Chapter 5

Using and Managing Virtual Machines

"We all live every day in virtual environments, defined by our ideas."
—Michael Crichton

The introduction by Amazon of its Elastic Compute Cloud (EC2) service in 2006 marked the true beginning of cloud computing. EC2 is based on virtualization technology, which allows one server to run independent, isolated operating systems (OSs) for multiple users simultaneously. Since then Microsoft, Google, and many others have introduced virtual machine (VM) services based on this technology.

In this chapter, we first provide a brief introduction to virtualization technology, and then proceed to describe how to create and manage VMs in the cloud. We start with creating a VM on EC2 and show how to attach an external disk. We then describe Microsoft's solution, Azure, and show how to create VM instances via both the Azure portal and a Python API.

The open source community has also been active in this space. Around 2008 three projects—Eucalyptus from the University of California Santa Barbara [212], OpenNebula from the Complutense University of Madrid [202], and Nimbus from the University of Chicago [191]—released cloud software stacks. Later NASA, in collaboration with Rackspace, released OpenStack, which is widely supported. We describe in this chapter one OpenStack-based system called Jetstream, a facility funded by the U.S. National Science Foundation, and show how to create OpenStack VMs on Jetstream. We provide additional information on Eucalyptus and OpenStack in chapters 12 and 13, respectively.

5.1 Historical Roots

Any modern computer has a set of basic resources: CPU data registers, memory addressing mechanisms, and I/O and network interfaces. The programs that control the computer are just sequences of binary codes corresponding to instructions that manipulate these resources, for example to `ADD` the contents of one register to the contents of another.

There are also important instructions for performing context switches, in which the computer stops executing one program and starts executing another. These state management instructions plus the I/O instructions are termed *privileged*. Such instructions are usually directly executed only by the OS, because you do not want users to be able to access state associated with other computations.

The OS has the ability to allow user programs (encapsulated as processes) to run the unprivileged instructions. But as soon as the user program attempts to access an I/O operation or other privileged instruction, the OS traps the instruction, inspects the request, and, if the request proves to be acceptable, runs a program that executes a safe version of the operation. This process of providing a version of the instruction that looks real but is actually handled in software is called **virtualization**. Other types of virtualization, such as virtual memory, are handled directly by the hardware with guidance from the OS.

In the late 1960s and early 1970s, IBM and others created many variations on virtualization and eventually demonstrated that they could virtualize an entire computer [104]. What resulted was the concept of a **hypervisor**: a program that manages the virtualization of the hardware on behalf of multiple distinct OSs. Each such OS instance runs on its own complete VM that the hypervisor ensures is completely isolated from all other instances running on the same computer. Here is an easy way to think about it. The OS allows multiple user processes to run simultaneously by sharing the resources among them; the hypervisor below the OS allows multiple OSs to share the real physical hardware and run concurrently. Many hypervisors are available today, such as Citrix Xen, Microsoft Hyper-V, and VMWare ESXi. We refer to the guest OSs running on the hypervisors as VMs. Some hypervisors run on top of the host machine OS as a process, such as VirtualBox and KVM, but for our purposes the distinction is minor.

While this technical background is good for the soul, it is not essential to learning how to create and manage VMs in the cloud. In the remainder of this chapter we dig into the mechanics of getting science done with VMs. We assume that you are familiar with Linux and focus in our examples on creating Linux VMs. This choice does not imply that Windows is not available. In fact, all three public clouds we talk about in this book allow you to create VMs running Windows just as easily as Linux. So if you need a Windows VM for some of your work, rest assured that almost everything that we present works for Windows, too. We try to point out the occasional exceptions to this rule.

Each of the three public clouds and the NSF Jetstream cloud has a web portal that guides you through the steps needed to create and manage VM instances. If you have never used a cloud, you are well advised to start there. We introduce selected interfaces and describe how to get started with each.

5.2 Amazon's Elastic Compute Cloud

We describe first how to create VM instances on Amazon's Elastic Compute Cloud service and then how to attach storage to our VMs.

5.2.1 Creating VM Instances

We start on the Amazon portal at `aws.amazon.com`, where we can log in or create an account. Figure 5.1 shows what we see when we log in. We are interested in VMs, so we click on `EC2` or `Launch a Virtual Machine`. This brings us to another series of views, with instructions on how to launch a basic "Amazon Linux" instance. We can then specify our desired host service `Instance Type`, which determines the number of cores that our VM is to use, the required memory size, and network performance. Literally dozens of choices exist, ordered from small to large, and priced accordingly (more on this below).

