Karel de Grote Hogeschool

The Lab - Vagrant

Team
Dries De Houwer
Arno Heyvaert
Lenny Van de Winkel

1 Summary

1 Summary **2**

2 Project description **4**
- 2.1 Purpose 4
- 2.2 Corporate context 4

3 What is Vagrant **5**

4 Vagrant Installation **6**
- 4.1 Windows 6
- 4.2 Linux 6
- 4.3 MacOS 6

5 How does Vagrant work? **7**
- 5.1 Plugins and Providers 7
- 5.2 Vagrant Boxes 7
- 5.3 Vagrantfile 8
- 5.4 Networking 10
- 5.5 Provisioning 10
- 5.6 Triggers 11

6 Vagrant and hypervisors **12**
- 6.1 Default provider and how to change it 12
- 6.2 Multiple providers and machines in one Vagrantfile 13
- 6.3 Multi-purpose Vagrantfile 14

7 Vagrant and the cloud **16**
- 7.1 Google Cloud Platform 16
  - 7.1.1 The plugin 16
    - 7.1.1.1 Plugin installation 16
    - 7.1.1.2 Setting up 16
    - 7.1.1.3 Setting up a simple cloud instance 17

8 Vagrant and provisioning tools **20**
- 8.1 Shell 20
  - 8.1.1 Inline script 20
  - 8.1.2 External script 21
- 8.2 Ansible 21
  - 8.2.1 Ansible provisioner 21
  - 8.2.2 Ansible-local provisioner 22
- 8.3 Docker 22

9 Vagrant for Developers **24**

10 Vagrant for DevOps Engineers **26**

10.1 Use case 26
10.2 Deploying a web application in the cloud 26
10.2.1 Getting GCE ready 27
10.2.2 Deploying 29
10.3 Creating a Vagrant box 32
10.3.1 From scratch 32
10.3.1.1 Virtual machine creation 32
10.3.1.2 Software installation and configuration 37
10.3.2 Based on a base box (forking) 38
10.3.3 Packaging boxes 39
10.3.3.1 Self made base box 39
10.3.3.2 Based on a base box (forking) 39
10.3.4 Adding custom boxes to the list of local boxes 40
10.3.5 Testing local boxes 40
10.3.6 Uploading Vagrant boxes to Vagrant Cloud 41
10.3.6.1 CLI 42
10.3.6.2 Vagrant cloud website 43
11 Conclusion 47
12 Sources 48

2 Project description

The goal of this project is to explore the possibilities of Vagrant. We will explore the possibilities of this tool, how it can be used by developers and operators and what the possibilities of Vagrant are inside a corporate setting. All code examples can be found on our Git repository.

2.1 Purpose

The purpose of this project is to explore the full potential and capabilities of Vagrant in different contexts (Dev, DevOps, et cetera) and how it can be used with other (provisioning) tools and hypervisors.

Vagrant can deploy virtual environments locally and in the cloud. We will make a comparison between Vagrant and other IaC tools such as Terraform, AWS Cloudformation, ….

2.2 Corporate context

We will explore the possibilities of Vagrant in a corporate setting and will find out if it is beneficial if companies use it in their development, testing and production environments.

3 What is Vagrant

Vagrant is an open source tool created by Hashicorp. Vagrant can be used to create virtual machine environments and manage these environments. The tool can use multiple hypervisors and provisioning tools to create and manage these virtual machines.

Vagrant uses "provisioners" and "providers". Provisioners are tools that allow the user to configure and customise virtual machines. Popular provisioning tools are Chef, Puppet, Ansible, Powershell & Bash scripts, …. Providers are services that are being used by Vagrant to create the virtual environments. Some examples of these providers are Virtualbox, Hyper-V, KVM, Docker, VMware, ….

4 Vagrant Installation

4.1 Windows

We can install Vagrant on Windows by browsing to the official website, downloading and then running the Vagrant binary.

4.2 Linux

Vagrant can either be installed through using a binary or through a package manager (apt, yum, dnf, homebrew). The binary and the commands to install Vagrant are listed on the official website.

4.3 MacOS

Vagrant can either be installed using the binary or Brew. The binary and the command to install Vagrant are listed on the official website.

5 How does Vagrant work?

5.1 Plugins and Providers

Vagrant makes use of plugins to add additional functionality that the core installation does not provide. Plugins are an important feature of Vagrant and the chance that we will have to install and use plugins while setting up an environment is very likely. Plugins can be installed with the `vagrant plugin install` command like shown below.

```
# Installing a plugin from a known gem source
vagrant plugin install my-plugin

# Installing a plugin from a local file source
vagrant plugin install /path/to/my-plugin.gem
```

When a plugin is successfully installed , it will automatically be loaded and ready for use.

A great example of the usage of plugins in Vagrant are providers. These providers are the platforms on which Vagrant can set up an environment. Fresh out of the box Vagrant supports three providers, these are Virtualbox , Hyper-V and Docker. When we plan to use Vagrant for production purposes, we probably want to use other providers than these three. Extra providers can then be installed with the help of plugins.

5.2 Vagrant Boxes

Vagrant uses boxes. A box is a base image of a virtual machine. Vagrant uses these boxes as a base image when creating a virtual environment. There is a wide range of boxes available on the official website. We can compare these boxes to Docker images.

We can also create our own Vagrant boxes but more on that later.

In the example below, we will define a virtual environment with an Ubuntu 18.04 box .
We can do this in 2 ways:

1. Using the CLI
2. Inside our Vagrantfile

CLI:
The command below creates a base Vagrantfile with comments and some example code. The name of this Ubuntu 18.04 box is "hashicorp/bionic64".

```
vagrant init hashicorp/bionic64
```

Vagrantfile:

We can easily change the box image when we already have a Vagrantfile. The only thing that needs to be changed is the "config.vm.box" property.

```
Vagrant.configure("2") do |config|
  config.vm.box = "hashicorp/bionic64"
end
```

5.3 Vagrantfile

Vagrantfiles are used by Vagrant to determine which type of machine is needed and how to configure and provision these machines. The syntax is written in Ruby (but considering it's mostly simple variable assignment, knowledge of the Ruby programming language isn't necessary).

The actual filename for Vagrantfiles is "Vagrantfile" and only one Vagrantfile can be used per project. When using any vagrant command, Vagrant looks for the Vagrantfile in the directory that the command was used, climbing down the directory tree until it finds one. You can also set the "VAGRANT_CWD" environment variable to specify a specific path.

An important concept with Vagrantfiles, is that Vagrant actually loads multiple Vagrantfiles and merges them into one. This allows us to set defaults or override prior settings. The order in which Vagrantfiles are loaded goes as follows:

1. Vagrantfile that is packaged with the box.
2. Vagrantfile in the Vagrant home directory (defaults to ~/.vagrant.d).
3. Project directory.
4. Multi-machine overrides (see documentation).
5. Provider-specific overrides (see documentation).

More details about Vagrantfiles can be found in the documentation. A simple example of a Vagrantfile can be found on the next page.

This Vagrantfile makes use of KVM to deploy an Ubuntu20.04 virtual machine. We use the inline shell to change the root password and set the timezone of the vm to Asia Ho Chi Minh. We also change the hostname to ubuntu20.04.

```ruby
# -*- mode: ruby -*-
# vi: set ft=ruby :

VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
    config.vm.provider :libvirt do |libvirt|
        libvirt.driver = "kvm"
    end
    config.vm.box = "generic/ubuntu2004"
    config.vm.box_version = "3.1.16"
    config.vm.provision:shell, inline: <<-SHELL
        echo "root:rootroot" | sudo chpasswd
        sudo timedatectl set-timezone Asia/Ho_Chi_Minh
    SHELL

    config.vm.define "ubuntu20.04" do |ubuntu|
        ubuntu.vm.hostname = "ubuntu20.04"
    end

    config.vm.provision:shell, path: "bootstrap.sh"
end
```

5.4 Networking

The core installation of Vagrant comes with high-level networking options for port forwarding, connecting to a public network and the ability to create a private network. These network options are in a way that they work on multiple providers like VirtualBox and VMware.

