending the following line, so that the shared volume would be automatically mounted at boot under the directory `/data`:

```
<Master_Node_private_IP>:/data /data nfs defaults 0 0
```

I verified that the shared volume was accessible from the Worker Node.

```
ubuntu@bpd1-worker-node:~$ ll /data
total 24
drwxrwxrwx  3 nobody nogroup  4096 Jun 17 07:16 ./
drwxr-xr-x 24 root    root    4096 Jun 17 06:46 ../
drwx-----  2 root    root    16384 Jun 16 17:38 lost+found/
-rw-rw-r--  1 ubuntu  ubuntu    0 Jun 17 07:16 this_is_a_shared_NFS_volume
```

I installed the BWA application on the Worker Node:

```
sudo apt install -y bwa
```

I took a snapshot of the Worker Node virtual machine (AMI) through the AWS web interface. In this way, the Worker Nodes could be easily replicable when more computational power would be needed. It would be possible to deploy new Worker Nodes by simply instantiating new virtual machines from the AMI, without the need for manual configuration.

## 1.4 Submission of a Test Job to the HTCondor Cluster

I instantiated a new Worker Node from the relative AMI. Subsequently, I created a new volume through the AWS interface from a snapshot containing test data used during the BDP1 course (`snap-09ee52d8038fb8094`, BDP1\_2020). This snapshot contained NGS reads from 3 different patients. Each patient had a folder with around 500 fasta files, with 1000 reads each. The new volume was mounted on the Master Node under the directory `/data`, replacing the empty volume used before. The `/etc/fstab` file was updated accordingly and the `nfs-server` service restarted. Finally, I tested that the new volume was accessible from the Worker Nodes.

```
ubuntu@bpd1-worker-node:~$ ll /data
total 28
drwxrwxrwx  4 root root  4096 Apr 26 13:38 ./
drwxr-xr-x 24 root root  4096 Jun 17 06:46 ../
drwxr-xr-x  5 root root  4096 Apr 19 14:39 BDP1_2020/
drwx----- 2 root root 16384 Apr 26 09:37 lost+found/
```

The test job consisted in aligning ten fasta files (`read_1.fa` to `read_10.fa` from patient 1) to the human genome build hg19, also stored on the shared volume. I used the BWA alignment tool (Li H. and Durbin R. 2009) for the scope. This tool takes advantage of indexing the genome for speeding up the alignment of low-divergent reads. The index for the hg19 build was already in the volume snapshot, so there was no need to compute it from scratch. I copied the test fasta files from the shared volume to the home folder of the Master Node to simulate a real workflow. I created test job file `alignment_test.job`, with the following content:

```
#####
##### Alignment Test #####
#####

##### The program that will be executed #####

Executable = alignment_test.py
readnum = $(Process)+1
arguments = read_${INT(readnum)}.fa

##### Input Sandbox #####

Input      = read_${INT(readnum)}.fa
transfer_input_files = read_${INT(readnum)}.fa

##### Output Sandbox #####

Log        = read_${INT(readnum)}.log
# will contain condor log

Output     = read_${INT(readnum)}.out
# will contain the standard output

Error      = read_${INT(readnum)}.error
# will contain the standard error

##### condor control variables #####

should_transfer_files = YES
when_to_transfer_output = ON_EXIT
```

```
Universe = vanilla

#####

Queue 10
```

The script `alignment_test.py` called as executable in `alignment_test.job` was the following:

```
#!/usr/bin/python
import sys,os
from timeit import default_timer as timer

start = timer()
dbpath = "/data/BDP1_2020/hg19/"
dbname = "hg19bwaidx"
queryname = sys.argv[1]
out_name = queryname[: -3]
fatile = queryname
samfile = out_name + ".sam"
gzipfile = out_name + ".sam.gz"
saifile = out_name + ".sai"
md5file = out_name + ".md5"

print "Input: ", queryname

command = "bwa aln -t 1 " + dbpath + dbname + " " + fatile + " > " + saifile
print "launching command: " , command
os.system(command)

command = "bwa samse -n 10 " + dbpath + dbname + " " + saifile + " " + fatile + " > " + samfile
print "launching command: " , command
os.system(command)

# Checksums
print "Creating md5sums"
os.system("md5sum " + samfile + " > " + md5file)

print "gzipping out text file"
command = "gzip " + samfile
print "launching command: " , command
os.system(command)

# Transfer files to shared volume and clean the Output Sandbox
print "Moving files and clearing the Output Sandbox"
os.system("mv " + gzipfile + " /data/outputs/" + gzipfile)
os.system("mv " + md5file + " /data/outputs/" + md5file)
os.system("rm " + saifile)

execution_time = timer() - start

print "Total execution time: " + str(execution_time)
print "exiting"

exit(0)
```