One important step during the launch process involves providing a key pair: the cryptographic keys that you use to access your running instance. If this is your first experience with EC2, you may be asked to create a key pair early in the process. You should do so. Give it a name and remember it. You then download the *private key* file to a secure place on your laptop where you can access it again. The corresponding *public key* is stored with Amazon. Just before you launch your instance, it asks you which key pair you want to use. After you select it, the public key is loaded into the instance. The other important choices involve storage options and security groups. We return to those later. Once you launch the instance you can monitor its status, as shown in figure 5.2 on the next page, where you see two stopped instances and one newly launched instance. The *Status Checks* shows that the new instance is still initializing. After a few moments, its status changes to a green check mark to indicate that the instance is ready to launch.

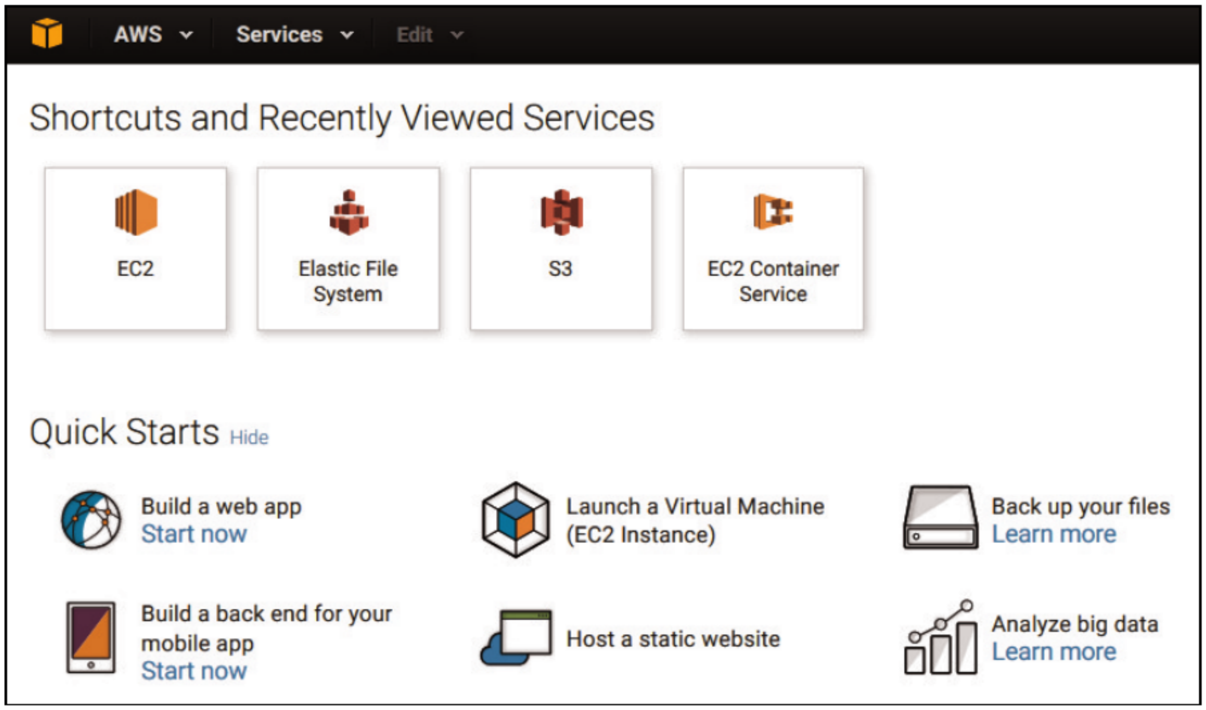
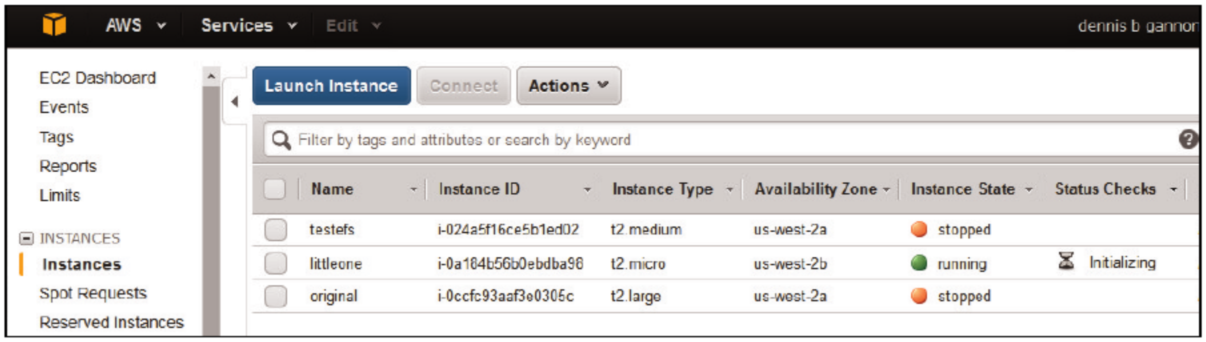