If these high-level options are not specific enough for your use case there is no need to worry as all providers have their own networking options that you can configure which makes you able to fine tune your networking configuration based on your chosen provider(s).

Networks are configured within a Vagrantfile. We can configure network settings by using the config.vm.network option. Every network has an identifier followed by configuration arguments. Below is an example of a basic network configuration where we port forward port 80 on the guest OS to port 8080 on our host.

```
Vagrant.configure("2") do |config|
  # ...
  config.vm.network "forwarded_port", guest: 80, host: 8080
end
```

It is possible to use multiple networks inside of our virtual environment. We can achieve this by defining multiple config.vm.network options in our Vagrantfile. Most providers use the same order in which the networks are defined inside a Vagrantfile.

5.5 Provisioning

Vagrant makes use of provisioners. Provisioners allow users to automatically install additional software onto a machine. This is a very useful feature of Vagrant since standard boxes are most likely not fit perfectly for certain use cases. By making use of provisioners like Ansible , Chef , Puppet … we can automate this process so that we don't have to do this extra configuration and installation manually every time we spin up an environment.

5.6 Triggers

Vagrant is able to use machine triggers in order to execute specific commands before or after a command. You can specify this by using .before or .after when defining your trigger an example of a trigger that will be run after the up command can be found below.

```
config.trigger.after :up do |trigger|
 ...
end
```

 A trigger needs a command key (also called actions) as the first argument in order to know when the trigger needs to fire. These command keys can be defined as a single key or as an array. Some actions are :up, :destroy, :halt and :package. If your trigger needs to run after every action you can just use the ':all' key. In the code block below are some examples of how to use command keys in triggers.

```
config.trigger.after :up do |trigger|
...
end

# multiple commands for this trigger
config.trigger.before [:up, :destroy, :halt, :package] do |trigger|
...
end
```

In case we want our trigger to run a script we can use the run option or run remote if we want to execute the script remotely. Some of the settings for these options are Inline to run an inline script, path to specify path to your script file. Below is an example of a trigger that uses an inline script.

```
config.trigger.after :up do |trigger|
  trigger.info = "More information"
  trigger.run = {inline: "bash -c 'echo \"hey there!!\" > file.txt'"}
end
```

Another handy feature that Vagrant triggers provide is to use them to execute some ruby like in the example below.

```
ubuntu.trigger.after :up do |trigger|
  trigger.info = "More information"
  trigger.ruby do |env,machine|
    greetings = "hello there #{machine.id}!"
  end
end
```

6 Vagrant and hypervisors

Vagrant makes use of providers, which are services that are being used by Vagrant to create virtual environments. Vagrants uses hypervisors to accomplish this. There are 3 default hypervisors that are supported: Virtualbox, Hyper-V and Docker. We can also use other providers by installing plugins. More information about plugins can be found at chapter 5.1 of this paper or in the official documentation.

Different providers offer different purposes. While Virtualbox might do everything we desire when testing our application, it is recommended to use VMware in production environments because it's more stable and performant than Virtualbox.

6.1 Default provider and how to change it

Vagrant uses Virtualbox as default provider. This can be changed by setting the "VAGRANT_DEFAULT_PROVIDER" environment variable inside a Vagrantfile. The example below shows the configuration that allows us to use Virtualbox as the default provider.

```
ENV['VAGRANT_DEFAULT_PROVIDER'] = 'virtualbox'
```

It is also possible to specify a provider when creating the virtual environment. This can be done by using the "-- provider=[provider]" parameter when running the "vagrant up" command. The example below sets the provider to Virtualbox.

```
vagrant up --provider=virtualbox
```

It is also possible to override default provider settings. The example below changes the specified that is used to a box specifically created for VMware.

```
Vagrant.configure("2") do |config|
  config.vm.box = "bionic64"

  config.vm.provider "vmware_fusion" do |v, override|
    override.vm.box = "bionic64_fusion"
  end
end
```

6.2 Multiple providers and machines in one Vagrantfile

Vagrant can use multiple providers simultaneously. All of this can be achieved by specifying the different providers in a Vagrantfile. An example of such a configuration can be found below. This example will create one VM called "web". We will use Virtualbox to create this VM. A second VMwill also be created. This VM is called "db" and this VM will use VMware desktop.

```
# ...
Vagrant.configure("2") do |config|
  config.vm.define "web" do |web|
    # ...
    web.vm.provider :"virtualbox" do |v|
      # ...
    end
  end

  config.vm.define "db" do |db|
    # ...
    db.vm.provider :"vmware_desktop" do |v|
      # ...
    end
  end
end
```

6.3 Multi-purpose Vagrantfile

Certain providers have certain benefits. Some people prefer to use Virtualbox, others prefer VMware, KVM or other providers. Some hypervisors only support one or two operating systems (KVM, Hyper-V). While all the devs might use Virtualbox, you and your team could use VMware. This makes it hard when multiple people need to use the same Vagrantfile. This is where multi-purpose Vagrantfiles come into play.

An example of a multi-purpose Vagrantfile can be found below. We use the "generic/ubuntu2004" box and the specific box url for Virtualbox, Hyper-V and VMware Desktop. The provider can be set by using the "vagrant up --provider=..." command or by uncommenting one of the ENV lines in the code.

```
# Uncomment the next line to set the default provider to 'VMware
Desktop'
# ENV['VAGRANT_DEFAULT_PROVIDER'] = "vmware_desktop"

# Uncomment the next line to set the default provider to 'Hyper-V'
# ENV['VAGRANT_DEFAULT_PROVIDER'] = "hyperv"

Vagrant.configure("2") do |config|
  config.vm.box = "generic/ubuntu2004"

  config.vm.provider 'virtualbox' do |virtualbox|
      config.vm.box_url =
"https://app.vagrantup.com/generic/boxes/ubuntu2010/versions/3.6.8/provi
ders/virtualbox.box"
  end

  config.vm.provider 'vmware_desktop' do |vmware|
      config.vm.box_url =
"https://app.vagrantup.com/generic/boxes/ubuntu2010/versions/3.6.8/provi
ders/vmware_desktop.box"
  end if Vagrant.has_plugin?('vagrant-vmware-desktop')

  config.vm.provider 'hyperv' do |hyperv|
      config.vm.box_url =
"https://app.vagrantup.com/generic/boxes/ubuntu2010/versions/3.6.8/provi
ders/hyperv.box"
  end if Vagrant.has_plugin?('vagrant-vmware-desktop')
end
```

Note: there are some things that we need to keep in mind when using a multi-purpose Vagrantfile. One of the most important things is that not all boxes have optimised versions for certain providers. You can check the supported providers on the box webpage on the official website. Another important thing to keep in mind is that this example doesn't check which plugin is installed and doesn't use the corresponding hypervisor. Vagrant will use the default provider and will show an error message when the default provider or plugin isn't installed.

7 Vagrant and the cloud

We can use Vagrant with a lot of local providers (hypervisors) but in certain use cases it might be useful to deploy in the cloud. Unfortunately, Vagrant doesn't natively support cloud providers, however there are several plugins available that make this possible. For this example, we're going to try this on Google Cloud Platform.

7.1 Google Cloud Platform

7.1.1 The plugin

Vagrant doesn't support Google Cloud Platform (GCP) natively (or any Cloud provider for that matter), so we'll need to use a plugin that will allow Vagrant to work with GCP. All the necessary documentation can be found here.

7.1.1.1 Plugin installation

The readme page of this plugin is very clear, and the installation of this plugin is easy. All we have to do to install this plugin is to execute the command below.