Ten instances of this test job were run on the cluster. The following outputs of `condor_q` and `condor_status` were recorded after submission:

```
ubuntu@bdp1-master-node:~$ condor_q
```

-- Schedd: ip-172-31-8-109.ec2.internal : <172.31.8.109:9618?... @ 06/18/20 08:03:52

OWNER	BATCH_NAME	SUBMITTED	DONE	RUN	IDLE	TOTAL	JOB_IDS
ubuntu	ID: 24	6/18 08:03	-	-	10	10	24.0-9

Total for query: 10 jobs; 0 completed, 0 removed, 10 idle, 0 running, 0 held, 0 suspended  
Total for ubuntu: 10 jobs; 0 completed, 0 removed, 10 idle, 0 running, 0 held, 0 suspended  
Total for all users: 10 jobs; 0 completed, 0 removed, 10 idle, 0 running, 0 held, 0 suspended

```
ubuntu@bdp1-master-node:~$ condor_status
```

Name	OpSys	Arch	State	Activity	LoadAv	Mem
slot1@ip-172-31-8-22.ec2.internal	LINUX	X86_64	Claimed	Busy	0.000	3979
slot2@ip-172-31-8-22.ec2.internal	LINUX	X86_64	Claimed	Busy	0.000	3979
slot1@ip-172-31-14-66.ec2.internal	LINUX	X86_64	Claimed	Busy	0.000	3979
slot2@ip-172-31-14-66.ec2.internal	LINUX	X86_64	Claimed	Busy	0.000	3979

	Total	Owner	Claimed	Unclaimed	Matched	Preempting	Backfill	Drain
X86_64/LINUX	4	0	4	0	0	0	0	0
Total	4	0	4	0	0	0	0	0

The cluster was able to complete the task. The test job produced the following output files:

```
ubuntu@bdp1-master-node:/data/outputs$ ll
```

total 408

Permissions	Count	User	Group	Size	Date	Time	File
drwxrwxrwx	2	nobody	nogroup	4096	Jun 18	09:12	./
drwxrwxrwx	5	root	root	4096	Jun 17	15:11	../
-rw-r--r--	1	nobody	nogroup	45	Jun 18	08:57	read_1.md5
-rw-r--r--	1	nobody	nogroup	31	Jun 18	08:56	read_1.sam.gz
-rw-r--r--	1	nobody	nogroup	46	Jun 18	09:07	read_10.md5
-rw-r--r--	1	nobody	nogroup	55074	Jun 18	09:07	read_10.sam.gz
-rw-r--r--	1	nobody	nogroup	45	Jun 18	08:57	read_2.md5
-rw-r--r--	1	nobody	nogroup	41129	Jun 18	08:57	read_2.sam.gz
-rw-r--r--	1	nobody	nogroup	45	Jun 18	08:57	read_3.md5
-rw-r--r--	1	nobody	nogroup	31	Jun 18	08:56	read_3.sam.gz
-rw-r--r--	1	nobody	nogroup	45	Jun 18	08:57	read_4.md5
-rw-r--r--	1	nobody	nogroup	54914	Jun 18	08:57	read_4.sam.gz
-rw-r--r--	1	nobody	nogroup	45	Jun 18	09:03	read_5.md5
-rw-r--r--	1	nobody	nogroup	77299	Jun 18	09:03	read_5.sam.gz
-rw-r--r--	1	nobody	nogroup	45	Jun 18	09:03	read_6.md5
-rw-r--r--	1	nobody	nogroup	55883	Jun 18	09:03	read_6.sam.gz
-rw-r--r--	1	nobody	nogroup	45	Jun 18	09:00	read_7.md5
-rw-r--r--	1	nobody	nogroup	31	Jun 18	08:58	read_7.sam.gz
-rw-r--r--	1	nobody	nogroup	45	Jun 18	09:00	read_8.md5
-rw-r--r--	1	nobody	nogroup	31	Jun 18	08:58	read_8.sam.gz
-rw-r--r--	1	nobody	nogroup	45	Jun 18	09:07	read_9.md5
-rw-r--r--	1	nobody	nogroup	53760	Jun 18	09:07	read_9.sam.gz

