1 How-to/Lab : setting up an AWS server

The goals of this lab are for you to work out how to:

- Get free credits from AWS as a student
- Create and secure a server in AWS
- Start the server and ssh into it
- Download code within the server and compile it
- Manage budget and cost efficiency

There is a narrated walk-through of the first 3 steps available in panopto, which is around 25 minutes long. However, you are recommended to read the instructions as well.

There are also instructions for how to start servers Azure, if you want to play around with servers that rely on having credit.

2 Changelog

- v3 (2021/01/28) : Initial release
- v4 (2021/02/02)
  - Add note to highlight that AWS credit allocation mechanism has changed, but free tier still works
  - New information on getting Azure credits
  - New information on creating an Azure server
  - New video on starting an Azure server
- v5 (2021/02/03)
  - Note on how to deal with permissions problems on SSH keys on WSL
  - Added section numbers

3 Brief intro to cloud computing

For our purposes there are two main types of cloud computing of interest:

- *Server-based* : you gain access to a full operating-system that you control and can do anything you want with: install packages, compile programs, start services... whatever. This is almost directly equivalent to having a Linux virtual machine running on someone else's computer rather than your own, with all the pit-falls that implies - if you wreck the remote operating system, it is your job to make sure you have back-ups.

- *App-based* : you are able to write and deploy custom logic within a particular language and/or framework, such as python, go, or Java. There are usually restrictions on the types of code you can write, but there are often advantages in terms of extra services like back-up, scaling, and logging.

This document focuses on server-based nodes, with the assumption that:

- You are performing development locally in a virtual-machine.
- You want to deploy and access the same code remotely in a cloud-hosted server.

AWS (Amazon Web Services) is a commercial platform that provides a very large number of services, including cloud compute, networking, data processing, machine learning, media processing, and all kinds of services needed to build a software or data-oriented system. For the most part this document focusses just on the cloud computing side of things, i.e. the ability to run your code somewhere on a remote server in the cloud.

Note that there are a large number of cloud computing providers, many of which will provide free credits to students (or to anyone) - see the notes at the end. AWS is not necessarily the best solution, it is just free and convenient.

4 Getting AWS credits

Any student can get $100 in credits from Amazon, which can be applied to most AWS services. So across your team you have quite a large budget that can be used, though there is no expectation that you will need all of it. Much of the project can probably be done using less than $20 of credit, and if you are careful you could do it all completely on the free tier.. Note that you may want to use AWS in other scenarios (e.g. in your summer project), so it is worth trying to conserve money where possible.

Just to emphasise: you should not need to pay out of pocket for any of this. Get the free credits and use them.

4.1 Standard AWS account

Note(2021/02/01): *Amazon changed the way they distributed credits in early Jan, and want students to use AWS Educate Starter accounts. This process is now being pursued with Amazon, and (hopefully) allows for credits within a few working days. However, you should be able to follow the instructions on creating an AWS instance without any credits at all, as it all works on the free tier. Just create an AWS account, then skip to the instructions on creating an instance. Note that you want to select machine instances with the "free tier" flag, as is done in the video.*

The easiest way of accessing Amazon credits is through a "normal" AWS account, i.e. the same type you would use if you were using AWS in practise. If anyone signs up for AWS then they can get access to the free tier, and any student can get $35 of free credit. But because Imperial is an AWS educate partner, you can actually get more than this via AWS Educate - currently $100. This grant renews each year, so if you need AWS resources in 3rd or 4th year then more credits will become available.

The process for getting the full $100 is:

1. *Create an AWS account*: http://aws.amazon.com/. It will ask you for a credit or debit card, but this will not be charged if you are only using it for things on this module and make sure you start and stop instances as needed.
2. *Register with AWS Educate*: https://aws.amazon.com/education/awseducate/students/ Register with your imperial email address and select your institution as Imperial. This should allow it to recognise you as attending a partner institution.
3. *Request credits within AWS Educate*: Once logged into AWS Educate go to the link for classroom credits, and request credits. This should give you a credit code, which is an alphanumeric string. *Note: you are on AWS Educate if you are in the* `awseducate.com` *domain, and AWS proper if you are in the* `aws.amazon.com` *domain. If you can't find the link for credits, make sure you are logged into the AWS educate site.*
4. *Apply credits within AWS*: Log in to the AWS console, go to "billing" and "redeem credits". You should be able to enter your credit code, and the $100 credit will show up in your account.