Figure 5.1: First view of the Amazon portal.



Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks
testefs	i-024a5f16ce5b1ed02	t2.medium	us-west-2a	● stopped	
littleone	i-0a184b56b0ebdba98	t2.micro	us-west-2b	● running	Initializing
original	i-0ccfc93aaf3e0305c	t2.large	us-west-2a	● stopped	

Figure 5.2: Portal instance view.

To connect to your instance, you need to use a secure shell command. On Windows the tool to use is called *PuTTY*. You need a companion tool called *PuTTYgen* to convert the downloaded private key into one that can be consumed by PuTTY. When you launch it, you use `ec2-user@IPAddress`, where the *IPAddress* is the IP address you can find in the Portal Instance View. The PuTTY *SSH* tab has an *Auth* tab that allows you to upload your converted private key. On a Mac or Linux machine, you can go directly to the shell and execute:

```
ssh -i path-to-your-private-key.pem ec2-user@ipaddress-of-instance
```

The following listing uses the Python Boto3 SDK to create an Amazon EC2 VM instance. It creates an `ec2` resource, which requires your `aws_access_key_id` and `aws_secret_access_key`, unless you have these stored in your `.aws` directory. It then uses the `create_resources` function to request creation of the instance.

```
import boto3
ec2 = boto3.resource('ec2', 'us-west-2')
ec2.create_instances(ImageId='ami-7172b611', 't2.micro',
                     MinCount=1, MaxCount=1)
```

The `ImageId` argument specifies the VM image that is to be started and the `MinCount` and `MaxCount` arguments the number of instances needed. (In this case, we want five instances, but we will accept a single instance if that is all that is available.) Other optional arguments can be used to specify instances. For example, the *instance type*: Do you want a small virtual computer, with limited memory and computing power, or a big one with many cores and lots of storage? (As we discuss below, you pay more for the latter.) Having created the instance(s), we define and call a `show_instances` function that uses `instances.filter` to obtain and display a list of running instances. The last line shows the result.

```
# A function that lists instances with a specified status
def show_instance(status):
    instances = ec2.instances.filter(
        Filters=[{'Name':'instance-state-name','Values':[status]}])
    for instance in instances:
        print(instance.id, instance.instance_type,
              instance.image_id, instance.public_ip_address)

show_instance('running')
('i-0a184b56b0ebdba98', 't2.micro', 'ami-7172b611', '146.137.70.71')
```

