CS 470: Unix/Linux Sysadmin
Spring 2025 Lab 7
public cloud with Azure and code repository with GitLab

Things from the prior labs that are required to begin this lab:

- none really, but grading this lab is done from your lab 4 Ubuntu VM

Nothing in this lab is on the critical path for upcoming labs.

co-authoring credits:

- 2020 TA Obada Alagha, who really was a co-author for the first rev of this lab, on AWS
- 2021 student Mikayel Melikyan, for porting the original 2020 AWS lab to Azure
- 2022 TA Jordan Shands-Sparks, who integrated the CA we stood up in the then-new lab 1b
- 2023 TA Dan Houston, who added a GitLab issue to the ending of this lab
- 2024 TA Max Carrillo, who fixed the Git commands towards the end of the lab

This lab is to get you familiar with the concept of public cloud, while introducing you to SSH tunneling, VPNs, and GitLab to set up a source repository.

In the summer of 2021, both Amazon Web Services and the campus IT administration ghosted us when we asked for the free educational credits we'd used in this class in 2020. It took Amazon a full year to finally explain why: they had eliminated the educational program that used to give students free credits. As such, we fired Amazon and AWS, and replaced them with Microsoft and its Azure public cloud service. With Azure, you can get $100 of free credits, with very few questions asked, simply for having an e-mail address that ends with .edu … and none of the pain of waiting for worthless bureaucrats to lift a finger.

The modern public cloud really started with Amazon, and with AWS, which began as an internal interface for Amazon to manage its vast internet commerce site behind the scenes. In order to become the proverbial "everything store" it's become today, web services had to become separated from data warehouses of product information behind them. Those data warehouses, in turn, had to be separated from the payment systems. All of these services, for all of those shopping customers, had to become fault-tolerant, expendable, cloneable, and scalable in a distributed fleet across multiple fault-tolerant geographical zones. In fact, the first public brand name for what is now known as "AWS" was "EC2," short for "elastic computing cloud," to emphasize the scalability services behind it. When one subset of services needed more computing power, memory, storage … they made an API for it.

Rapidly, with the knowledge you've developed if you didn't have it already, you can see the budding beginnings of a first public cloud offering underpinning the various needs of running the largest e-commerce site on the internet: virtualizing RAM, CPU, disk, and network, everything you need to stand up a data center. With the advent of public cloud as a service, you no longer need to stand up a data center or a computer room inside your business.

Most companies don't buy servers at all any more … they typically just outsource e-mail to Microsoft or Google, file sharing to another service if required, and start buying desktop and laptop computers. If they need servers, they'll typically engage a public cloud service and start setting up VMs. Many

public cloud customers don't even need APIs, just a cost-effective replacement for servers, and a web-based interface for setting up VMs and networks, and collecting payment is all that's needed.

Again, we'll be using Microsoft's Azure. Microsoft, like in a lot of things, was not the first to market, but didn't want to be left out of the party. In typical Microsoft fashion, Azure is a few years behind its competition, didn't get it right for the first few versions, and still has lots of catching up to do. Enough cloud history, though. Let's do some stuff.

You have limited time and "credits" available on here, while you're learning … so please don't idle on the website, especially with a VM running. The more you use, the less credits you have left. Once you're done grading at the rapidly-approaching end of class, you can use those credits on Azure to play with whatever you please.

The service we'll be standing up on our Azure VM is a code repository. The modern standard for version control is Git. Git is both a protocol and the eponymous reference implementation for the software, which was originally designed by none other than Linus Torvalds to support the development of the Linux kernel when nothing else was good enough. Git supports branching, distributed development, efficiently handles big codebases with pluggable merging strategies, HTTPS and SSH for code security in transit, and encryption to protect against tampering and accidental corruption in storage.

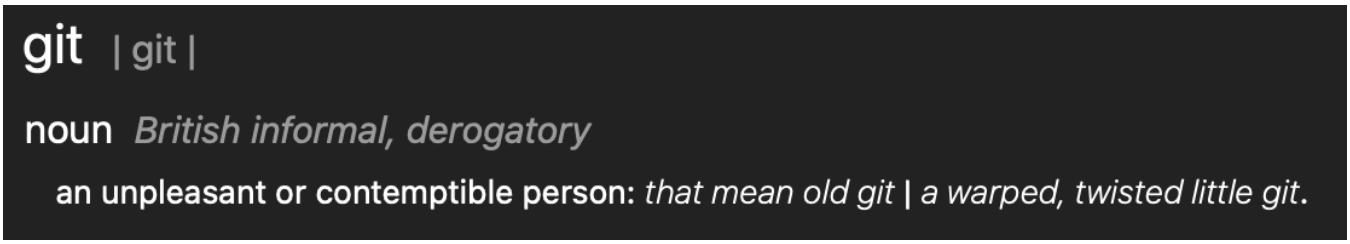
Like OpenBSD project founder Theo ReDaadt and yours truly, Linus has been historically known for strong opinions and a lack of concern for others' feelings. Here's Linus famously commenting on the working relationship the Linux kernel team has with graphics chip vendor Nvidia.

Linus has acknowledged this behavior, and even taken a hiatus from the kernel project to ponder his behavior. Here's how Gizmodo Australia put it in their article's headline …

Linux Founder Takes Some Time Off To Learn How To Stop Being An Arsehole

https://gizmodo.com.au/2018/09/linux-founder-takes-some-time-off-to-learn-how-to-stop-being-an-arsehole/

In British English, a "git" is somebody kind of like Linus …



… and he's acknowledged, "I'm an egotistical bastard, and I name all my projects after myself. First Linux, now `git`." According to the `README.md` for `git` to this day …

He described the tool as "the stupid content tracker" and the name as (depending on your mood):

- random three-letter combination that is pronounceable, and not actually used by any common UNIX command. The fact that it is a mispronunciation of "get" may or may not be relevant.
- stupid. contemptible and despicable. simple. Take your pick from the dictionary of slang.
- "global information tracker": you're in a good mood, and it actually works for you. Angels sing, and a light suddenly fills the room.
- "goddamn idiotic truckload of sh*t": when it breaks

Don't take my word for it. See https://github.com/git/git/blob/master/README.md or `man git`.