From this point on you can spend your credits however you want.

5 Create a (tiny, free) Ubuntu 18.04 machine instance

You can create an instance from the "Launch Instance" button in the AWS EC2 Console.

5.1 Creating the instance

5.1.1 Step 1: Choose an Amazon Machine Image (AMI)

For AMI type choose "Ubuntu Server 18.04 LTS (HVM)", then hit "Select".

The AMI is very similar to the ISO (DVD-Image) you use to create a new virtual-machine.

5.1.2 Step 2: Choose an Instance Type

Select the free tier "t2.micro" (you could choose a more expensive one, but then you need to spend money).

Go to "Next: Configure Instance Details"

5.1.3 Step 3: Configure Instance Details

You should be able to leave them at the defaults (though it is interesting to look at all the options by hovering over the (i) buttons).

5.1.4 Step 4: Add Storage

You can leave at the defaults, but again, it is interesting to read. If you ever need to work with big-ish data then these options matter a lot, but for this module you should find that local storage on the server is sufficient.

5.1.5 Step 5: Tag Instance

We don't need this, but it is useful if you have 20 instances and you need to be able to identify which is which (Err, don't create 20 instances unless you are rich).

5.1.6 Step 6: Configure Security Group (i.e. Firewall settings)

This one is quite important, as your server will be alive on the internet, open to the world, so you need to limit access to you. We will use use one port, allowing SSH, though we will allow it to be accessible from anywhere.

1. Select "Create a new security group". (It should be auto-selected, and the defaults listed below should be correct as well).
2. Make sure the "Type" on the left is SSH (Secure Shell Server).
3. Protocol and Port Range will then be fixed to TCP and 22.
4. For Source, specify Anywhere. This is so you can login from wherever your happen to be (so could anyone else, but SSH's inherent security will stop them logging in without the key).

5. For security group name, choose something meaningful like "ssh-only". *You may wish to add more security groups later if you want to open up other ports.*

Do Next: a dialogue should pop up saying "Select an existing or new key pair."

5.1.7 Step 7: Selecting a key pair

First, do *not* proceed without a key pair. These things are important, as they are the thing that allows you to SSH into your instance.

1. Read the description of key pairs that it shows to you.
2. Select "Create a new key pair".
3. Choose a key pair name. I would suggest putting your name or login, and some hint about date. For example, I might use "dt10-key-pair-2021-01-19".
4. Download the key pair. This thing is important for as long as the instance is running, so keep it somewhere safe.
   However, you can always generate more key pairs if you lose one. If you are on a shared unix machine, change the permissions so that only you can access it:

   ```
   chmod og-rwx dt10-key-pair-2021-01-19.pem
   ```

   Note: if you find the permissions don't stick, you may wish to copy the file to a different drive. See note on key-file permissions.
5. Finish the process, and your instance will launch.

Just re-emphasising the important of key pairs: they are essentially the front-door key to your server.

If you ever accidentally put your key-pair somewhere publically accessible, then you should abandon that key-pair and create a new one. You can also just delete keys on a regular basic if you want, particularly if it is easy to recreate server images easily. It is possible (and a good idea) to protect your key-pair with a passphrase as well, or import an existing ssh key, but the details start to get more complicated.

5.2 Connecting to the instance

Use the "View Instances" button, or just go back to the AWS dashboard (it doesn't matter how you get there).

You should now be able to see an instance running in the dashboard, with a green symbol, and probably a status that says "Initialising". If you click on that row, then the bottom of the dashboard will show you details about it.

The thing we need to connect is the DNS or IP address of the instance - either will work to identify it. For example, I have previously received:

`ec2-54-201-95-131.us-west-2.compute.amazonaws.com`

as an instance, which is correspondingly at the IP address:

`54.201.95.131`

To connect to the server, you need to SSH to it.

5.2.1 Linux/Cygwin

You can ssh directly from the command line, using:

`ssh -i <path-to-your-key-pair> ubuntu@<dns-name-of-your-server>`

That should drop you into a command line on the remote server. From this point it is very similar to being logged into a virtual machine.