The time required to complete the task on a single fasta file ranged from 62.77 to 422.54 seconds, with an average of 227.7 seconds.

```
ubuntu@bdp1-master-node:~$ cat read_*.out |grep "Total execution time:"
Total execution time: 62.765912056
Total execution time: 417.533436775
Total execution time: 116.913583994
Total execution time: 66.0304908752
Total execution time: 115.42266202
Total execution time: 383.357795
Total execution time: 377.316680908
Total execution time: 158.487968922
Total execution time: 161.080944061
Total execution time: 422.54254508
```

## 1.5 Data Management Model

The general data management model followed was that of transferring the executables and the input fasta files using the HTCondor Input Sandbox. Those files are generally small and their transfer is not likely to overload the Master Node. Moreover, input data are likely to be uploaded dynamically to the cluster by users when new sequencing experiments are performed. This approach makes the upload of new input files to the cluster more straightforward. On the contrary, the large hg19 genome file and genome index files would be made available to the Worker Nodes through the shared NFS volume. The Output Sandbox will contain the condor log, the standard output, and the standard error. These files are really small and need to be immediately available to the submitter for inspecting the proceeding of the job. The aligned reads will instead be compressed and put in the shared volume, to avoid moving large data on the Output Sandbox.

## 2 Use of Docker Containers for the Task

Docker (<https://www.docker.com/>) is a container management system that can be used to easily ship and deploy applications on a variety of platforms. In this section, I repeated the same task already accomplished but this time encapsulating the BWA application in a purpose-built Docker container. HTCondor was still used to orchestrate the execution of jobs, taking advantage of the `docker` universe.

### 2.1 Configuration of the Nodes

I created a virtual machine from the Worker Node AMI that I developed in the previous steps. I installed Docker on it and I added the current user to the `docker` group:

```
sudo apt install docker.io
sudo usermod -aG docker $USER
```

I added also the user `condor` to the `docker` group:

```
sudo usermod -aG docker condor
```

In order to make the shared NFS `/data` volume accessible to Docker containers, the file `/etc/condor/condor_config` was modified by appending the following lines to it:

```
# Docker configs
DOCKER_VOLUMES = BDP1_DATA
DOCKER_VOLUME_DIR_BDP1_DATA = /data
DOCKER_MOUNT_VOLUMES = BDP1_DATA
```

I created a new AMI from this virtual machine to be able to replicate the Worker Node instances as required by the application. The Master Node did not need any additional configuration.



## 2.2 Creation of a Containerized version of the Application

I built a containerized version of the BWA application using **Docker**. A container image for the BWA application was already available on DockerHub ([biocontainers/bwa](#)), but for educational purposes, a new image was built from scratch. The Ubuntu Docker image was used as a base. The Docker image [saulpierotti/bwa](#) was built from the following **Dockerfile** and pushed to DockerHub:

```
FROM ubuntu
COPY ./alignment_test.py alignment_test.py
RUN chmod +x alignment_test.py
RUN apt update
RUN apt install -y bwa
RUN apt install -y python
```

The file `alignment_test.py` mentioned in the Dockerfile was the same script used for the non-containerized version of the application.

## 2.3 Execution of the Test Job

I modified the test job file in order to use the **docker** universe instead of the **vanilla** universe. I also added the name of the Docker image to be used for the task (`saulpierotti/bwa`) and the path of the application executable inside the container.

```
#####
##### Alignment Test Docker #####
#####

##### The program that will be executed #####

docker_image = saulpierotti/bwa
Executable = /alignment_test.py
readnum = $(Process)+1
arguments = read_$INT(readnum).fa

##### Input Sandbox #####

Input      = read_$INT(readnum).fa
transfer_input_files = read_$INT(readnum).fa

##### Output Sandbox #####

Log        = read_$INT(readnum).log
# will contain condor log

Output     = read_$INT(readnum).out
# will contain the standard output

Error      = read_$INT(readnum).error
# will contain the standard error

##### condor control variables #####

should_transfer_files = YES
when_to_transfer_output = ON_EXIT

Universe = docker

#####

Queue 10
```