Enough back-story; let's get cracking.

part zero: get free loot

1. First, let's sign up for a free $100 of Microsoft Azure credits. Go to the following URL ...

   https://azure.microsoft.com/en-us/free/students/

   ... and click on the "start free" button. Microsoft will start interrogating you about your eligibility and your status as a student. **As such, it's really important to use your @sdsu.edu e-mail address.**

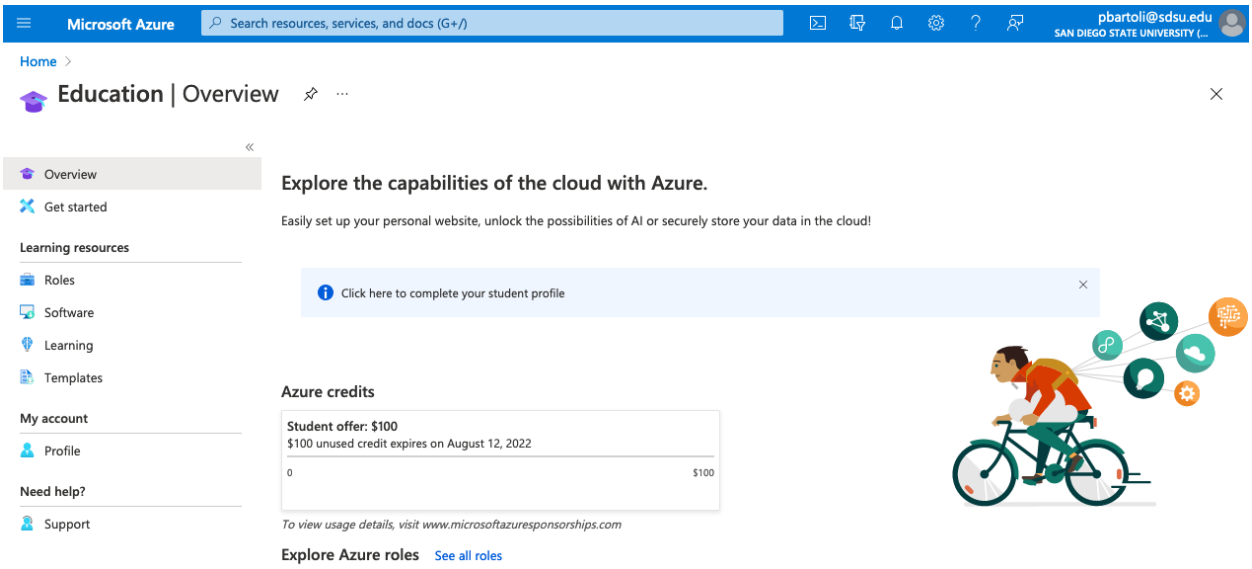
   Auditors, you can use your cssc*@edoras.sdsu.edu account as an e-mail address to sign up

here ... but before you do, make sure to create a `~/.forward` file on EDORAS with `vi` and put a real e-mail address you want your EDORAS account to forward to in that file. You can use mail on EDORAS to test that the forwarder works.

Back to Azure. Microsoft will ask you for your phone number, to further validate you, both for multi-factor authentication, and so they have somebody to call before they essentially lease you a virtual server in their data center. This is primarily intended to deter spammers, who plagued the early cloud by exploiting trial offers like this one, and using cloud trial credits to fill all our inboxes with advertisements.

Enter a valid phone number, because they will send a verification code. Yes, Microsoft is evil, but this free is still free.

Once you have been successfully verified as a student, you will be greeted by this unoffensive Windows 10-looking screen, and when you click on "Overview" in the left-hand side toolbar, you should be shown that you have $100 in Azure credits, good for a year.



2. **This step is optional**, but extremely useful to take advantage of while you're still in school and can get free loot: sign up for the GitHub Student Pack at the following URL ...

   https://education.github.com/benefits?type=student

   This will also send a link to your school e-mail (.edu) to your existing GitHub account to verify that you're a student. The GitHub Student Pack also provides another way to get at $100 of Azure credits, so it ties in with this lab.

part one: set up a VM

Generally, there's a point-and-click way to do everything inside Azure, and there's an API for everything, so you can do it with the command line, or with code, in what we now call "infrastructure as a service" (IaaS … remember our thing about as-a-service before?).

We're just going to do it the easy way, via the web interface, because we've got a lot to cover.

3. Click on the three lines menu in the upper left-hand corner of the Azure screen and select "Virtual Machines." Then click "Create" to create a new VM, then "Azure Virtual Machine."

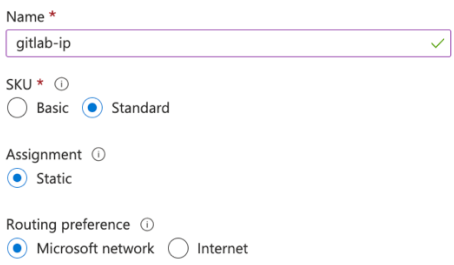
   virtual machine name: **gitlab**
   region: choose whatever is closest to you, something like US West for most of us
   availability options: no infrastructure redundancy required
   security type: standard
   image: Ubuntu Server 24.04 LTS – x64 Gen 2
   VM architecture: x64
   size: **Standard_D2s_v3 - 2 vcpus, 8 GiB memory** (the cheapest option)
   authentication type: **SSH public key**
   username: **your LDAP username used for previous labs**
   SSH public key source: **Generate new key pair**
   key pair name: **azure_gitlab**
   public inbound ports: **Allow selected ports**
   select inbound ports: **click on the dropdown, check SSH (22) and HTTPS (443)**

   Now click on "Next : Disks" at the bottom. Don't create the VM yet!

   !! IMPORTANT NOTE: your VM is billed hourly, and you **must** shut down your VM at the end of this lab to avoid draining your free credits. Don't worry, you won't be billed $70 right away. You're billed a pro-rated amount for every hour your VM is running.

4. On the "Disks" screen, the default size is fine … but select "Standard SSD." We're saving a few credits here. Leave the rest untouched. Then click on "Next : Networking."

5. On the "Networking" screen, click "Create new" under "Public IP," then in the pane that comes in from the right, select a static assignment if it isn't already. This may cost a couple more credits, but is well worth it to not have to repeatedly change DNS to be able to aim properly at it.



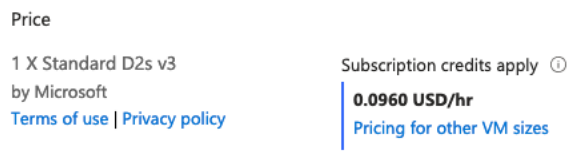
Then click on “Next : Management.”