For example, using my key `dt10-key-pair-2021-01-19.pem` and the IP address `54.201.95.131` I would do:

`ssh -i dt10-key-pair-2021-01-19.pem ubuntu@54.201.95.131`

If you recieve an error from ssh that the keyfile is too open, you need to change the permissions using `chmod` so that only your user cam read and write it. If you find that the permissions don't seem to stick (e.g. on WSL), then see the not on keyfile permissions.

5.2.2 Windows (Putty)

There is a great ssh terminal for windows called PuTTY, which I recommend as a good windows client if you aren't already working from the console in your host. To use it, you need to convert the `.pem file` to a putty `.ppk` file:

1. Start PuTTYgen (one of the programs that comes with putty)
2. Conversions -> Import Key.
3. Select your `.pem`file and open it.
4. At this stage you can choose a passphrase using the "Key passphrase" box, which will be used to encrypt the private key. Personally I prefer to have a passphrase, as otherwise anyone who gets the key can get any of my running instances. However, you can leave it blank, and just protect the private key file well.
5. File -> Save Private Key.
6. You may get prompted about an empty passphrase, just ignore if you didn't want one.

7. Choose a .ppk file to save it as.

You can now start PuTTY itself. You might want to set this up once and save it:

1. Session: "Host Name (or IP address)": Put the DNS name of your amazon instance.
2. Connection -> SSH -> Auth: Specify the private .ppk file you just created.
3. Connection -> Data: In "Auto-login username" put "ubuntu".
4. Connection -> "Saved Sessions": Choose some name for this connection, e.g. "AWS", and hit save.
5. Hit "Open"

You should now be dropped into your remote server. If you switch to a new instance you will need to change the host settings, but the rest should stay the same. Similarly, if you change private key you'll need to modify the ppk it points at. Note that you can have multiple sessions, so you might have one set up for your own private server, and another for when you connect to a group server.

6 Managing the instance

Each instance can be in one of a few states:

1. *Pending*: once launched, the instance goes through some initial checks and is allocated to a machine.
2. *Running*: the machine is up and active, and you can SSH in and use it.
3. *Stopped*: the machine is not currently active, but the state of the hard disk is maintained, and can be run again.
4. *Terminated*: the machine is destroyed, including the hard disk.

Azure uses very similar concepts for the VM states, just with slightly different names.

Broadly speaking, you can think of the "running" state as "using electricity", and the "stopped" state as "using up a hard disk". AWS will always charge you for electricity and disk space, but generally charges more for electricity.

There is no real harm in destroying and instance at the end of each testing session and recreating it for the next one. Just remember that you will lost any files stored on the "disk" of the server, and you will need to set the server up again at the start of the next session. However, the server setup is quite easy to script...

(At this point you can jump forwards to "Setting up the instance" if you are using AWS).

7 Getting Azure credits (added 2021/02/02)

1. Go to Azure education and sign into Azure for Education.
2. Hit "Sign In". After some authentication, this leads to the "Azure Education Hub" on "portal.azure.com".
3. Go to "Redeem student credits". It should be a fairly visible button.
4. Fill in the profile information.
5. Fill in information about whether they can contact you.
6. Wait a bit (and decide if you want to give feedback on the sign-up process...). This took about 30 seconds for me.
7. You should end up back in the education hub.
8. To check you have credits, go to "Account profile", and you should find you have $100 listed under "Azure credits."

No credit card or payment method is needed.

8 Creating an Azure server (added 2021/02/02)

(Video walkthrough)

The broad steps for creating an Azure server are very similar to those for creating an AWS instance, though with some slightly different terminology.

A concept that Azure uses that

1. Go to portal.azure.com
2. Select "Create a resource", which is a big "+" button.
3. Select "Ubuntu server 18.04 LTS"
4. Under "subscription", select "Azure for Students". (This is essentially saying where the money is coming from).
5. Under "Resource Group" create a new resource group, e.g. "InfProc".
    - The Resource Group concept exists in AWS, but is more exposed in Azure. It recognises that often you want to start and stop multiple resources (servers, databases, ...) together, and so makes it easier to manage them all at once.