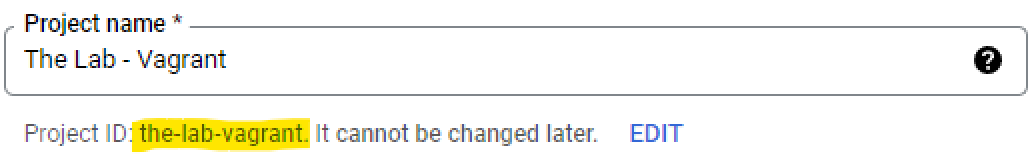
```
vagrant plugin install vagrant-google
```

7.1.1.2 Setting up

Using the GCP plugin is straight-forward. We need three things to get things going:

- The Google Cloud Project ID
- A service file with access
- SSH key set up in the project metadata

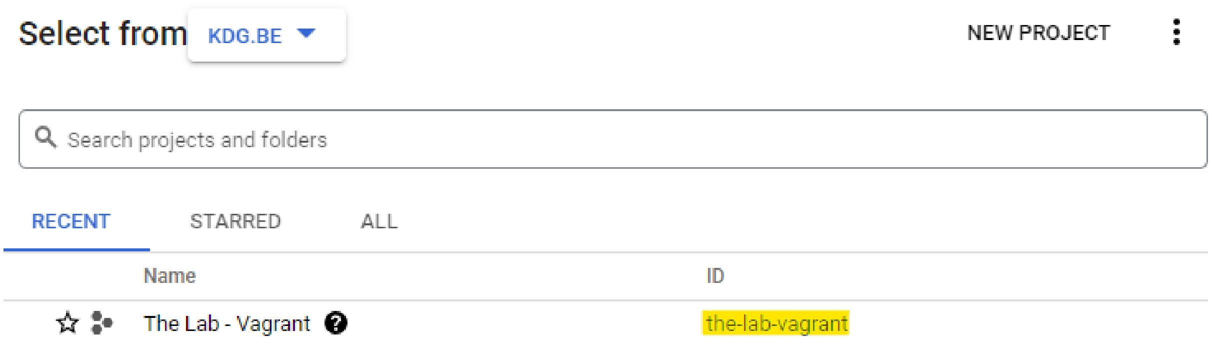
The project ID is displayed upon creation of the project:



Project name *
The Lab - Vagrant

Project ID: the-lab-vagrant. It cannot be changed later.

We can also use already created projects. We can find the project ID by clicking on the list on the top left in the GCP console:



Name	ID
The Lab - Vagrant	the-lab-vagrant

Getting a service file can be done by going to the GCP console: *IAM and admin > Service accounts > Create service account > Follow the steps > Click on your created service account > Keys > Add key > JSON*.

In order to set up a local SSH key so we can SSH into our instances and provision them, we'll need to add our public SSH key to the project's metadata. Info about the creation of SSH keys can be found here. We will add our public key to the GCE metadata after we've created our key pair. We can do this by using the console and following the next steps: *Compute Engine > Metadata > SSH Keys > Edit*

There are a lot of configuration options. We won't go over all of the options but more info can be found in the README file of the plugins Github repository.

7.1.1.3 Setting up a simple cloud instance

Because of the way the plugin works, you cannot use Vagrant Boxes. However, Google Cloud provides you with a load of images you can use for different operating systems. Provisioning is still possible.

We then need to configure a Vagrantfile per the plugin:

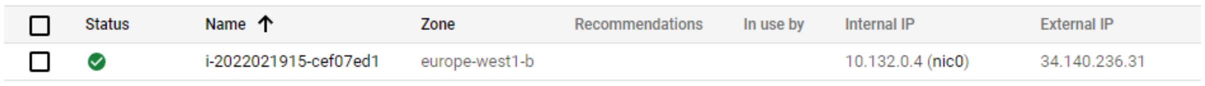
```
Vagrant.configure("2") do |config|
  config.vm.box = "google/gce"

  config.vm.provider :google do |google, override|
    google.google_project_id = "GOOGLE_CLOUD_PROJECT_ID"
    google.google_json_key_location = "/path/to/private-key.json"

    google.image_family = 'ubuntu-2004-lts'

    override.ssh.username = "USERNAME"
    override.ssh.private_key_path = "~/.ssh/id_rsa"
  end

end
```

The data needed is self-explanatory. We're going to deploy a Ubuntu 20.04 LTS machine and try to run a few basic SSH commands on it in the **europe-west1-b** zone. This leaves us with the following Vagrantfile:

```
Vagrant.configure("2") do |config|
  config.vm.box = "google/gce"

  config.vm.provider :google do |google, override|
    google.google_project_id = "the-lab-vagrant"
    google.google_json_key_location =
"servicefile/the-lab-vagrant-a41e33c1ab2d.json"

    google.image_family = "ubuntu-2004-lts"

      google.zone = "europe-west1-b"

    override.ssh.username = "DESKTOP-ENGK677\vdwle"
    override.ssh.private_key_path = "C:/Users/vdwle/.ssh/google_compute_engine"
  end

end
```

We can then use the `vagrant up --provider=google` to deploy this instance.

After a minute or so, the Compute Engine is set up:



☐	Status	Name ↑	Zone	Recommendations	In use by	Internal IP	External IP
☐	✓	i-2022021915-cef07ed1	europe-west1-b			10.132.0.4 (nic0)	34.140.236.31

We can also provision this instance. As a minimal example, we're going to update the instance and install Apache2. We end up with this Vagrantfile:

```
Vagrant.configure("2") do |config|
  config.vm.box = "google/gce"

  config.vm.provider :google do |google, override|
    google.google_project_id = "the-lab-vagrant"
    google.google_json_key_location =
"servicefile/the-lab-vagrant-a41e33c1ab2d.json"

    google.image_family = "ubuntu-2004-lts"

      google.zone = "europe-west1-b"

    override.ssh.username = "vagrantlenny"
    override.ssh.private_key_path =
"C:/Users/vdwle/.ssh/google_compute_engine"
  end

  config.vm.provision:shell, inline: <<-SHELL
    apt update
      apt upgrade -y
      apt install apache2 -y
  SHELL

end
```

As long as our SSH is configured properly, this should run seamlessly and we should have an Ubuntu 20.04 LTS machine that has Apache2 running.

8 Vagrant and provisioning tools

As we stated earlier in this paper, Vagrant makes use of provisioning tools to take care of the configuration. Vagrant supports a wide array of provisioning tools of which we will discuss a few here.

8.1 Shell

The shell provisioner allows users that don't have much experience with full fledged configuration management tools like Ansible, Chef, Puppet … to still easily make further configuration on their boxes. The shell scripts can be defined in two ways, Inline or external.

8.1.1 Inline script

An inline script is a script that is given to Vagrant directly within the Vagrantfile. It is the easiest way for people that are relatively new to the IT world. In the code below are 2 examples of an Inline script.

```
Vagrant.configure("2") do |config1|
  config1.vm.provision "shell",
    inline: "echo Hello, World"
End
##########################
$script = <<-SCRIPT
echo I am provisioning...
date > /etc/vagrant_provisioned_at
SCRIPT

Vagrant.configure("2") do |config2|
  config2.vm.provision "shell", inline: $script
end
```

8.1.2 External script

When we use an external script we must define the path to the script on our host machine or an url to a remote location where the script is located . Vagrant will then upload the script onto the guest machine where it will be executed. Below are 2 examples of how to use an external script in Vagrant.

```
Vagrant.configure("2") do |config|
 config.vm.provision "shell", path: "script.sh"
end
################################################
Vagrant.configure("2") do |config|
  config.vm.provision "shell", path:"https://example.com/provisioner.sh"
end
```

8.2 Ansible

With the ansible provisioner we are able to use ansible playbooks to automate the configuration on our created environment. More specifically Vagrant has 2 ways in which we can use Ansible. These are the normal ansible provisioner and the ansible_local provisioner.

8.2.1 Ansible provisioner

In the Ansible provisioner the playbook is executed on the Vagrant host. This means that our guest vm doesn't need to have ansible installed. The .playbook identifier is used to specify the location of the playbook we want to run. Below is an example of how to use the ansible provisioner.

```
Vagrant.configure("2") do |config|

  #Run Ansible from the Vagrant Host

 config.vm.provision "ansible" do |ansible|
    ansible.playbook = "playbook.yml"
  end
end
```

8.2.2 Ansible-local provisioner

The main difference with the ansible-local provisioner is that with this provider the ansible playbook gets executed from the guest machine. This ofcourse means that the playbook must be present there. To achieve this Vagrant will sync the project directory to ./Vagrant in our guest machine. Below is an example of how to use the ansible-local provisioner in Vagrant

```
Vagrant.configure("2") do |config|
  # Run Ansible from the Vagrant VM
  config.vm.provision "ansible_local" do |ansible|
    ansible.playbook = "playbook.yml"
  end
end
```

As you can see in the code there is no need to first install ansible yourself. The Vagrant provisioner does this for you. You can however customise the way Vagrant instals Ansible with the install_mode option. If you don't want Vagrant to install Ansible automatically you can set the install option to false.