The number of Worker Nodes in the cluster was increased to two, so to make it identical to the cluster used for the non-containerized version of the application. The job was run on the HTCondor cluster, but after some time it was put on hold due to the exceeding of memory limits (the workers were AWS `t2.large` machines with 8 Gb of RAM).



```
ubuntu@bdp1-master-node:~$ condor_q -hold -af HoldReason
Error from slot1@ip-172-31-72-142.ec2.internal: Docker job has gone over memory limit of 3979 Mb
Error from slot2@ip-172-31-67-190.ec2.internal: Docker job has gone over memory limit of 3979 Mb
Error from slot2@ip-172-31-72-142.ec2.internal: Docker job has gone over memory limit of 3979 Mb
Error from slot1@ip-172-31-72-142.ec2.internal: Docker job has gone over memory limit of 3979 Mb
Error from slot2@ip-172-31-67-190.ec2.internal: Docker job has gone over memory limit of 3979 Mb
Error from slot1@ip-172-31-67-190.ec2.internal: Docker job has gone over memory limit of 3979 Mb
Error from slot2@ip-172-31-72-142.ec2.internal: Docker job has gone over memory limit of 3979 Mb
Error from slot1@ip-172-31-67-190.ec2.internal: Docker job has gone over memory limit of 3979 Mb
```

To solve the issue, I changed the memory reservation policy for the slots in the Worker Nodes such that one slot would use all the available memory and CPU, instead of sharing evenly the resources among the two slots. This was done by appending the following line to the file `/etc/condor/condor.config`:

```
SLOT_TYPE_1 = cpus=100%, ram=100%
NUM_SLOTS_TYPE_1 = 1
```

From the Master Node, the command `condor_status` detected the new allocation policy:

```
ubuntu@bdp1-master-node:~$ condor_status
```

Name	OpSys	Arch	State	Activity	LoadAv	Mem	ActvtyTime
ip-172-31-76-128.ec2.internal	LINUX	X86_64	Unclaimed	Idle	0.000	7959	0+00:00:03
ip-172-31-78-75.ec2.internal	LINUX	X86_64	Unclaimed	Idle	0.000	7959	0+00:04:36

	Total	Owner	Claimed	Unclaimed	Matched	Preempting	Backfill	Drain
X86_64/LINUX	2	0	0	2	0	0	0	0
Total	2	0	0	2	0	0	0	0

The job ran as planned with this new configuration, producing the expected output files in 131.8 seconds on average.

```
ubuntu@bdp1-master-node:~$ cat read_*.out|grep "Total execution time:"
Total execution time: 114.857194901
Total execution time: 203.258821964
Total execution time: 114.834064007
Total execution time: 61.2280139923
Total execution time: 60.9483208656
Total execution time: 97.7416031361
Total execution time: 97.6998240948
Total execution time: 185.807624817
Total execution time: 185.478761911
Total execution time: 202.292255878
```

## 2.4 Comparison of the Containerized Application with the Native Application

The use of Docker containers can potentially offer greater flexibility in modifying the application or its configuration. With the native implementation changing the application involves creating a new AMI, terminating the Worker Node instances of the cluster, and instantiating new Worker Nodes from the new AMI. In a real-world scenario, it could be possible to specify in the job description itself the packages to be installed or modified in the Worker Nodes, but this approach is nonetheless cumbersome and could be prohibited by the administrative policies of the cluster. With Docker containers, changing the application involves only changing the name of the Docker image to be used. However, the main drawback of the containerized application is increased resource requirements. This point emerged even in my small test job, where the memory available to the slots was not sufficient for executing the task. The increased resource consumption of the containerized application can be attributed to the virtualization overhead. This increased overhead translates in either a less performant

application (fewer jobs can be run in a given amount of time) or a more expensive infrastructure (for instance I may have chosen to use an instance type with more memory). One way to partially mitigate the virtualization overhead is to include in the Docker image only the components that are strictly required by the application. Using as a base the smallest possible image is another viable solution (for instance I could have used Alpine Linux instead of Ubuntu as a base image).