6. Microsoft Azure has thoughtfully provided us with an option to auto-shutdown VMs to avoid draining our credits, just in case you forget. This is an important fail-safe mechanism. Turn on "**Enable auto-shutdown**," then change the **time zone** to **(UTC-08:00) Pacific Time**, or adjust accordingly to your time zone so that your VM will be awake when you are.

   Also note: make sure the shutdown time is far enough from your current time, so your VM doesn't decide to shut down in the middle of you doing this lab.

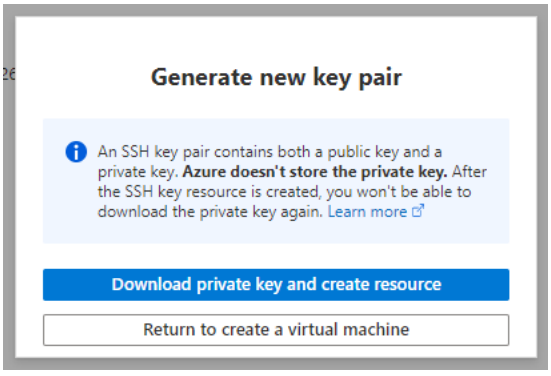
7. Click on “Next : Monitoring” then “Next : Advanced” then “Next : Tags,” and make sure to read all the options available after each click, but leave everything untouched.

8. Click on “Next : Review + create,” and pay attention to pricing. This is what I got ...

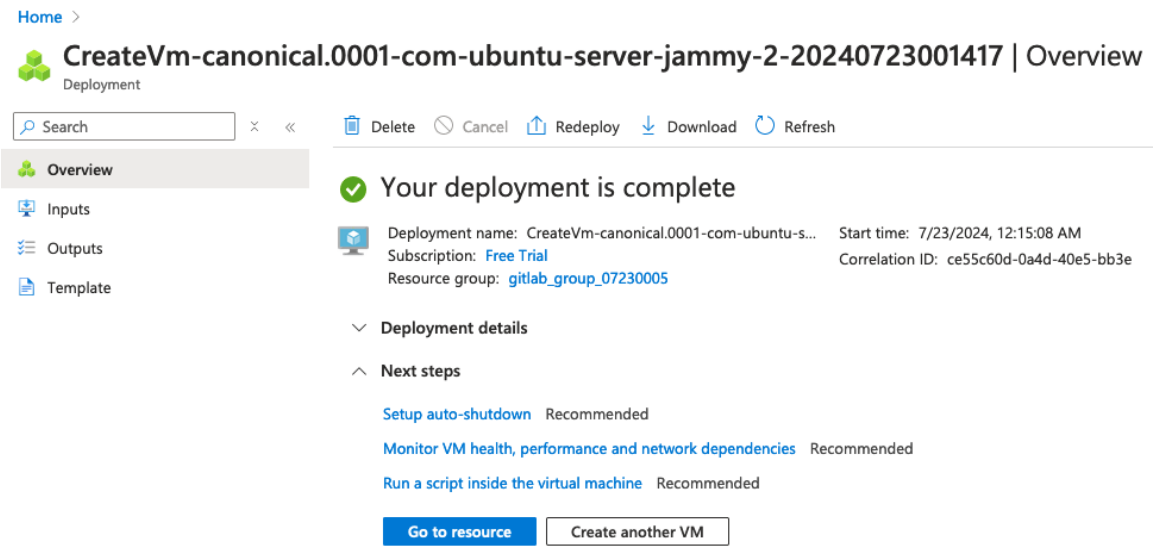


... so about a dime per hour. If this lab takes you 2 hours to complete, you'll be billed about 20 cents, provided that you remember to shut it down after you're done. You should have plenty of credits to play with after class is over.

Check all your settings one more time, and then click on “Create.” Make sure to download your new private key in the dialog that pops up ... you cannot download it later, and if you misplace it you will have to re-key your VM via the resource group UI.

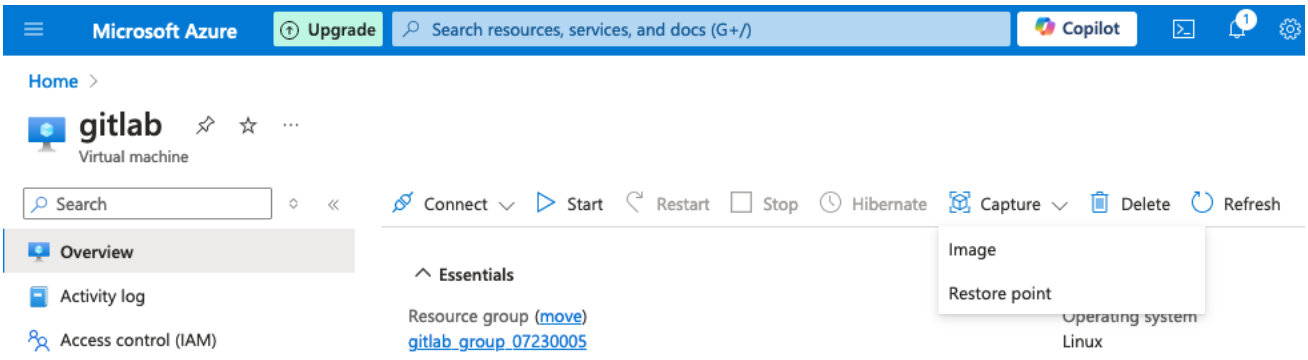


Azure will tell you the deployment is in progress, and when it's done, it will tell you that your deployment is complete. Note the relative cleanliness of Microsoft's Azure interface. Microsoft isn't typically known for aesthetics ... but when and if you try Amazon's AWS, you'll see what we mean here. The AWS web interface is fugly.



You should eventually be told your deployment is complete. Click on "Go to resource," and in the resource screen, note your VM's public IP address.

Shutting down and rebooting your VM can be done on the command line, of course, but there's another means of doing so here. Just like in lab 6, Azure's web interface has power controls on each VM's resource page where not only can you stop, start, and restart an instance, you can also hibernate (pause) it, or use snapshot-like functionality (the capture menu).