8.3 Docker

The Docker provisioner is another popular choice when it comes to provisioning tools in Vagrant. Just like with the ansible local provisioner Vagrant will take care of installing the needed software for you (Docker in this case).

The easiest way to use the docker provisioner is to just pull an dockerhub image and run it. This is done with the code below.

```
Vagrant.configure("2") do |config|
  config.vm.box = "hashicorp/bionic64"
  config.vm.provision "docker" do |d|
    d.run "rabbitmq"
  end
end
```

In case we want to use an official image as a base we need to just pull it and not run it. To just build the image you can use the pull_images option followed by the name of the image.

```
Vagrant.configure("2") do |config|
  config.vm.box = "hashicorp/bionic64"
  config.vm.provision "docker" do |d|
    d.pull_images "ubuntu"
  end
end
```

Finally it's of course possible for you to build your own images which is done by using the build_image option. Below is a use case of building your own image with the docker provisioner.

```
Vagrant.configure("2") do |config|
 config.vm.box = "hashicorp/bionic64"
 config.vm.provision "docker" do |d|
   d.build_image "/vagrant/app"
 end
end
```

9 Vagrant for Developers

Vagrant supports both local providers and cloud providers which makes it an ideal tool for first testing your environments locally before pushing it to the cloud. In this part we are going to focus on how to use Vagrant to deploy an environment locally using Vmware.

More specifically we are going to deploy two virtual machines, called "db" and "webapp" respectively. The "db" VM will host a MySQL 5.7 database while the "webapp" VM will host a .NET web application. We will make use of the 'generic/ubuntu1804' box for both our virtual machines. This box can be found on the Vagrant cloud.

Since the order in which we define our Virtual machines matters in Vagrant, we will first start configuring our db VM. The reason being that the .NET application on our web app will need to connect to the database. The code snippet below will deploy the db VM with an IP address on a Host-Only network.

```
config.vm.define "db" do |db|
    db.vm.boot_timeout= 600
    db.vm.box = "generic/ubuntu1804"
    db.vm.network "private_network", ip: "192.168.75.50"

    db.vm.provider "vmware_workstation" do |vmware|
    vmware.vmx["memsize"] = '2048'
    vmware.vmx["displayname"] = "db"
    end
end
```

In order to put a MySQL database on this virtual machine we will have to make use of provisioners. In this case we will make use of Docker. We do this by adding the code block shown below.

```
db.vm.provision "docker" do |d|
      d.run "mysql:5.7",
      args: "-e MYSQL_ROOT_PASSWORD=secretPass -e  MYSQL_USER=testuser -e
MYSQL_PASSWORD=testPass -e MYSQL_DATABASE=appdb -p 3306:3306"
    end
```

Next up is our webapp VM. As you can see the hardware- and network configuration is very similar to the db VM.

```
config.vm.define "webapp" do |webapp|
    webapp.vm.boot_timeout= 600
    webapp.vm.box = "generic/ubuntu1804"
    webapp.vm.network "private_network", ip: "192.168.75.10"
    webapp.vm.synced_folder "./playbooks", "/vagrant/playbooks"

    webapp.vm.provider "vmware_workstation" do |vmware|
      vmware.vmx["memsize"] = '2048'
      vmware.vmx["displayname"] = "webapp"
    end
end
```

The only change is the .vm.synced_folder option which allows us to mount a directory from our host OS to our Virtual machine. We make use of this feature to get our Ansible playbooks which we will use to deploy our .NET app on our VM and because we are using the ansible_local provisioner.

Since our host OS is Windows we have to use the ansible_local provisioner. The reason being that Ansible can't be installed on a Windows OS. An alternative for this is to set up another VM using Linux on which you install Ansible. You can then use this VM to deploy your playbooks in your environment. However for the sake of this use case we are going to stick with the ansible_local provisioner option.

```
webapp.vm.provision "shell",
      inline: "sudo add-apt-repository -y ppa:deadsnakes/ppa && sudo apt
update -y && sudo  apt install -y python3.9 && sudo useradd -s /bin/bash -d
/home/ansible/ -m -G sudo ansible"

webapp.vm.provision "ansible_local" do |ansible|
  ansible.playbook = "./playbooks/dotnetapp_playbook.yml"
```

As you can see in the code above we first use the shell provisioner to execute some bash commands that are necessary to run the Ansible playbook.

10 Vagrant for DevOps Engineers

Vagrant can be useful for DevOps Engineers too because it can work in many different ways, such as with different hypervisors, locally, in the cloud, or even a hybrid approach. In this section of this paper, we're going to explore what Vagrant can mean for DevOps Engineers with a common use case.

10.1 Use case

In this analysis, we're going to cover two scenarios:

- Deploying a test version of a web application with a database backend in the cloud
- Creating a Vagrant box for Developers to quickly locally test

10.2 Deploying a web application in the cloud

In order to realise this scenario, we need to prepare a few things:

- **An application**
  We have a .NET web application with database functionality that we're going to use here.
- **A Cloud Provider**
  In this instance we're going with Google Cloud Platform. The reason being that it has an up-to-date plugin for Vagrant.

See Vagrant and the cloud for information about setting up the plugin and getting ready.

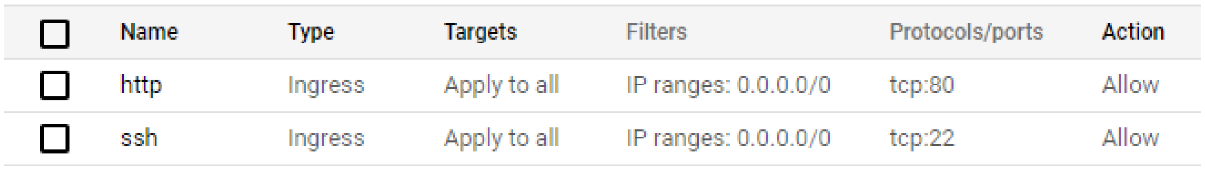
Immediately, we hit a few limitations: The Google Cloud plugin **only** lets us deploy and provision Compute Engine instances. This means that we cannot create a VPC network, firewall rules or even a SQL server.

This ultimately means that we have to pre-create a VPC network with the necessary firewall rules and a SQL server with a fixed user.

10.2.1 Getting GCE ready

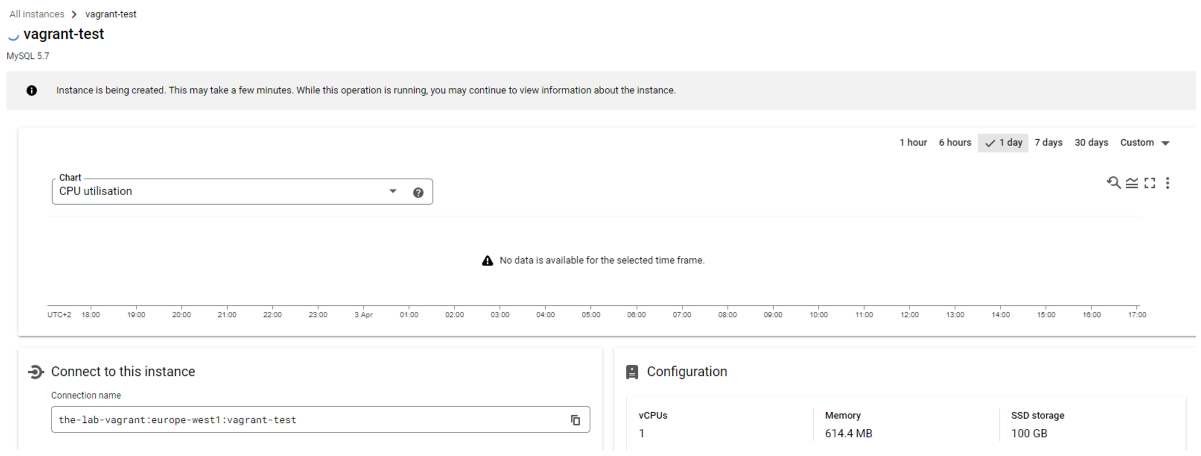
First we're going to create a VPC network. Pick a name and configure it however you want to. We leave everything on default besides the subnet creation mode which is set to automatic. Our network will be named "*vagrant-devops-test*".