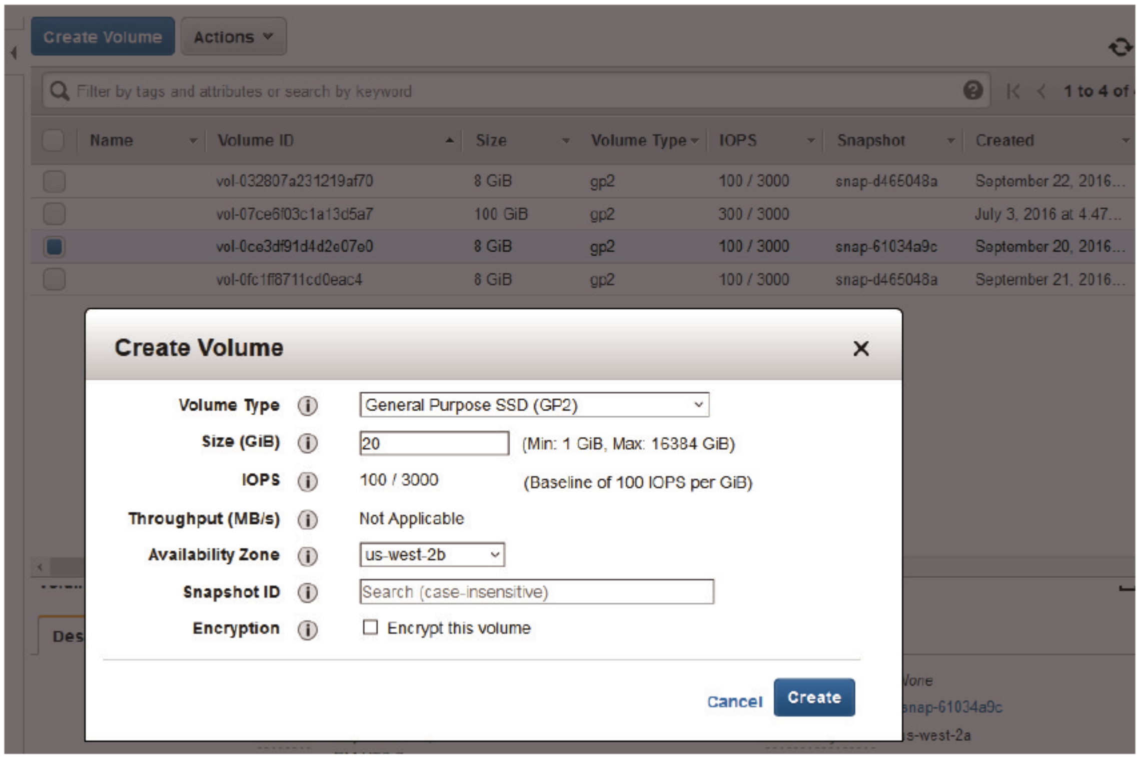
Notebook 7 provides more examples, showing, for example, how to suspend and terminate instances, check instance status, and attach a virtual disk volume.

5.2.2 Attaching Storage

We now discuss the three types of storage that, as noted in chapter 2, can be attached to a VM: instance storage, Elastic Block Store, and Elastic File System. **Instance storage** is what comes with each VM instance. It is easy to access, but when you destroy a VM, all data saved in its instance storage goes away.

We allocate **Elastic Block Store (EBS)** storage independent of a VM and then attach it to a running VM. EBS volumes persist and thus are good for databases and other data collections that we want to keep beyond the life of a

VM. Furthermore, they can be encrypted and thus used to hold sensitive data. To create an EBS volume, go to the *volumes* tab of the EC2 Management console and click `Create Volume`. The dialog box in figure 5.3 allows you to specify volume size (20 GB here), encryption state, *snapshot ID*, and *availability zone*.



	Name	Volume ID	Size	Volume Type	IOPS	Snapshot	Created
		vol-032807a231219af70	8 GiB	gp2	100 / 3000	snap-d465048a	September 22, 2016...
		vol-07ce6f03c1a13d5a7	100 GiB	gp2	300 / 3000		July 3, 2016 at 4:47...
■		vol-0ce3df91d4d2e07e0	8 GiB	gp2	100 / 3000	snap-61034a9c	September 20, 2016...
		vol-0fc1ff8711cd0eac4	8 GiB	gp2	100 / 3000	snap-d465048a	September 21, 2016...

Create Volume

Volume Type	General Purpose SSD (GP2)	
Size (GiB)	20	(Min: 1 GiB, Max: 16384 GiB)
IOPS	100 / 3000	(Baseline of 100 IOPS per GiB)
Throughput (MB/s)	Not Applicable	
Availability Zone	us-west-2b	
Snapshot ID	Search (case-insensitive)	
Encryption	☐ Encrypt this volume	

Figure 5.3: Creating an EBS volume.

We selected the `us-west-2b` availability zone because we want to attach the volume to the instance created earlier. The easiest way to make the attachment is through the *Actions* tab in the Volume management console. However, you can do much of the volume attaching and mounting in Python. First let us look at the list of our current volumes. The following is a transcript of an IPython session.

```
In [3]: vols = ec2.volumes.filter(Filters=[])
In [4]: for vol in vols:
                print(vol.id, vol.size, vol.state)
        ('vol-032807a231219af70', 8, 'in-use')
        ('vol-0bdd0584d0833e691', 20, 'available')
        ('vol-07ce6f03c1a13d5a7', 100, 'in-use')
        ('vol-0ce3df91d4d2e07e0', 8, 'in-use')
        ('vol-0fc1ff8711cd0eac4', 8, 'in-use')
```