These buttons interact with hypervisor tools inside the guest, so each button here will try to cleanly perform the corresponding operation if possible first. For instance, the restart and stop buttons will try to cleanly shut down or reboot the operating system via an agent inside the VM, and if that fails or times out, then they will try a hard shutdown or a hard reset.

part two: configuring your instance

9. First, of course, let's set up DNS for our new system so we can refer to it by name. Add a lookup to your host computer's `hosts` file for your Azure VM's IP address, and the hostname `gitlab.cs470.internal`.

   If you're using Windows and WSL, you'll need to restart your WSL instance to have it regenerate your `hosts` file in WSL. Close all WSL windows, and then run the following command in a command prompt or PowerShell window:

   ```
   wsl.exe --shutdown
   ```

10. In the name server from lab one, on OpenBSD, create a forward lookup to gitlab.cs470.internal for your Azure VM's IP address, and make sure to increment the serial number of your cs470.internal zone file before restarting your name server. Make sure the lookup works.

11. Go to wherever you saved your SSH key file (likely your Downloads folder). It will have a `.pem` extension.

    In case you haven't noticed already, or missed it in the new lab zero, WSL mounts your Windows operating system drive on `/mnt/c` ... so if you're on Windows, `cd /mnt/c` using Ubuntu in WSL. Now you can view (and modify) your Windows directories and files through the Ubuntu shell ... cool, huh? For me, when in Windows and WSL, I use the command …

    ```
    cd /mnt/c/Users/peter/Downloads
    ```

    … to get to my downloads folder. Of course, you'll insert whatever your username is in place of mine to get there on your system.

    Those of you using Macs or Linux, will have a folder called `Downloads` in your home directory, and remember, the tilde character (~) is a shortcut for the path to your home directory, and your `.ssh` folder is at `~/.ssh`.

    Whichever operating system you're on, once you're there, move the key file (the one ending in `.pem`) into your `.ssh` directory, and give it the name `~/.ssh/id_azure` ...

12. `cd` into your `~/.ssh` directory and change the permissions of your key file to be **read-only** for the owner only, with no permissions for anyone else. This is important for SSH to not loudly complain at you about how you store your key material. You'll be using `chmod` to do this, of course ... and you need to figure out the octal permissions "number" or proper switches to use, but you should either have the skills to figure this out by now, or you're leeching off your friends, and should knock it off.

    Test logging into your new instance …

    ```
    $ ssh -i ~/.ssh/id_azure peter@gitlab.cs470.internal
    ```

You should be prompted to add the SSH host key fingerprint to your `known_hosts` file like the first time you SSH into any system, and once you agree, you should be passwordlessly logged in.

13. Note that you logged in using only an SSH key. Also note, this SSH key had no password to protect it. So, if that keyfile got out … so would access to your instance. Let's fix that.

    ```
    $ ssh-keygen -p
    ```

    `ssh-keygen` will ask you to choose a password. If you haven't made sure that your permissions are appropriate for crypto key material, `ssh-keygen` is going to politely pitch a fit.

14. Copy over your other SSH public key, the one you generated way back in lab zero, into the `authorized_keys` keyring file on our Azure VM.

    ```
    $ ssh-copy-id -f -i ~/.ssh/id_rsa -o 'IdentityFile ~/.ssh/id_azure'
    peter@gitlab.cs470.internal
    ```

    You can ignore most of the output. If you did everything correctly, you should now be able to log into your Azure instance just like the rest of your other VMs, without having to invoke the SSH key that came along with the instance.

15. As in step #13 from lab 4, let's take a couple minutes to install packages used later by your instructor for grading purposes: `csh` (via `tcsh`), GNU `binutils`, `net-tools`, and `gcc`.

    ```
    $ sudo apt update
    $ sudo apt install tcsh binutils net-tools gcc
    ```

part three: SSH port forwarding

As well as having a VM, you have a Virtual Private Cloud (VPC), a private NAT'd network in the cloud just like your VMware NATwork that hosts all your VMs on your lab machine.

```
$ ssh gitlab.cs470.internal ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
qlen 1000
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default
qlen 1000
    inet 10.1.0.4/24 metric 100 brd 10.1.0.255 scope global eth0
       valid_lft forever preferred_lft forever
```

You can see the familiar loopback network above, along with the 10.1.0.0/24 network Azure has set up to NAT to. This is the VPC network. You're able to do anything with that network, including running hosts that aren't publicly visible. No traffic gets into the VPC network from the outside unless you explicitly take steps to allow things in … it's yours, hence the "private" in "virtual private cloud" (VPC).

Bringing this back to GitLab, e-mail connectivity is almost mandatory while setting up and using GitLab. There are ways around it, but new user invitations are sent over e-mail, and if we were to get extensively into heavy GitLab usage … virtually all notifications are best done via e-mail. Since we aren't using a public domain (we're using cs470.internal) for our lab e-mail, we really need GitLab in our private network, and the way we'd typically do that would be with a VPN.

A persistent site-to-site VPN tunnel is precisely what we often use to connect various parts of a larger private network, across physically disparate locations, like our business headquarters, to its satellite offices, and to a data center presence, whether it be hosted physical systems or a cloud service like AWS or Azure. This would allow our systems, including any at Azure, to talk to each other across the private network, behind the firewalls, just like our VMs inside VMware Fusion or Workstation are able to speak to one another directly, but using an encrypted virtual wire (the VPN) where those communications need to transit public networks.

In this part of the lab, I really wanted you to set up a point-to-point VPN between Azure and that VMware NATwork, but there are too many obstacles … most notably that our lab machines are often our laptops, or on consumer internet connections, so there's not a fixed IP address at that side of the connect. So, I had to figure out a way around that … and that way is to use SSH port forwarding again.

Enough story time, however …

16. We need to get our Azure instance's e-mail traffic back to our FreeBSD VM … so we want to use the *remote* forwarding capability to send traffic back from the remote system, over the SSH connection back from server to client, to a port on our local private cloud.