6. Choose a "virtual machine name" (e.g. "InfProc-server")
7. Choose a machine "Size"
    - The "B1_s" size gives you 1 CPU and 1GiB, and costs £5.66/month
    - The Azure "VM Size" is roughly equivalent to the "instance type" in AWS
    - For comparison, a "t2.micro" in AWS is ~£8.30/month
8. Select credentials in Administrator Account
    - I would suggest using SSH public key rather than password (for consistency with AWS)
    - The information about AWS public keys also applies to Azure
    - It is also worth changing "username" to "ubuntu" for consistency with AWS
9. You can leave the inbound port rules at their defaults of allowing incoming SSH
    - As with AWS, you can come back later and change firewall rules if you need more open ports
10. You can click through the remaining steps
    - An interesting Azure option is available on the "Management" page called "Auto-shutdown". If you're worried about forgetting to close servers, this is an interesting addition.
11. You should end up on a page that says "Validation passed" with a happy green tick. You can now click "Create".
12. Things will happen and pages will refresh. As with AWS, it takes time for the machine to transition into the running state.
13. If you go back to the Azure console, you should be able to see your virtual machine. Click on the machine, and then you can find the IP address.

14 From this point on it is the same as AWS (or any other remote server), and you can follow the instructions in connecting to the instance.

9 Setting up the instance

A good mental model for working with your new server instance is that it is just a virtual machine, but not on your computer. So just like you need to set up your environment in a VM, you need to setup the environment in your server. And work you do on the server will persist across power down and power up of the server, and is only list if you terminate (destroy/delete) the server.

9.1 Setting up the instance

By default, your instance has almost nothing on it. Try running g++:

```
g++ -v
```

From this point you know how to install things as necessary using `apt`.

It is a good idea to capture any setup information in a script that lives alongside your code. You can then run this script in either a new virtual machine or a new remote server and get the same set of packages.

9.2 Getting code over to your machine

Unlike a virtual machine, the remote server is

You now have a few options for getting code over to your machine:

- Copying files over via scp (file transfer via SSH).
- `cat`ing files down the SSH connection (not really recommended, but occasionally very useful).
- Pulling code over via git.

I am going to recommend getting the code via git, as it is a nice way of doing things, and makes it easier to bring any patches you make in the test environment back out to github. If you make modifications on the remote server, don't forget to "push" any changes back into github, and then (if necessary) to "pull" changes back down to your normal working repository.

The main sticking point with SSH is authentication, as your AWS instance will be able to communicate with github, but doesn't have access to your keys.

9.2.1 git + HTTPS

You can use `https` to move code backwards and forwards over git, but this requires you to type/paste in your password each time you push or pull. It is simpler in the short term, but wastes some time long-term. The better solution is to use SSH.

9.2.2 git + SSH

You could transfer your SSH keys over and use `ssh-agent` remotely, but it is better to keep your keys where you control them, using a method called SSH agent forwarding.

First, make sure you are currently authenticated with github, by doing:

```
ssh git@github.com
```

or the equivalent in PuTTY. If you receive something like `Permission denied (publickey).`, then you haven't got an agent set up - github has some instructions on how to do this. Start `ssh-agent` or `pageant`, load your github SSH keys in (these are distinct from your AWS keypair), then try again. Hopefully eventually you will see something like:

```
Hi m8pple! You've successfully authenticated, but GitHub does not provide shell access.
```

This shows that you successfully have agent authentication working on your local machine. We can now use authentication forwarding (`-A`) to allow the remote server to access your local authentication agent:

```
ssh -A -i <path-to-your-key-pair> ubuntu@<dns-name-of-your-server>
```

You should end up on the remote server again, and if you do the following **within the SSH session on the remote server**:

```
ssh git@github.com
```

You should see that you are authenticated with github on the other machine.

You can now issue git clone command to get your repository remotely, then do commit/push/pull as normal.

9.3 Editing code on the remote instance

I would not recommend doing much editing on the remote instances, it should be more for testing, debugging, tuning, and experimentation. But inevitably you will need to change some source files, and need some way of editing the files remotely (you don't want to be pulling for each edit). There is a command line editor called nano installed on pretty much all unix machines which you can use to make small changes. For example, to edit the file `wibble.cpp`, just do:

```
nano wibble.cpp
```

You'll end up in an editor which behaves as you would expect. Along the bottom are a number of keyboard short-cuts, with `^` representing control. The main ones you'll want are:

- `^X` (ctrl+x) : Quit the editor (it will prompt to save).
- `^O` (ctrl+o) : Write the file out without changing it.
- `^G` (ctrl+g) : Built-in documentation.

Other text editors can be used or installed (emacs, vim, ...), but I would suggest nano for most tasks you will encounter here, unless you have already decided on a preferred command-line editor.