We see that the 20 GB volume that we created in the portal session above is *available*, so let's attach it to our instance, as follows.

```
In [5]: vol = ec2.Volume('vol-0bdd0584d0833e691')
In [6]: vol.attach_to_instance(InstanceId='i-0a184b56b0ebdba98',
                               Device='/dev/xvdh' )
        {u'AttachTime': datetime.datetime(2016,9,23,18,15,49,308000,
                                                    tzinfo=tzutc()),
         u'Device': '/dev/xvdh',
         ... more attach metadata not shown ...
        }
```

We must next mount the volume. We cannot use Boto3 for that task, but must log into the instance and issue commands to the instance OS. If we have many instances, we may want to script this task in Python. We can use the `ssh` command to feed the mount commands to the instance directly. We first define a helper function to invoke `ssh` and pass a script, as follows.

```
In [7]: def myexec( pathtopem, hostip, commands):
    ssh = subprocess.Popen(['ssh', '-i', pathtopem,
                'ec2-user@%s'%hostip, commands ],
        shell=False, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
    result = ssh.stdout.readlines()
    if result == ['error']:
        error = ssh.stderr.readlines()
        print >>sys.stderr, "ERROR: %s" % error
        return "error"
    else:
        return result
```

This function requires the path to your private key, the IP address of the instance, and the command script as a string. To do the mount, we can call this function as follows. We first create a file system on our new virtual device. Next we make a directory `data` at the root level and then mount the device at that point. To verify that the job got done, we do a `df` to display the free space.

```
In [8]: command = 'sudo mkfs -t ext3 /dev/xvdh\n \
                  sudo mkdir /data\n \
                  sudo mount /dev/xvdh /data\n \
                  df\n '
In [9]: myexec('path-to-pem-file', 'ipaddress', command)
    [output from the mkfs command ... ]
    'Filesystem     1K-blocks    Used Available Use%Mounted on\n',
    '/dev/xvda1       8123812 1211120   6812444  16% /\n',
    'devtmpfs          501092      60    501032   1%/dev\n',
    'tmpfs             509668       0    509668   0%/dev/shm\n',
    '/dev/xvdh       20511356   45124  19417656   1% /data\n'
```

We have now successfully created and mounted our 20 GB EBS volume on the new instance. One shortcoming of EBS storage is that it can be mounted only on one instance at a time. However, we can detach an EBS volume from one instance and then reattach it to a different instance in the same availability zone.

If you want a volume to be shared with multiple instances, then you can use the third type of instance storage, called Elastic File System, that implements the Network File System (NFS) standard. This takes a few extra steps to create and mount, as shown in the companion Jupyter notebook.

How much do you want to pay? When it comes to paying for cloud computing, public cloud services present a bewildering range of options. On Amazon, for example, these range from less than a cent per hour for a *nano* instance to several dollars per hour for a big-storage or graphical processing unit (GPU) system. And that is just for **on-demand instances**, instances that you request when you need them and pay for by the hour. **Reserved instances** provide lower costs (by up to 75%) when you reserve for between one and three years. And **spot instances** allow you to bid on spare Amazon EC2 computing capacity. You indicate the price that you are prepared to pay, and if Amazon has unused instances and your bid is above the current bid price, you get the machines that you asked for—with the proviso that if your bid price is exceeded during the lifetime of your instances, the instances are terminated and any work executing is lost. Spot prices vary considerably; but you can save a lot of money in this way, especially if your computations are not urgent.

Further complicating things is the fact that prices for different instance types, and especially for spot instances, can vary across Amazon regions. Thus, a really cost-conscious cloud user might be tempted to search across different instance types and regions for the best deal for a particular application. That would be a time-consuming process if you had to do it yourself, but researchers have built tools to do just that. Ryan Chard, for example, has developed a cost-aware elastic provisioner that can reduce costs by up to 95% relative to a less sophisticated approach [90].

5.3 Azure VMs

Microsoft's VM service was announced as Windows Azure in 2008, released to the public in 2010, extended to support Linux and its Python API in 2012, and rebranded as Microsoft Azure in 2014. As in the other public clouds, launching and managing a VM on Azure via the portal is straightforward. As seen in figure 5.4 on the following page, you have many VMs to choose from. Of special interest to us is the "Linux Data Science VM," which contains the R server, Anaconda Python, Jupyter, Postgres database, SQL server, Power BI desktop, and Azure command line tools, and many machine learning tools.