    In order to do this manually, we'd use this command on our local host operating system …

    ```
    $ ssh -R 2525:10.42.77.72:25 gitlab.cs470.internal
    ```

    … this means you're forwarding port 2525 from the remote system to TCP port 25 (SMTP) on the IP address 10.42.77.72 (my FreeBSD VM) from your SSH client. However, this is painful, so let's use and abuse the SSH config file instead of doing this manually. Add the following block of configuration to `~/.ssh/config` on your host system …

    ```
    Host gitlab.cs470.internal
    RemoteForward 2525 10.42.77.72:25
    ```

    … and log out and log back into your Azure instance. In order to test this port forwarder, we'll need the `netcat` utility on our Azure instance (often called `nc`), so first install it …

    ```
    $ sudo apt -y install netcat-openbsd
    ```

    … and then use it, on the remote Azure instance, to cause a connection to open to TCP port 2525 on the loopback adapter of your Azure instance. If you did everything correctly, you should see an SMTP service banner for your FreeBSD VM's mail server …

    ```
    $ nc localhost 2525
    ```

```
220 freebsd.cs470.internal ESMTP Sendmail 8.18.1/8.18.1; Tue, 23 Jul 2024
20:43:29 -0700 (PDT)
```

… because TCP port 2525 on the loopback adapter of your Azure instance is now a tunnel back to your FreeBSD mail server's SMTP port.

!! IMPORTANT NOTE: you need to keep the SSH connection open in order for the mail channel to remain open back to your FreeBSD VM. If you are not logged into your Azure instance via SSH with the port forward set up correctly, mail will not flow back to your mail server.

17. Just like we did for the SMTP service on our other Ubuntu instance in lab 4, let's again install `postfix`.

    ```
    $ sudo apt install postfix
    ```

    Set it up as "internet with smarthost" and specify `gitlab.cs470.internal` as the system mail name. The instance may have a DNS name in the domain cloudapp.net pre-populated, but that DNS name won't resolve outside your Azure VPC network, and won't allow us to get e-mail back to our mail server.

    When you're asked for an SMTP relay host, supply `[127.0.0.1]:2525` … aiming it at the SSH tunnel we set up in the prior step.

    Edit `/etc/aliases` and add the following line to forward root's mail …

    ```
    root: peter@cs470.internal
    ```

    … and of course, replace my username with yours, and run `newaliases` to tell the mail subsystem to re-process the aliases database.

    ```
    $ sudo newaliases
    ```

18. Now, let's configure our Azure instance to send e-mail via our SSH port forwarder; we've already discussed the use of an SMTP "smart host." Let's set up `postfix` appropriately …

    ```
    $ sudo vi /etc/postfix/main.cf
    ```

    … let's open up the `postfix` configuration file. Add, set, or confirm the following variables …

    ```
    myhostname = gitlab.cs470.internal
    mydomain = cs470.internal
    myorigin = gitlab.cs470.internal
    inet_protocols = ipv4
    relayhost = [127.0.0.1]:2525
    ```

    … note these options are scattered throughout `main.cf` and you'll need to find them. You don't really need to put each option where they occur in the file, but it's recommended, and it's really important you get the syntax correct, especially for the `relayhost` option.

19. Now, let's reload `postfix` to force it to read the new configuration settings.

    ```
    $ sudo postfix reload
    ```

    Note port 2525 above is the SSH port forwarder that we just checked with `nc`. Now, let's test sending an e-mail over that channel. Install `mailx` like with our other Ubuntu VM …

    ```
    $ sudo apt -y install bsd-mailx
    ```

    … and then use it to send a test mail …

    ```
    $ echo "test" | mail -s test peter@cs470.internal
    ```

    Now check your mail client. If you got it right, you've got mail!

part four: GitLab

We're going to install GitLab on the Azure VM to serve as our own git repository manager. GitLab, like GitHub, is an "on premises" server product you can use to host your repository on your own systems **and** a public hosting site for projects. We're using GitLab, though, not GitHub ... which not coincidentally is owned by Microsoft, who uses it to spy on our coding and sell us the results.

https://copilot.github.com/

When first we wrote up this lab using Azure back in 2021, Microsoft was just beginning to use the "Copilot" branding for AI-based features in GitHub and Visual Studio. Now, it's pervasive, throughout their web browser (Edge), their operating system, and their entire line of products.

If you've ever used `git` or Github, you've likely only worked with individual repositories. GitLab is a comprehensive repository management system that will manage multiple repositories … and documenting its use is outside the scope of this exercise, but you are encouraged to look into it for sure, as we chose our examples here to be utilitarian to your future endeavors.

https://docs.gitlab.com/

GitLab, as part of its own freemium model, comes both in an Enterprise Edition ("EE") with lots of features for business and professional support, and a Community Edition ("CE") that comes free of charge, and with only community support. We are, of course, going to use the free CE.

20. Going to GitLab's package website for GitLab CE, I searched for the most recently version of a 64-bit Intel (`x86_64`) package for Noble Numbat, the code name for Ubuntu 24.04 (ubuntu/noble). This ended me up at this page.

    Following the directions on that page, I added the GitLab CE repository by running the command in blue, in the upper right hand corner.

```
$ curl -s https://packages.gitlab.com/install/repositories/gitlab/gitlab-
ce/script.deb.sh | sudo bash
```

When it's finished, it should say ...

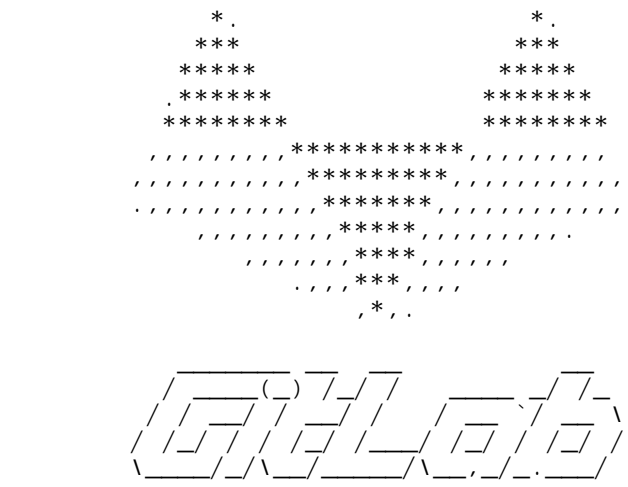
```
The repository is setup! You can now install packages.
```

21. At long last, let's install GitLab …

```
$ sudo apt install gitlab-ce
```

`apt` should do a familiar dance …

```
Setting up gitlab-ce (17.11.0-ce.0) ...
It looks like GitLab has not been configured yet; skipping the upgrade
script.
```



```
Thank you for installing GitLab!
GitLab was unable to detect a valid hostname for your instance.
Please configure a URL for your GitLab instance by setting `external_url`
configuration in /etc/gitlab/gitlab.rb file.
Then, you can start your GitLab instance by running the following command:
  sudo gitlab-ctl reconfigure

For a comprehensive list of configuration options please see the Omnibus
GitLab readme
https://gitlab.com/gitlab-org/omnibus-gitlab/blob/master/README.md
```

22. Like GitLab recommends, edit `/etc/gitlab/gitlab.rb`, change `external_url` to `http://gitlab.cs470.internal`, and then reconfigure GitLab. This should take about 5 minutes.