It is possible to get quite a good remote editing experience with Visual Studio and SSH. If that works for you, then go for it, but note that:

- It lures you in to doing actual development in a "production" machine, when really that should happen locally.
- VS code needs to run a relatively heavy-weight server-side component, which pushes up the memory and CPU requirements for the remote server.

9.4 Running code on the remote server

To run code on the remote server, you just... run the compiled program. It will run just as if you ran a program that you compiled in your virtual machine.

If you have long-running persistent servers, then you can configure them to run as a Daemon, which is a type of persistent service. However, it is often much easier to simply run the service from the console - you can always open another SSH connection to run other commands. The advantages are:

- It is a lot more difficult to "forget" that a server is running.
- You are less likely to get confused if two servers try to bind to the same ports.
- Logs from the server can easily be viewed in real-time just by having your server print to `stderr`.
- You don't need to worry about how to configure or setup a persistent daemon (which is relatively complex).

9.5 Leaving code running

Something you may wish to do is to run a server that stays alive even if the SSH connection goes down. In those cases it is recommended that you use the GNU `screen` program. This lets you run a program in such a way that you can disconnect and reconnect the console (keyboard and display) to the program.

For example, you can run the editor `nano` in `screen`:

```
$ screen nano wibble.txt
```

Type some text into the nano screen (it doesn't matter what).

You can now disconnect the program from the terminal by pressing `ctrl+a` then `ctrl+d`. This should disconnect you from the screen session and drop you back into the command line. However, if you run `ps -a` (to list all processes), you will see that `nano` is still running.

You can reconnect by running:

```
$ screen -r
```

This will reconnect the terminal, and you should find yourself back in the original `nano` session, with the text you typed still on screen.

10 Suggested exercises

Once you have got used to creating servers, some suggested learning exercises are as follows:

1. Pick a git repository containing code that you have written (e.g. from 1st or 2nd year). Compile it and run it in a remote server.
2. Create a remote server, and give the key to someone else. Check that they can log into the server.
3. Run the python HTTP server on the remote host using

   ```
   $ python3 -m http.server
   ```

   This will create a web-server on the remote server, which will let you browse files in the directory. Work out how to connect to the remote web-server from your local machine.

   - *hint*: The python web-server listens on port 8000 by default. Do your remote server settings allow connections to TCP port 8000 through?
4. Create a minimal C++ TCP server, and compile and run it on the server. Create a TCP client locally that can connect to the client. Write a script that will:

   - SSH to the remote server
   - Install any needed packages
   - Download the code repository from git
   - Compile the server
   - Run the server

   Ideally this would all happen by running a single script. Note that you can probably *also* use such a script in a virtual machine.
5. Run two separate remote servers at the same time, and get them to send UDP packets to each other.

   - *hint*: you will need configure the remote server security group.
6. Run a UDP server locally, and try to get a UDP client running on a remote server in AWS to send a packet to your machine.

   - *hint*: This is almost guaranteed to require you to investigate the settings of your router, as it will interact with NAT.
   - *hint*: Your OS is likely to apply firewalling, so you might need to configure that.

- *hint*: There may be geo-political reasons why arbitrary UDP packets from remote servers can't make it back to your machine.

11 AWS Costs

AWS is a business, and so everything has a price. In cloud computing the cost efficiency in terms of opex are very clear, as you are charged based on how much of something you use, or for how long. While there are clearly still capex costs involved, such as buying the data-centres, servers and the networking equipment, these are usually smoothed out by the scale at which they operate and folded into the operating charges presented to clients.

Examples of things which cost money in AWS are:

- Compute time : measured in $/sec while your server is powered on.
- Data transfer : measured in $/GB for incoming and outgoing data.
- Data storage : measured in $/GB/month for data at rest

You are likely to find that compute costs dominate any project you are working on. The pricing for compute depends on the type of server you choose, and varies with the number of CPUs, amount of RAM, and the location of the server (i.e. which country it is in). Some examples of servers you might use are:

- t3.micro : 2xCPU, 1GiB RAM, $0.01/hour, $0.25/day
- t3.medium : 2xCPU, 4GiB RAM, $0.04/hour, $1.00/day
- t4g.2xlarge : 8xCPU, 32GiB RAM, $0.27/hour, $6.45/day

For most basic servers providing connectivity between clients the t3.micro or t2.micro is sufficient. For servers which need to maintain in-memory databases a t3.medium might be appropriate. If there is any significant computation happening on the server then you should pick an instance according to how much RAM and how much compute-power you need.