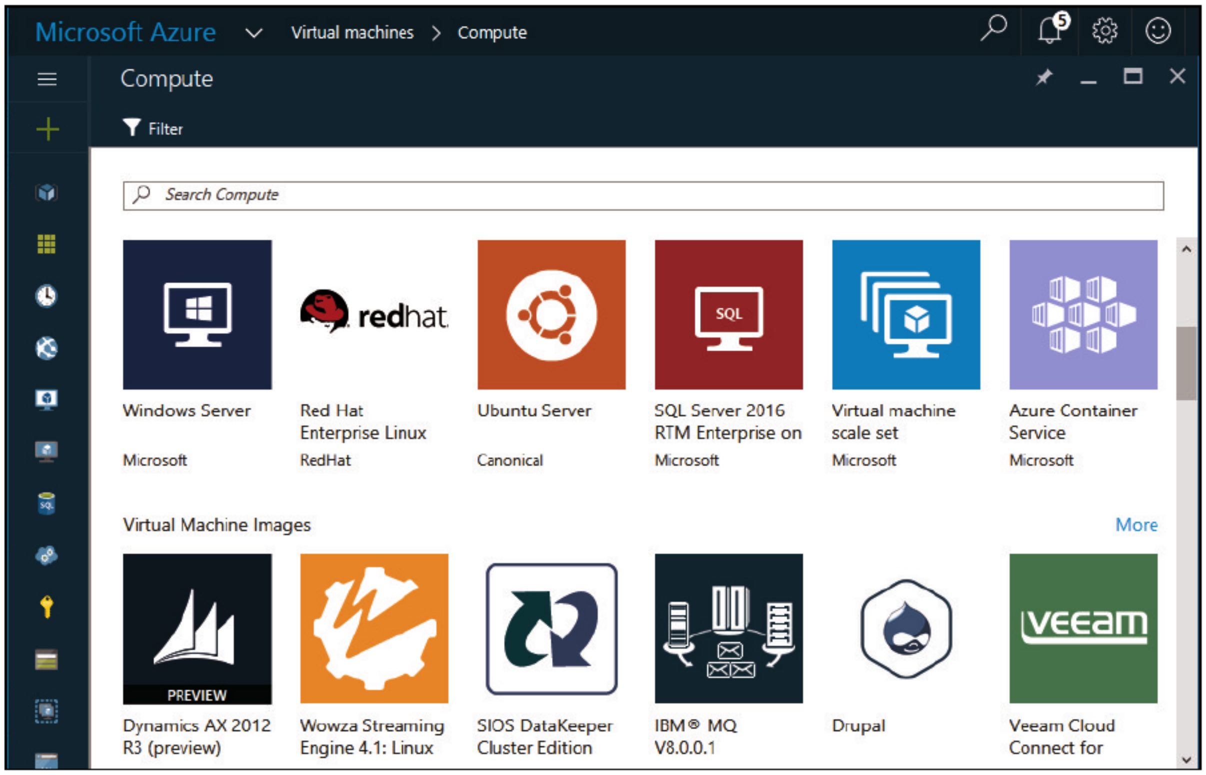


Figure 5.4: Azure portal VM launch page.

As with Amazon, one can write a Python script to launch this VM using the Azure Python SDK introduced in chapter 3. However, another SDK for Azure Python VM management, called **Simple Azure** [179] by Hyungro Lee, allows you to launch the Data Science VM as follows. (Once the status check returns `Succeeded`, your image is running, and you can connect to it.)

```
import simpleazure.simpleazure as sa
a = sa.SimpleAzure()
a.get_config()  # loads your credentials
img = a.get_registered_image(name="Azure-Data-Science-Core")
a.set_image(image = img)
a.set_location("West Europe")
a.create_vm()

#now check status
vars(a.get_status())
```

Using Simple Azure, one can also easily launch an *IPython* cluster and control it from Jupyter. Details are provided in the Simple Azure documentation [179].

5.4 Google Cloud VM Services

You can create VM instances from Google's Cloud portal in a manner similar to Azure and Amazon. A command line interface can also be used to manage VMs, but the Google Cloud Python SDK does not provide a way to launch a VM from a Python script. Google instead provides a unique system for running Python applications in the cloud called **AppEngine**. AppEngine applications are typically web services that scale automatically, a capability that can be useful for some scientific applications. We defer discussion of Google's compute offerings to chapter 7, where we discuss Google's Kubernetes and Cloud Container Service.