    At the end of that long process, you should see the following important blurb to aim you at where to find the default login credentials …

```
Notes:
Default admin account has been configured with following details:
Username: root
Password: You didn't opt-in to print initial root password to STDOUT.
```

```
Password stored to /etc/gitlab/initial_root_password. This file will be
cleaned up in first reconfigure run after 24 hours.

NOTE: Because these credentials might be present in your log files in plain
text, it is highly recommended to reset the password following
https://docs.gitlab.com/ee/security/reset_user_password.html#reset-your-root-
password.

gitlab Reconfigured!
```

… and GitLab should be running. How can you tell?

If it's not running, look under `/var/log/gitlab` …

Next, we need to access the GitLab launch webpage. If we're on our private software-defined network inside VMware on each of our computers, we're a whole lot less concerned about encrypting connections with important data, like passwords … so at every other juncture thus far, we've took shortcuts and cut corners. This time, however, we're going over the internet, and we're going to do this The Right Way™.

23. We could use SSH port forwarding to provide encryption here, but then we'd want to access HTTP content under URLs at gitlab.cs470.internal … both on its public IP, and via a loopback adapter with the SSH port forward … but that would be two different IP addresses for different services on the same hostname, and this **not** a workable configuration.

    The Right Way™, of course, is to use HTTPS, instead of HTTP … in order to do this, we'll need an SSL/TLS certificate again, this time for the hostname gitlab.cs470.internal. In a real-world scenario, we'd have real, publicly-usable domain name and would get a certificate from a third-party certificate provider, such as Let's Encrypt. I briefly thought about using `certbot` to get a certificate for both gitlab.cs470.internal AND a public hostname registered using FreeDNS (see lab 5b), but let's use our own CA here.

24. Edit `/etc/gitlab/gitlab.rb` again (sorry), and change `external_url` to have `https` instead of `http` in the protocol portion of the URL.

25. Copy your Azure SSH key over to your failsafe user's key ring on your OpenBSD VM.

    ```
    $ scp -p ~/.ssh/id_azure failsafe@openbsd:~/.ssh/id_azure
    ```

26. Just like we did in labs before, move your Azure VM's default OpenSSL configuration file out of the way …

    ```
    $ sudo mv /etc/ssl/openssl.cnf /etc/ssl/openssl.cnf.orig
    ```

    … then copy the `openssl.cnf` file from your OpenBSD VM up to your Azure VM, move it to `/etc/ssl`, `chown` it to `root:root`, and change the hostname at the end to `gitlab.cs470.internal`.

27. Next, make a directory for certificates for GitLab.

```
$ sudo mkdir -m 700 /etc/gitlab/ssl
```

28. Now, let's make an RSA key for our certificate…

```
$ sudo openssl genrsa -out /etc/gitlab/ssl/gitlab.cs470.internal.key 4096
```

29. Now, let's generate a certificate request (often called a "CSR"), with the above key.

```
$ sudo openssl req -new -key /etc/gitlab/ssl/gitlab.cs470.internal.key -out
/etc/gitlab/ssl/gitlab.cs470.internal.csr
```

    Again, you should recognize the next steps here. No locality name. Same organization name. For the common name, `gitlab.cs470.internal`. **Use your SDSUid e-mail as your e-mail address.**

30. Log into your OpenBSD VM as `failsafe` to copy the CA certificate up to your Azure VM.

```
$ scp -p -i ~/.ssh/id_azure /home/pki/data/cacert.pem peter@gitlab:~
```

    Now go to a root shell on your Azure VM to add the certificate to the list of trusted CAs.

```
# cat cacert.pem >> /opt/gitlab/embedded/ssl/certs/cacert.pem
```

31. Copy `gitlab.cs470.internal.csr` out of GitLab's privileged configuration folders, so we can copy it to OpenBSD.

```
$ sudo cp /etc/gitlab/ssl/gitlab.cs470.internal.csr /tmp/gitlab.cs470.internal.csr
```

32. Grab `gitlab.cs470.internal.csr` from OpenBSD. This is all one line …

```
$ scp -i ~/.ssh/id_azure peter@gitlab:/tmp/gitlab.cs470.internal.csr
/home/pki/newreq.pem
```

    … then have your CA sign it.

```
$ cd /home/pki && CA.pl -sign
```

33. Copy the signed certificate back over to GitLab. Again, single line command.

```
$ scp -i ~/.ssh/id_azure newcert.pem peter@gitlab:/tmp/gitlab.cs470.internal.crt
```

34. Back on GitLab, move it to the right location …

```
$ sudo mv /tmp/gitlab.cs470.internal.crt \
/etc/gitlab/ssl/gitlab.cs470.internal.crt
```

    … fix its ownership.

```
$ sudo chown root:root /etc/gitlab/ssl/gitlab.cs470.internal.crt
```

You should see something like this in your certificate directory when you're done …

```
$ sudo ls -l /etc/gitlab/ssl/
total 16
-rw-r--r-- 1 root root 7549 Jul 24 05:30 gitlab.cs470.internal.crt
-rw-r--r-- 1 root root 1907 Jul 24 05:27 gitlab.cs470.internal.csr
-rw------- 1 root root 3272 Jul 24 05:26 gitlab.cs470.internal.key
```

35. Now let's tell GitLab to reconfigure itself for HTTPS and reload its settings.

    ```
    $ sudo gitlab-ctl reconfigure
    ```

    If you got it right, you should see a listener active on port 443 with `netstat` …

    ```
    $ netstat -an | grep 443
    tcp        0      0 0.0.0.0:443             0.0.0.0:*               LISTEN
    ```

    … if you didn't get it right, look in `/var/log/gitlab` for error messages.

    For me during testing, I had to use `gitlab-ctl` to `stop` and `start` the services to get port 443 to come up instead of 80.