Then we need to configure some firewall rules for it. We want to make sure that we can SSH (port 22) into it, that port 80 is accessible and that we can access our SQL server. For the latter, we don't need a firewall rule here. The result should be two rules:



☐	Name	Type	Targets	Filters	Protocols/ports	Action
☐	http	Ingress	Apply to all	IP ranges: 0.0.0.0/0	tcp:80	Allow
☐	ssh	Ingress	Apply to all	IP ranges: 0.0.0.0/0	tcp:22	Allow

Lastly, set up a Cloud SQL server. We choose a basic, single-zone MySQL 5.7 server - note down the root password! For simplicity, public IP is enabled.


All instances > vagrant-test
1 hour
6 hours
1 day
7 days
30 days
Custom
UTC+2 18:00 19:00 20:00 21:00 22:00 23:00 3 Apr 01:00 02:00 03:00 04:00 05:00 06:00 07:00 08:00 09:00 10:00 11:00 12:00 13:00 14:00 15:00 16:00 17:00

vagrant-test
MySQL 5.7

Instance is being created. This may take a few minutes. While this operation is running, you may continue to view information about the instance.

Chart: CPU utilisation

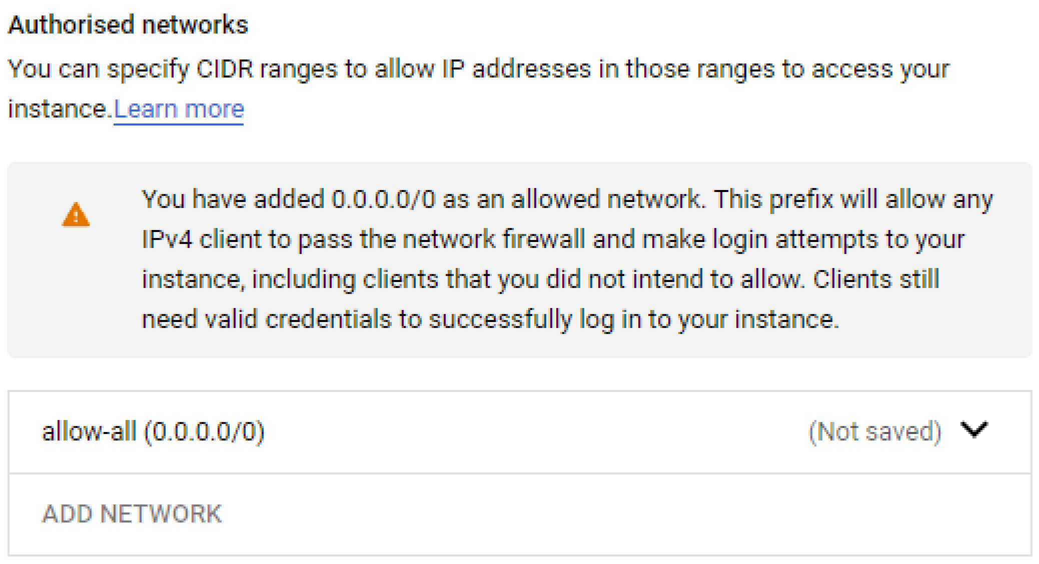
No data is available for the selected time frame.

Connect to this instance
Connection name: `the-lab-vagrant:europe-west1:vagrant-test`

Configuration

vCPUs	Memory	SSD storage
1	614.4 MB	100 GB

Once that's done, ensure your Compute Engine instance can connect. For this case, we're just going to allow everything to connect by adding "0.0.0.0/0" to the allowed list.



Authorised networks

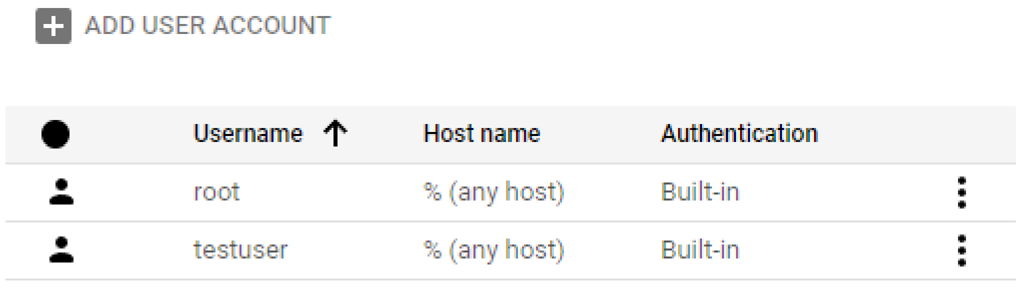
You can specify CIDR ranges to allow IP addresses in those ranges to access your instance. Learn more

You have added 0.0.0.0/0 as an allowed network. This prefix will allow any IPv4 client to pass the network firewall and make login attempts to your instance, including clients that you did not intend to allow. Clients still need valid credentials to successfully log in to your instance.

allow-all (0.0.0.0/0) (Not saved)

ADD NETWORK

Lastly, create a database and a user - note down the credentials!



ADD USER ACCOUNT

●	Username ↑	Host name	Authentication	
👤	root	% (any host)	Built-in	⋮
👤	testuser	% (any host)	Built-in	⋮

10.2.2 Deploying

Now it's a matter of getting a way to provision your Compute Engine instance. We're going to use an Ansible playbook (see 8.2 Ansible) which has been adapted for our specific case with the correct credentials:

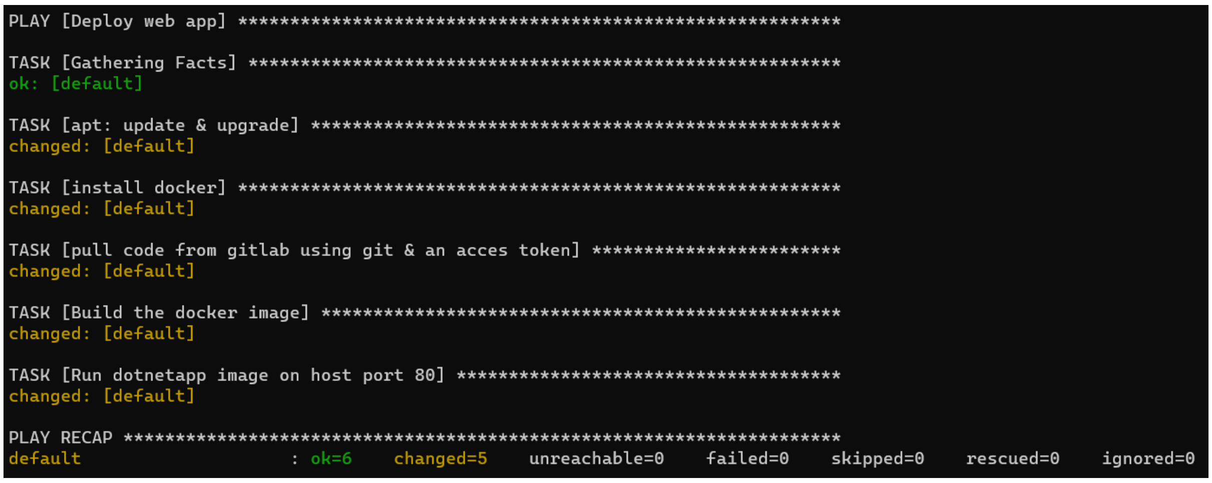
```
---
- name: Deploy web app
  hosts: all
  become: yes
  remote_user: ansible
  become_user: root

  tasks:
  - name: 'apt: update & upgrade'
    apt:
      update_cache: yes
      upgrade: safe
      state: latest
    register: updateUpgrade
  - name: 'install docker'
    apt:
      name: docker.io
      state: latest
      update_cache: yes
    register: installDocker
  - name: 'pull code from gitlab using git & an acces token'
    ansible.builtin.git:
      repo:
"https://access-token:*******@gitlab.com/kdg-ti/the-lab/teams/goeie-vraa
g/dotnetapp.git"
      dest: /var/code
    register: pullCode
  - name: 'Build the docker image'
    command: docker build -t dotnetapp /var/code
    register: buildOutput
  #- debug: msg="{{buildOutput.stdout}}"
  - name: 'Run dotnetapp image on host port 80'
    command: docker run -d -p 80:5000 -e
ASPNETCORE_ENVIRONMENT="Production" -e
connectionString="Server=34.79.166.140; Port=3306; Database=appdb;
Uid=testuser; Pwd=*******; SslMode=Preferred" dotnetapp
    register: runOutput
  # - debug: msg="{{runOutput.stdout}}"
```