5.5 Jetstream VM Services

The Jetstream cloud uses a web interface called Atmosphere that allows users to select and run VM images in a manner similar to that supported by the public clouds. The primary difference is that the list of images available includes many that are packaged specifically for the science community. Here are a few examples.

- CentOS 6 with the MATLAB system and tools preinstalled with licenses available from Jetstream
- Bio-Linux 8, which adds more than 250 bioinformatics packages to an Ubuntu Linux 14.04 LTS base, providing around 50 graphical applications and several hundred command line tools
- The Accurate Species TRee ALgorithm (ASTRAL) phylogenetics package
- CentOS RStudio, which includes Microsoft R Open and MKL (Rblas)
- Wrangler iRODS 4.1 and a setup script for easy generation of the iRODS client environment on XSEDE resources
- Docker, the platform for launching Docker containers
- EPIC Modeling and Simulations: Explicit Planetary Isentropic-Coordinate (EPIC) Atmospheric Model Based on Ubuntu 14.04.3

Numerous Ubuntu and CentOS distributions are also available with various software development tools.

Galaxy, the gold standard for bioinformatics toolkits, is available and widely used on Jetstream. The Galaxy server comes preconfigured with hundreds of tools and commonly used reference datasets. The Galaxy team at the Johns Hopkins University, which hosts the main Galaxy server, is able to offload user jobs to instances running on Jetstream. Approximately 200 Galaxy VM instances are running on Jetstream at any one time. The Galaxy project wiki [20] describes how to deploy the latest Galaxy image.

To create a VM on Jetstream, you must first request an account and allocation via the XSEDE allocation process `www.xsede.org/allocations`. Once that is assigned, you can sign on to the system's web portal where you see a page describing your allocation. The banner at the top is illustrated in figure 5.5.

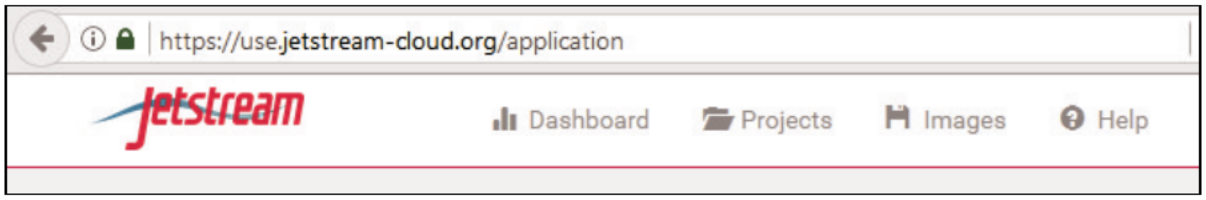


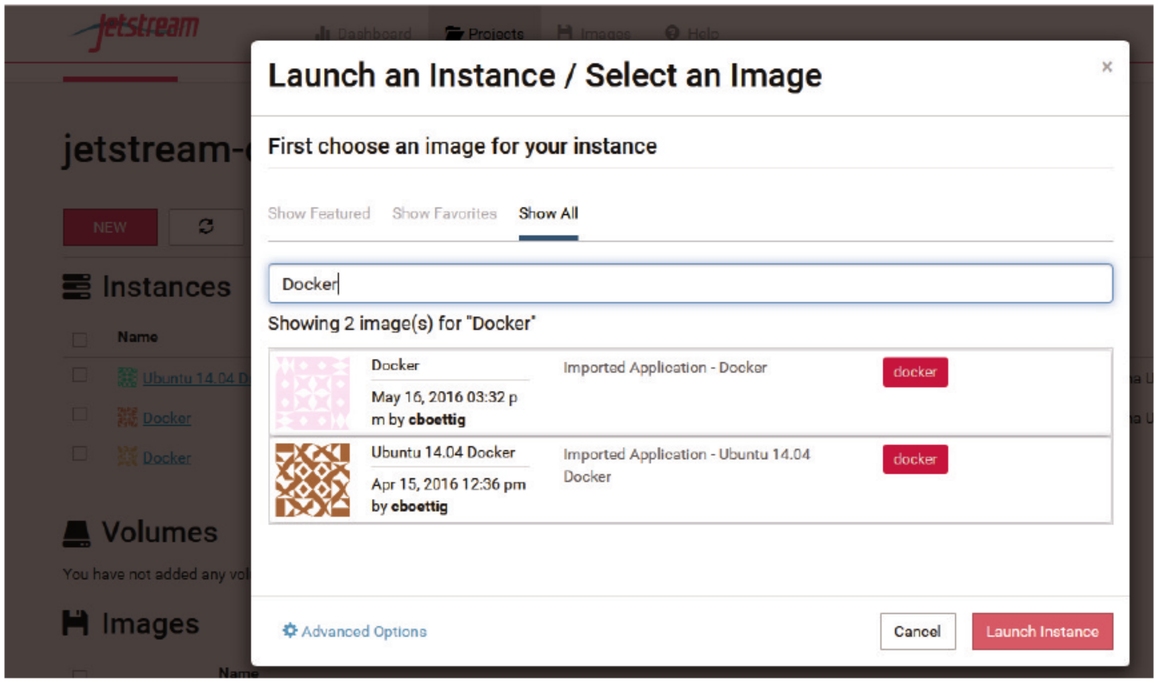
Figure 5.5: Banner for the Jetstream Atmosphere top level page.