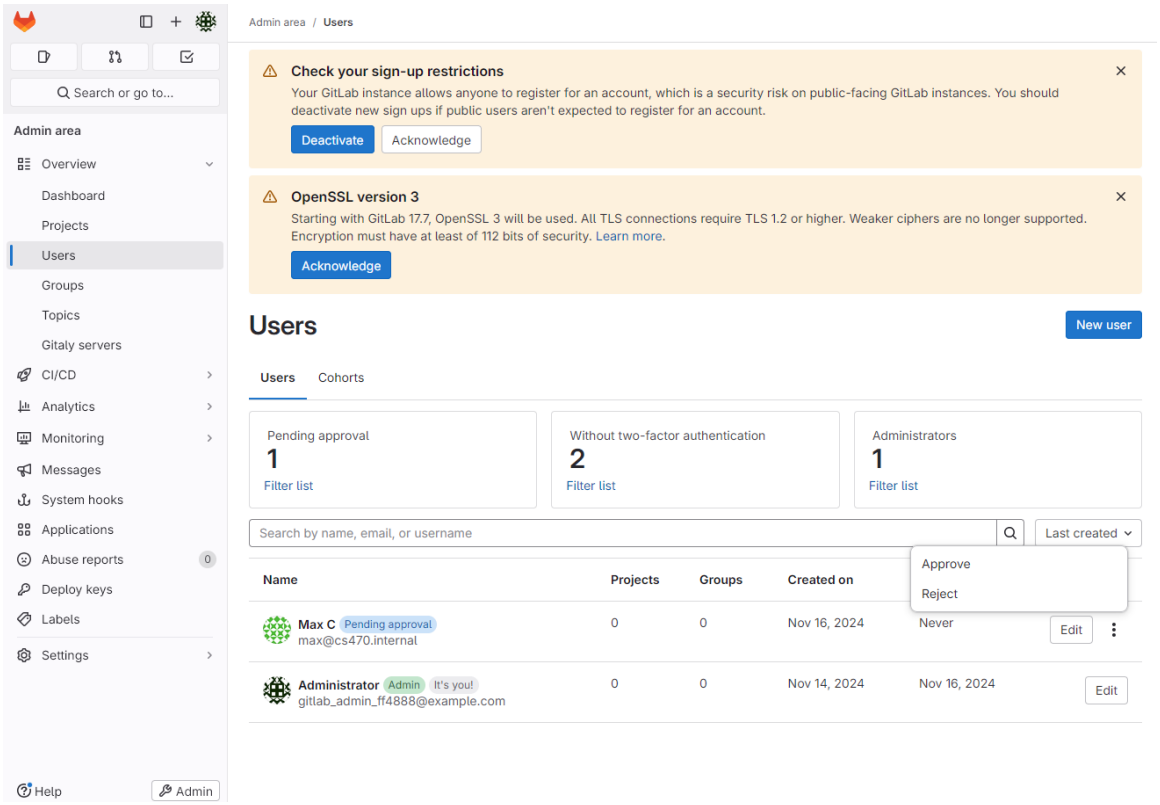
36. Now, open up a web browser, and go to https://gitlab.cs470.internal … you should **not** get a certificate or security warning; you should have a login splash for GitLab, asking you to change your password. If you get a 502 error, don't worry, this is common with GitLab … it sometimes takes a few minutes to get properly spun up. As you could see during the initial installation, it's a pretty big pile of software. Go get a snack or a drink, come back, and reload the tab/page in your browser.

    This is the password for the root GitLab account, so set one up, and then sign in as root using that password you just made. If you aren't asked to change your root password on your first visit to GitLab's web interface, log into it with the password in the file at the path `/etc/gitlab/initial_root_password`. If that file was wiped, you can reset the root GitLab account's password from the command line as follows …

    ```
    $ sudo gitlab-rake "gitlab:password:reset[root]"
    ```

    Then, go to the admin area and change the root password. Once you've done that, log out.

37. Then, register a new, regular non-root user using the web interface with your LDAP username and your LDAP user's @cs470.internal e-mail address. Log back into the root account, go to the Admin portal at the bottom left of GitLab's interface, and click "view latest users" under "Instance Overview."



You should receive an e-mail from GitLab once you've completed your user registration, or you've done something wrong.

IMPORTANT: If the webpage is saying it's taking too long to respond, just refresh it.

Congrats, GitLab is now set up.

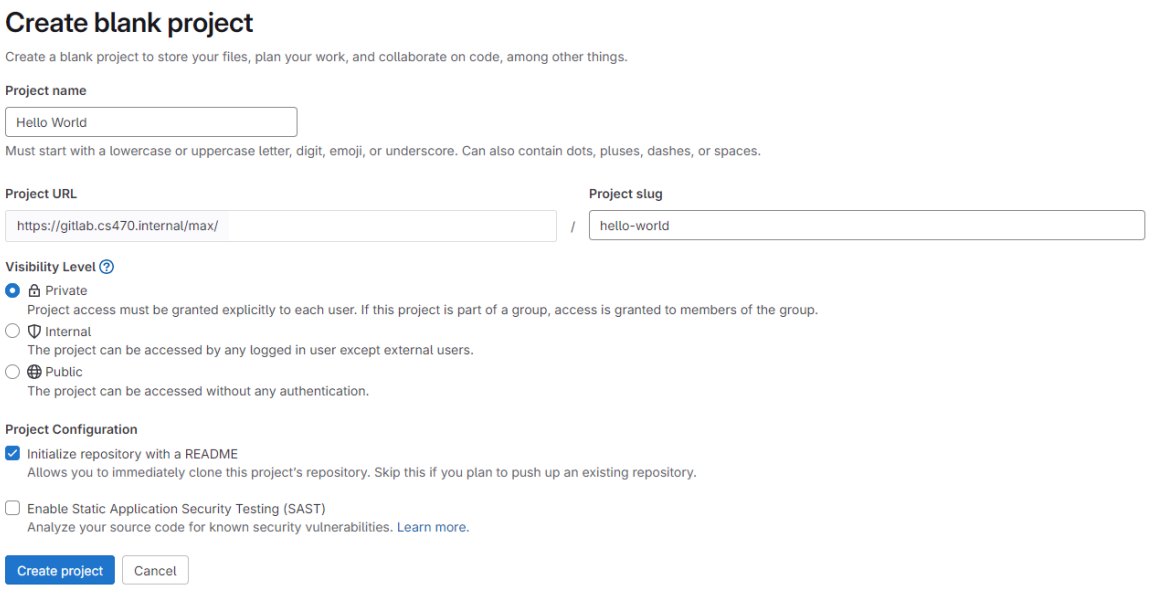
38. Go to the "Preferences" screen again after you log back in, and select "SSH Keys" from the menu on the left. Use the following command ...

```
$ cat ~/.ssh/id_rsa.pub
```

... to show your SSH public key, and then copy and paste it into the text entry box under "key."