Now create your Vagrantfile. Make sure you set the VPC network to the one you just created! We're going to use this configuration, mostly derived from our previous part talking about GCE:

```
Vagrant.configure("2") do |config|
  config.vm.box = "google/gce"

  config.vm.provider :google do |google, override|
    google.google_project_id = "the-lab-vagrant"
    google.google_json_key_location = "/mnt/k/The
Lab/vagrant/vagrant-devops/servicefile/the-lab-vagrant-a41e33c1ab2d.json
"


    google.image_family = "ubuntu-2004-lts"

      google.zone = "europe-west1-b"

      google.network = "vagrant-devops-test"

      google.subnetwork = "vagrant-devops-test"

    override.ssh.username = "lenny"
    override.ssh.private_key_path = "/home/lenny/.ssh/id_rsa"
  end
      config.vm.provision "ansible" do |ansible|
            ansible.playbook = "dotnetapp_playbook.yml"
  end
end
```

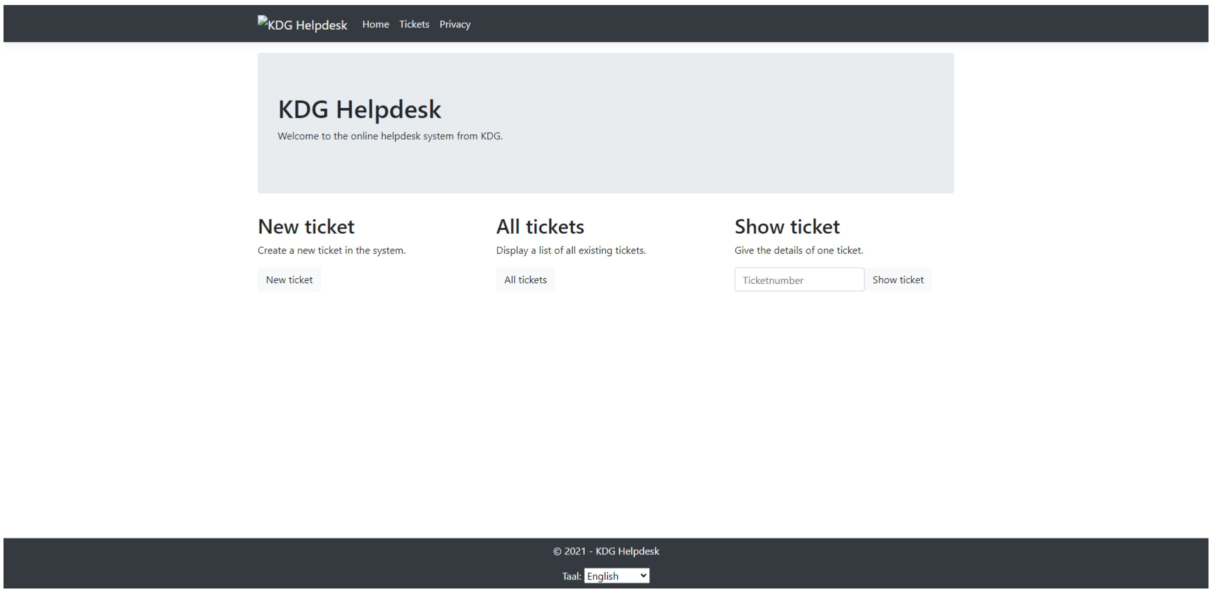
The very last step is to run "vagrant up --provider=google" to deploy.