The raw performances on the test job of the native and containerized application are not directly comparable, since for the native version each core of the Worker Nodes was assigned to a different slot and the available memory was split equally among them, while for the containerized version all the available resources were collected in a single slot. In the containerized version the running time of the application is thus smaller because the slots used were more performant than the ones used for the native application. It is important to note also that with the cluster configuration used for the native application two jobs can be run at the same time in each Worker Node, while with the configuration used for the containerized application only one job at a time can be run in each node.

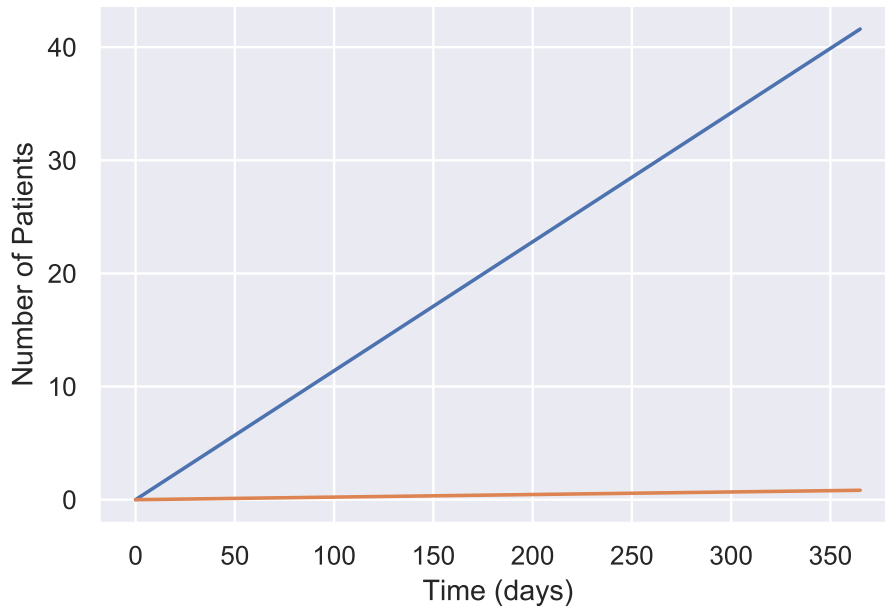
### 3 Expected Costs

A plausible non-trivial use-case that can be addressed using the HTCondor cluster developed in Section 1 (native application) is the alignment of a large number of fasta files deriving from a Next-Generation Sequencing (NGS) run to the human reference genome assembly hg19.

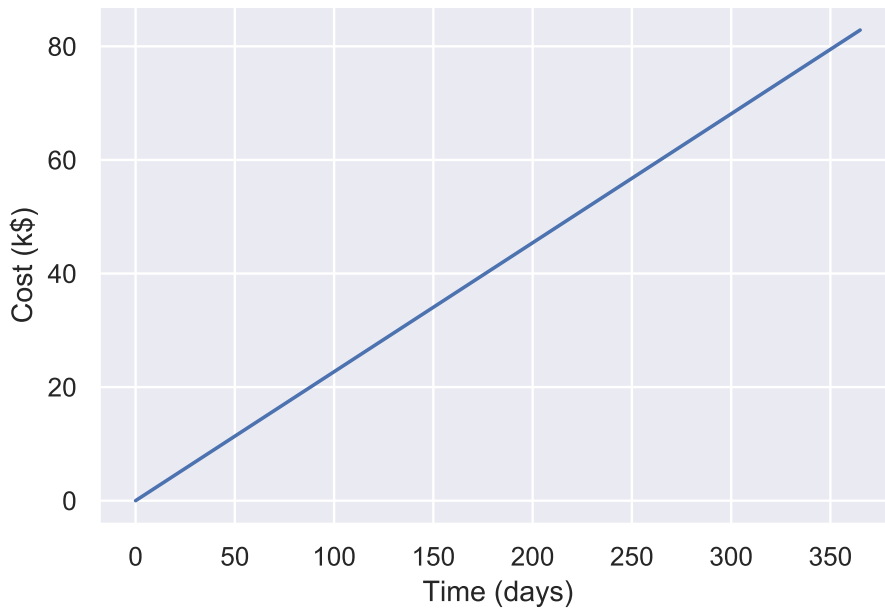
The sequencing platform manufacturer Illumina recommends aiming for a coverage of at least 30x in a human Whole-Genome Sequencing (WGS) experiment (<https://www.illumina.com/science/technology/next-generation-sequencing/plan-experiments/coverage.html>). Given that the human genome has a size of about 3.3 billion base pairs, aiming for a 30x coverage would result in a raw output from the sequencer of at least 99 billion base pairs. To be on the safe side, I considered a raw output of 100 billion base pairs per patient in the following cost estimate. A single NGS read of an Illumina MySeq sequencer can have a length of up to 300 base pairs (<https://www.illumina.com/systems/sequencing-platforms/miseq.html>). Here I considered a read length of 150 base pairs, which is recommended by the manufacturer for a WGS experiment (<https://www.illumina.com/science/technology/next-generation-sequencing/plan-experiments/read-length.html>). Under these assumptions, almost 667 million reads would need to be aligned for a single patient. This amounts to about 667000 fasta files of 1000 reads each.