You should first select the Projects tab in the banner, which allows you to access your current project or create a new one. Once you have a project, you can select it and you see a list of the instances, data volumes, and images that you have created with that project. By selecting the "NEW" button you can search for the image that is most interesting to you.

Figure 5.6 shows the Jetstream search function being used to locate an image with Docker installed. Once you select an image and click the `Launch Instance` button, you are prompted for the machine configuration. You can then monitor your deployment on your project page, as shown in figure 5.7, where we see two suspended small instances and a new medium-size instance.

5.6 Summary

We have described how to configure and launch a single VM on our candidate clouds. Each system provides a portal that makes configuration easy. The public clouds also provide command line SDKs that can be used to create and manage VMs programmatically. Python SDKs also exist for Amazon, Azure, and OpenStack. Google has an SDK for its AppEngine, which we do not cover here. We have found the Amazon Python SDK for launching VMs and managing the storage to be comprehensive. So, too, is the Azure SDK, but the Simple Azure package (which builds on the standard Azure SDK) makes launching VMs on Azure particularly



Launch an Instance / Select an Image

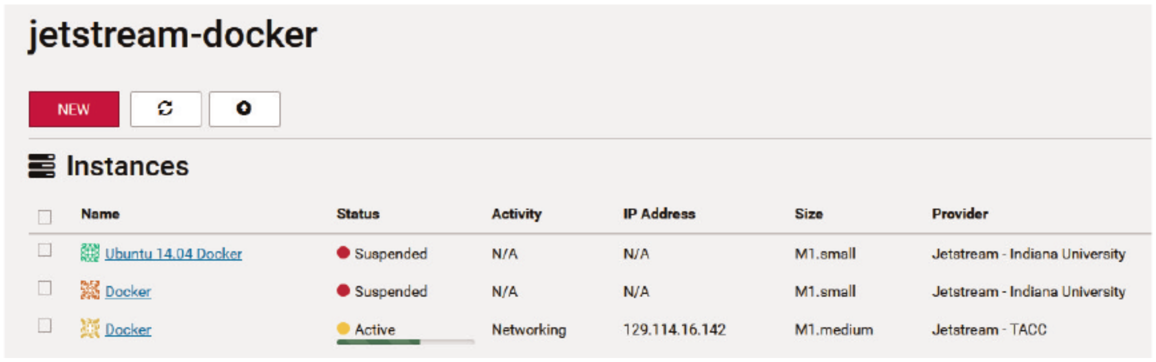
First choose an image for your instance

Docker

Showing 2 image(s) for "Docker"

Docker May 16, 2016 03:32 pm by **cboettig**	Imported Application - Docker	docker
Ubuntu 14.04 Docker Apr 15, 2016 12:36 pm by **cboettig**	Imported Application - Ubuntu 14.04 Docker	docker

Figure 5.6: Search and launch functions for a VM image on Jetstream.



jetstream-docker

Instances

Name	Status	Activity	IP Address	Size	Provider
Ubuntu 14.04 Docker	● Suspended	N/A	N/A	M1.small	Jetstream - Indiana University
Docker	● Suspended	N/A	N/A	M1.small	Jetstream - Indiana University
Docker	● Active	Networking	129.114.16.142	M1.medium	Jetstream - TACC

Figure 5.7: Project page showing the progress of the VM deployment.

simple. Jetstream's Atmosphere web interface makes launching VMs easy. The CloudBridge Python SDK for OpenStack can also be used: see notebook 6.

5.7 Resources

All the code samples described here are packaged as Jupyter notebooks, as described in chapter 17.