At the end, you should get the following:



```
PLAY [Deploy web app] ***********************************************************

TASK [Gathering Facts] **********************************************************
ok: [default]

TASK [apt: update & upgrade] ****************************************************
changed: [default]

TASK [install docker] ***********************************************************
changed: [default]

TASK [pull code from gitlab using git & an acces token] *************************
changed: [default]

TASK [Build the docker image] ***************************************************
changed: [default]

TASK [Run dotnetapp image on host port 80] **************************************
changed: [default]

PLAY RECAP **********************************************************************
default                    : ok=6    changed=5    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

And our web app is up!


KDG Helpdesk Home Tickets Privacy

KDG Helpdesk

Welcome to the online helpdesk system from KDG.

New ticket

Create a new ticket in the system.

New ticket

All tickets

Display a list of all existing tickets.

All tickets

Show ticket

Give the details of one ticket.

Ticketnumber Show ticket

© 2021 - KDG Helpdesk

Taal: English

10.3 Creating a Vagrant box

There are 2 ways to create boxes:

- From scratch
- Based on a base box

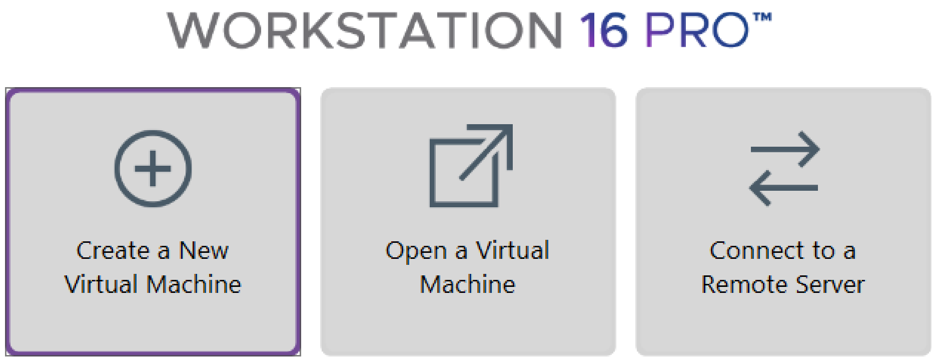
10.3.1 From scratch

We will now create our own box from scratch. We need to keep a couple things in mind when creating a base Vagrant box:

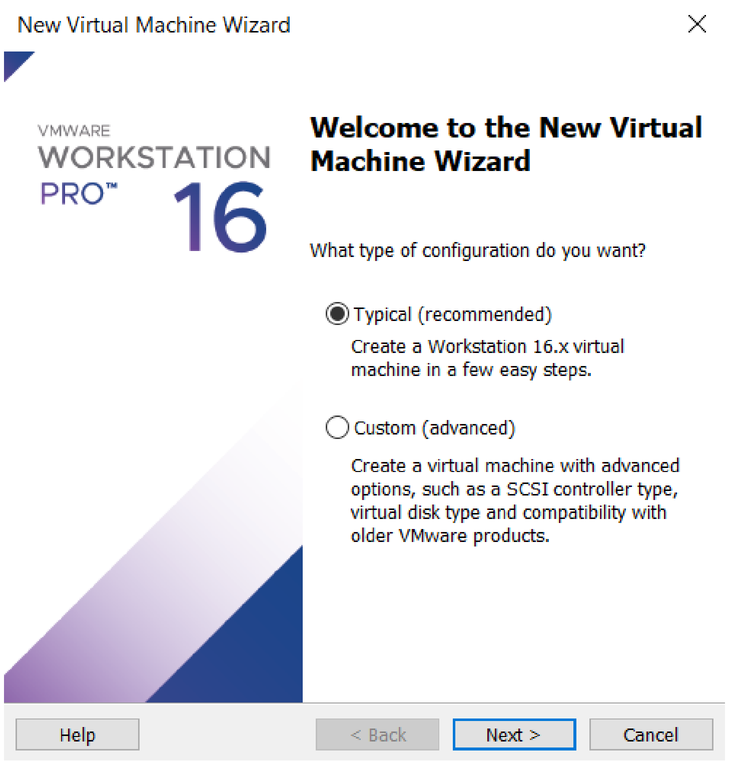
- Boxes are provider specific.
- Creating a box can be time consuming and tedious.

We will now create a Ubuntu server 20.04 base image for the VMware provider. In this example, we will be using VMware Desktop to create the virtual machine of which we will base our box of.

10.3.1.1 Virtual machine creation



We choose "Create a new Virtual Machine"



New Virtual Machine Wizard

VMWARE WORKSTATION PRO™ 16

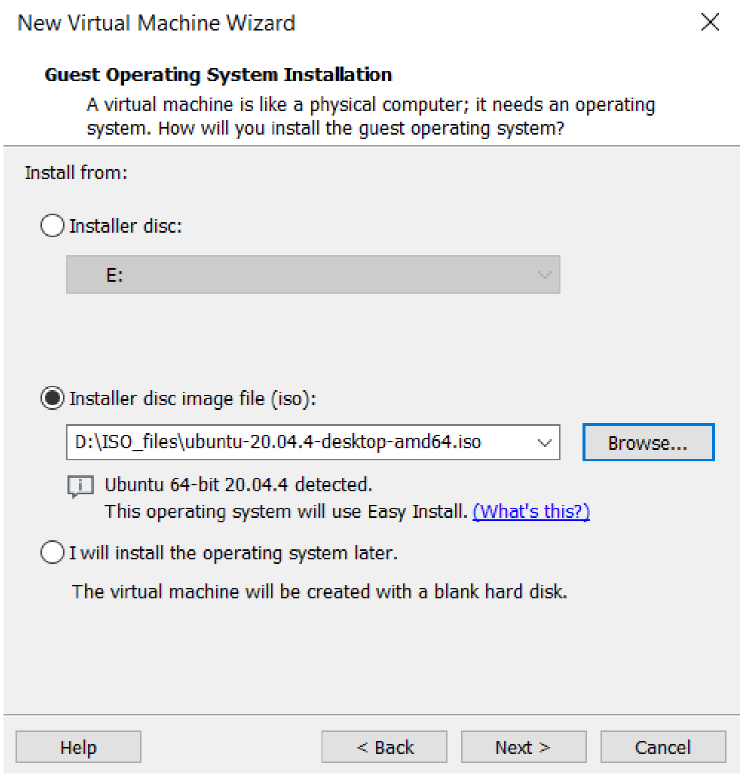
Welcome to the New Virtual Machine Wizard

What type of configuration do you want?

[✓ Typical (recommended)]
Create a Workstation 16.x virtual machine in a few easy steps.

Custom (advanced)
Create a virtual machine with advanced options, such as a SCSI controller type, virtual disk type and compatibility with older VMware products.

We are going to do a typical installation.



New Virtual Machine Wizard

Guest Operating System Installation
A virtual machine is like a physical computer; it needs an operating system. How will you install the guest operating system?