I roughly determined the time needed for aligning a fasta file containing 1000 reads of 150 base pairs each to the hg19 genome assembly in Section 1. This corresponds to 227.3 seconds on a single core of the AWS `t2.large` machine. Completing the alignment of the data produced in a single WGS experiment would thus require 151609100 seconds of CPU time.

The cluster described in this project consists of one Master Node and two Worker Nodes. The Worker Nodes have two cores each, so the cluster can employ a total of four cores. In this scenario, completing the task would require 37902275 seconds, which corresponds to more than 1 year. Of course, this running time is not acceptable for a real-world application. A more feasible approach would involve replicating the Worker Nodes to have more computational power. This is possible since the alignment of a read is independent of the alignment of other reads, and so the task can be parallelized effectively. For instance, the task could be completed in less than nine days using a cluster of 1 Master Node and 100 Worker Nodes. This will result in a cluster with a capacity of more than 3 patients per month. In a real use-case, the number of Worker Nodes in the cluster would be dimensioned relative to the expected workload and to the maximum acceptable processing time. The following graph shows the number of patients that can be completely processed as a function of time in a cluster containing 2 (orange) and 100 (blue) worker nodes.



To date (May 2020) the **t2.large** machine costs \$ 0.0928 per hour, and the cluster will require 100 of them for the proposed infrastructure. The **t2.medium** machine costs \$ 0.0464 per hour, and the cluster will require just 1 of them for the Master Node. Storage on general-purpose SSD EBS devices costs \$ 0.1 per Gb per month. A 1 Tb disk can be used for storing the hg19 genome and index, as well as the reads of the patients. The following graph depicts the costs associated with running the cluster as a function of time.



Note that costs for data movements in and out of the AWS system have been omitted. Moreover, in a real use-case scenario, a strict security policy for dealing with medical data should be also enforced, according to local regulations. This will likely increase the costs and the overhead required.

## 4 Application of Concepts from the BDP2 Course to the Task

In light of the topics covered in the BDP2 course, some improvements to the architecture described in this project seem possible. For starters, it would seem reasonable to implement autoscaling of the number of worker nodes in the HTCondor cluster. Another approach could be that of using a containerized system like a Kubernetes cluster

(EKS or self-provisioned) or a Docker Swarm for running the application. Finally, an additional possible solution is to use a serverless computing system like AWS Lambda or OpenFaaS (possibly on a self-scaling Kubernetes cluster).

Instead of using a shared NFS volume mounted on the Master Node, it could be more logical to store data on an AWS S3 bucket. This would reduce the strain on the vital Master Node and make the data more easily accessible from outside of the cluster.

## 5 References

Li H. and Durbin R. (2009) Fast and accurate short read alignment with Burrows-Wheeler Transform. *Bioinformatics*, 25:1754–60. [PMID: 19451168]

Douglas Thain, Todd Tannenbaum, and Miron Livny, ‘Distributed Computing in Practice: The Condor Experience’ *Concurrency and Computation: Practice and Experience*, Vol. 17, No. 2–4, pages 323–356, February–April, 2005.

Docker:

<https://www.docker.com/>

Recommended coverage for a WGS experiment:

<https://www.illumina.com/science/technology/next-generation-sequencing/plan-experiments/coverage.html>

Read length for a WGS experiment:

<https://www.illumina.com/science/technology/next-generation-sequencing/plan-experiments/read-length.html>

<https://www.illumina.com/systems/sequencing-platforms/miseq.html>

AWS pricing:

<https://aws.amazon.com/pricing>