39. In your browser, click the plus button in the top left of GitLab's interface, then click "New project/repository" in the dropdown menu. Choose to create a blank project.

Name it `hello-world` and select the option to initialize the repository with a `README.md` file.



40. Once you've created the repository, click the blue "code" button and copy the text under "clone with SSH." Then, on the command line (from within any machine you like now, except maybe your AIX instance), you should be able to use that data with `git` to clone the repository you just created over SSH ...

```
$ git clone git@gitlab.cs470.internal:peter/hello-world.git
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
```

41. `cd` into the `hello-world` folder and create a `hello_world.c` file to do everybody's first programming project. The output of your program should be precisely this ...

```
Hello World!
```

... with a newline at the end. Once you have it working, add the source code file to the repository's staged changes.

```
$ git add hello_world.c
```

Then let's set up our local `git` client to tag up our first commit properly. Use your LDAP user's email for `user.email`, and use your GitLab first and last name for user.name …

```
$ git config --global user.email "peter@cs470.internal"
$ git config --global user.name "Peter Bartoli"
```

These details should match the initial commit created by GitLab. You can see them using Git:

```
$ git log

commit bc5f27e6b12ee2a60cbe5e65fe125f564faaeab6
Author: Peter Bartoli <peter@cs470.internal>
Date:   Sun Nov 17 02:22:45 2024 +0000
```

```
    Initial commit
```

Then, commit your staged changes with the message “my first commit” and push them to the remote repository on GitLab:

```
$ git commit -m ”my first commit”
$ git push
```

Open up the browser again and go to the repo again … voila! It’s there!

42. Test downloading and building your repository contents from your lab 4 Ubuntu VM; this is where it will be graded from. Use the `git` client with SSH key-based authentication.

43. Now that you have a repository set up in GitLab, let's explore how to use the issue tracking feature to keep track of tasks, bugs, or feature requests. In your GitLab repository, click on "Issues" in the left sidebar or on the top navbar.

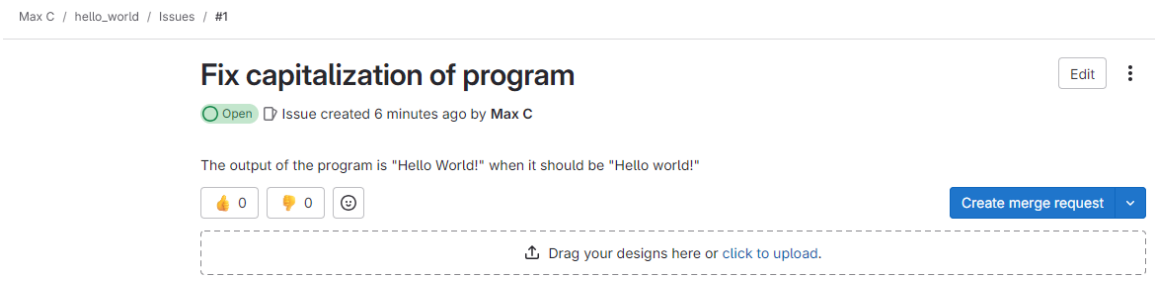
44. To create a new issue, click “issues” and then “new issue.” If you’re not already on your project’s page, you’ll have to select our first and only project.

45. Provide a descriptive title for your issue, such as “fixing capitalization in `hello_world.c`” … in the description box, provide more details about the issue. This is especially important in larger projects, where we’d typically describe the steps to reproduce the issue, what the expected behavior is versus the actual behavior, etc.

46. To organize your issues, you can assign labels to them. Click “Edit” on the “Labels” menu and create a new label, like “bug” or “enhancement.” Labels can help you filter and sort issues more effectively.

47. You can set a due date for the issue by clicking on the “Due Date” calendar icon and selecting a date. This can help you keep track of deadlines and prioritize tasks.

48. Once you have filled in the necessary information, click on the “Submit issue” button at the bottom of the page.



Take note of the number at the top left of the screen. In GitLab, every issue and merge request has a unique number that allows us to easily refer to that specific issue/merge request without typing its full name. Since I haven’t created any issues or merge requests prior to this

step, the issue's number should be #1.

49. Now that the issue has been created, you can view and manage it from the "Issues" tab. You can filter issues by their status (open or closed), assignee, label, or due date using the options at the top of the page.

50. You can also link your issues to specific merge requests (also known as pull requests) by referencing the issue number in the merge request title or description with a `#`. This creates a connection between the issue and the code changes, making it easier to track the progress of a task or bug fix.

51. You can also link your issues to specific merge requests by referencing the issue number in the merge request title or description. This creates a connection between the issue and the code changes, making it easier to track the progress of a task or bug fix.

52. Get to fixing that issue, modifying your "hello world" program's output to have only a single capital letter, at the very beginning of the output sentence. Once you've fixed the bug and added `hello_world.c` to your repo's staged changes, you can reference the issue using its issue number in your commit message …

```
$ git add hello_world.c
$ git commit --message "Fixed capitalization. Closes #1"
$ git push
```

… and GitLab will recognize the phrase "Closes #" in the commit message and automatically close the issue for you.

part five: patching and conclusion

Repeat the patching `cron` job setup and manually patch this system, as you did with your lab 4 VM, in parts two and eight of that lab. Since your lab 7 VM is also Ubuntu, the configuration is the same.

In future iterations of this lab, we'll hope to explore more public cloud stuff, like setting up our VM the less-easy way, via code, and exploring the ridiculous number of things you can do with Azure and AWS.

If you stay logged into your lab 7 VM with SSH port forwarding, you'll receive all the patching advisory mails in your mail server's inbox. There are some pretty compelling reasons **not** to do long periods of time, as well, or to do so sparingly.

In a public cloud, a VM instance is typically billed according to compute usage (CPU/RAM) and storage time. If your VM is powered down while you're not using it, you're only charged for storage, and your Azure credits will last longer. You can power down your Azure Ubuntu instance the same way you powered down your lab 4 VM, or you can power it down via the Azure web interface.

`</lab7>`