It is important to emphasise that the server compute cost only applies when the remote server is powered on. Usually there is no need to keep a server running for the entire duration of a coursework, particularly anything with significant memory resources - they can just be turned off and on as needed like you would a local virtual machine. There is also a significant security advantage to turning off servers when they are not actively being used, as it means that attackers have less time to look at it, and you are more likely to notice if it is compromised.

When the server is powered off there is still a cost according to the size of the image, but this is usually small. For example, a server's disk might take 8GiB of space, which at $0.5/GiB/month works out at $4/month/server.

The AWS free tier includes enough credits to run a t3.micro server permanently without costing anything, both in terms of compute and storage. Or you could

run a t3.medium contiuously for 1 month for $30.

The only remaining variable cost is network traffic, which is something to keep an eye on. You get 15GB/month free, with any extra costing $0.09/GB. So if you accidentally transfer 100GB of data without noticing, then it will cost $9 - however, it is difficult not to notice the transfer if you have thought about your system design and are actively managing it.

Overall the cost of AWS is something to be aware of, but not something to worry about. Just follow some basic practises:

- Do development on local virtual machines or directly in your host OS, not in the cloud.
- Use the cheapest/smallest server instances you can.
- Only turn on servers when you are actively using them.
- At the end of each development session log into the AWS dashboard, refresh, and check that the servers are stopped.
- Treat the private keys for any servers you create just like you would a physical key: a private key provides authorisation but not authentication, so anyone who possesses a copy can get into the server (just like a physical key).
- Occasionally check the billing dashboard to check where your credits are going.
- Stick to a single AWS region. The AWS Console only shows one region at once, so if you start servers in multiple regions you'll need to remember to check all the regions.

Over 7 years and about 400 students using much more expensive GPU instances ($2/hour or more), none of them has ever needed more than the free $100. In three cases (that I remember) they forgot to turn servers off, and after contacting Amazon support they were refunded.

11.1 Sharing resources within group

AWS provides quite sophisticated tools for managing group access to shared resources, but they are probably too complicated to be used in this context.

It is recommended that during your group and project work that you share server access by sharing private keys within the group. When working alone you can start your own server with your own key. During group work you can have one person start a server, then they can share the key within the group as necessary (and destroy it after, if they want).

12 Other cloud providers with free student tiers

Other providers include:

- Google cloud : Provides a wide variety of services, including server-based hosting
  - General Free Tier
- Microsoft Azure : Provides a wide variety of services, including server-based hosting. Good support for spinning up remote windows servers, if that's your thing.
  - General free services
  - Student credits ($100)
- Heroku: Mainly an app-based server, but very good at that. If you like languages like python, ruby, go, ... then this can offer a good managed choice, with a decent free tier.
  - Heroku for students

13 Random problem solving

13.1 Can't change permissions on key file

Something that might happen is that you try to set permissions on your key file, but they won't "stick". As a result, `ssh` will refuse to use the key file, because the permissions are too open. This is most likely to happen if you are using a drive in a host machine which is mapped into a virtual machine, and particularly if you use WSL. What is happening is that:

- `/mnt/c` is mapped to the windows `c:`
- The windows drive uses NTFS (i.e. windows) permissions
- When WSL maps the windows drive in, it treats all files as having `rwxrwxrwx` unix permissions (i.e. anyone can read, write, or execute the file).
- The call to `chmod` within WSL completes successfully, but in reality the permissions on the windows driver are unchanged.

The easiest way to deal with this is to move the key to a folder that is "inside" the guest operating system (i.e. within the virtual machine's hard drive image). A good directory for this would be a sub-directory of `~` (the home directory within the VM), as this is likely to be at `/home/USER` which will support native unix permissions.

So for example, if your key is currently in a location such as `/mnt/c/Projects/AWS/key.pem`, you could use it as follows:

```
$ mkdir -p ~/keys
$ mv /mnt/c/Projects/AWS/key.pem ~/keys/key.pem
$ chmod og-rwx ~/keys/key.pem
$ ssh -i ~/keys/key.pem ubuntu@IP_ADDRESS
```