Install from:

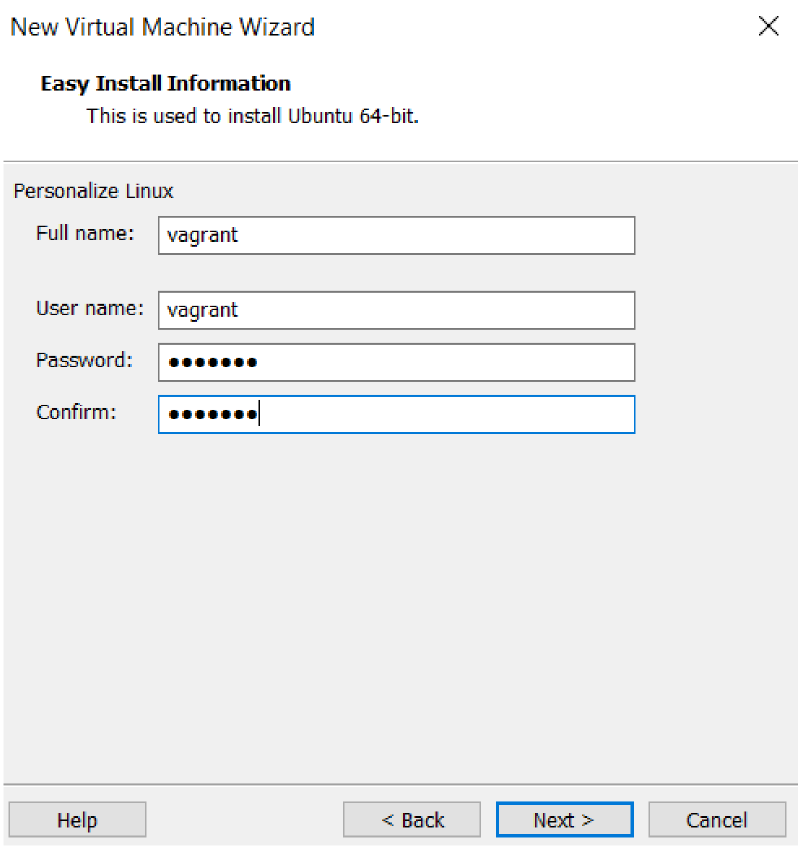
Installer disc:
E:

[✓ Installer disc image file (iso):]
D:\ISO_files\ubuntu-20.04.4-desktop-amd64.iso

Ubuntu 64-bit 20.04.4 detected.
This operating system will use Easy Install. (What's this?)

I will install the operating system later.
The virtual machine will be created with a blank hard disk.

We then choose our installer disc image. We use a Ubuntu server 20.04 iso file in this example.

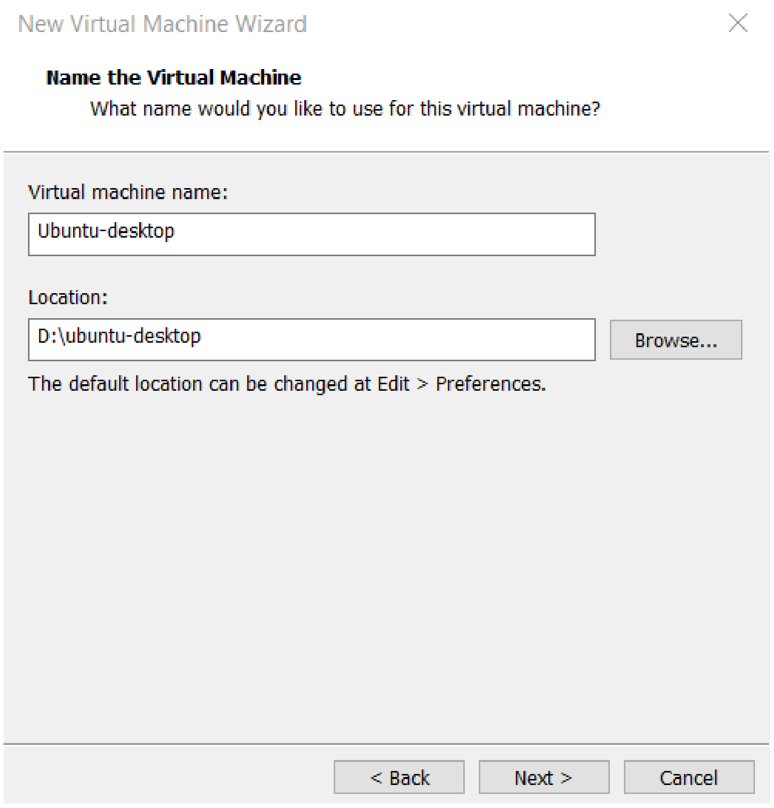

New Virtual Machine Wizard

Easy Install Information
This is used to install Ubuntu 64-bit.

Personalize Linux

Full name: vagrant
User name: vagrant
Password: •••••••
Confirm: •••••••

We then fill in our username and password. Vagrant expects a "vagrant" user that has "vagrant" as password.


New Virtual Machine Wizard

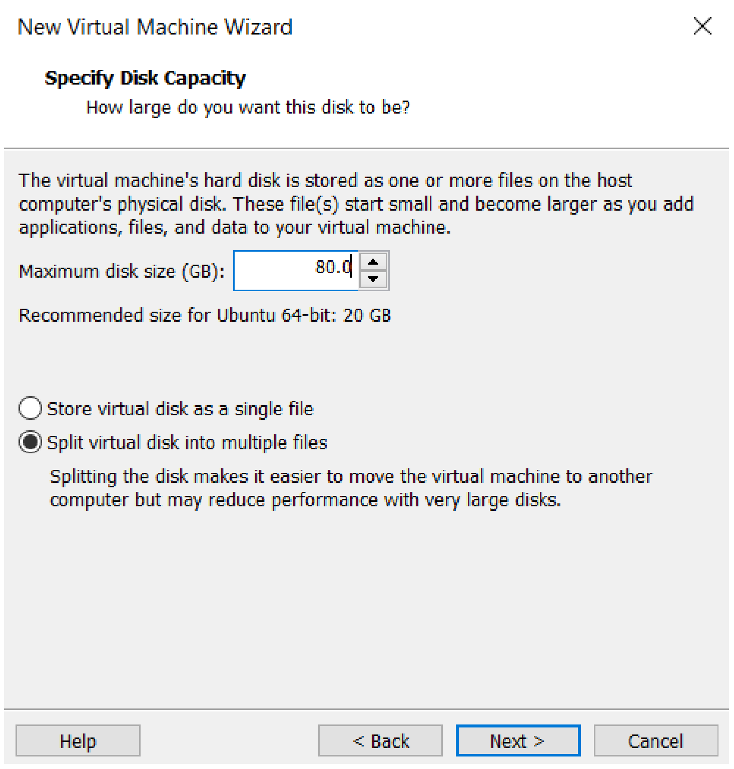
Name the Virtual Machine
What name would you like to use for this virtual machine?

Virtual machine name: Ubuntu-desktop

Location: D:\ubuntu-desktop

The default location can be changed at Edit > Preferences.

We then specify the Virtual Machine location and name.


New Virtual Machine Wizard

Specify Disk Capacity
How large do you want this disk to be?

The virtual machine's hard disk is stored as one or more files on the host computer's physical disk. These file(s) start small and become larger as you add applications, files, and data to your virtual machine.

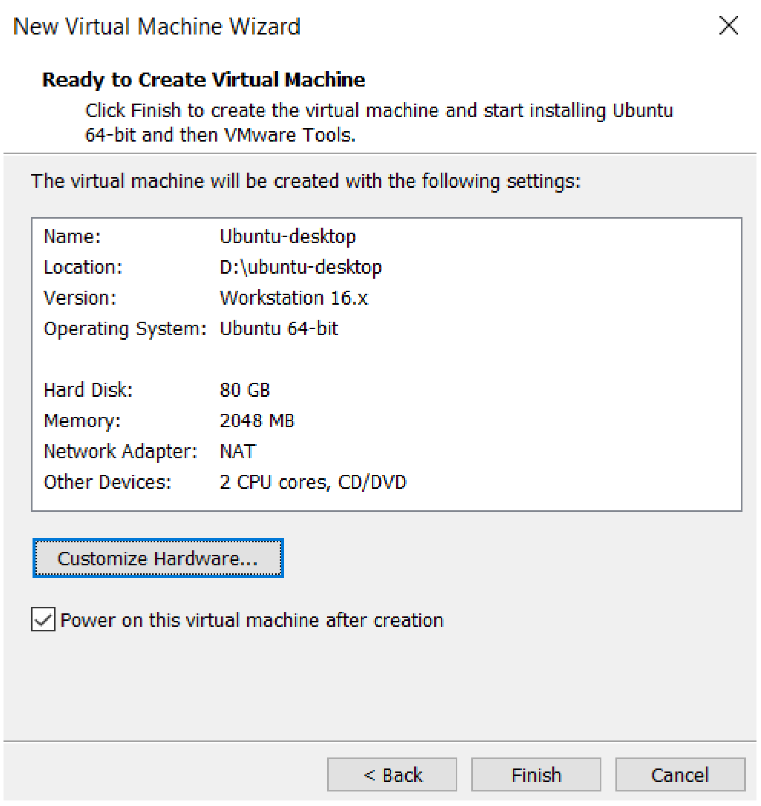
Maximum disk size (GB): 80.0

Recommended size for Ubuntu 64-bit: 20 GB

( ) Store virtual disk as a single file
(●) Split virtual disk into multiple files

Splitting the disk makes it easier to move the virtual machine to another computer but may reduce performance with very large disks.

We then specify the maximum disk size. It is important to set the maximum disk size high enough.


New Virtual Machine Wizard

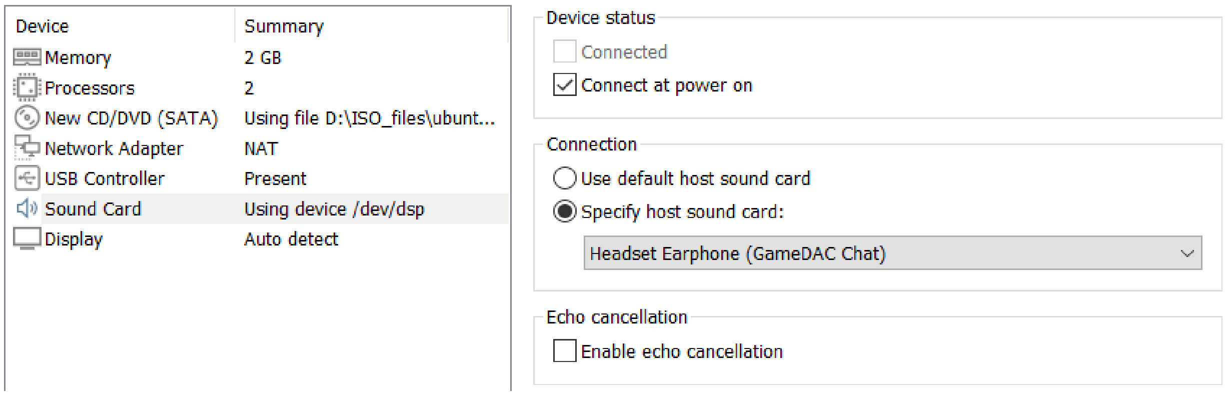
Ready to Create Virtual Machine
Click Finish to create the virtual machine and start installing Ubuntu 64-bit and then VMware Tools.

The virtual machine will be created with the following settings:

Name:	Ubuntu-desktop
Location:	D:\ubuntu-desktop
Version:	Workstation 16.x
Operating System:	Ubuntu 64-bit
Hard Disk:	80 GB
Memory:	2048 MB
Network Adapter:	NAT
Other Devices:	2 CPU cores, CD/DVD

☑ Power on this virtual machine after creation

We will then customise our Virtual Machine settings.



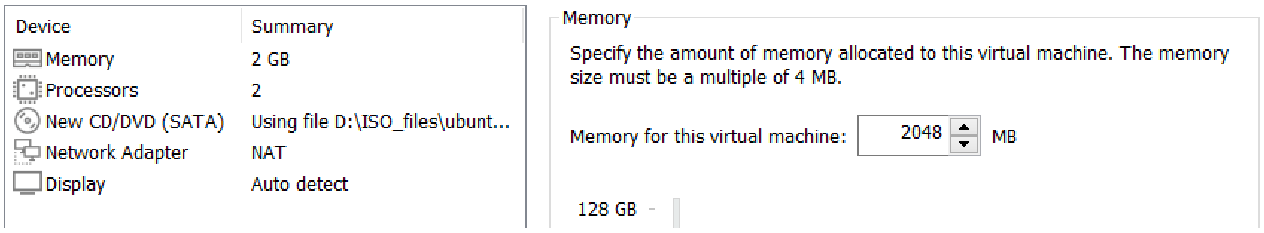
Device	Summary
Memory	2 GB
Processors	2
New CD/DVD (SATA)	Using file D:\ISO_files\ubunt...
Network Adapter	NAT
USB Controller	Present
Sound Card	Using device /dev/dsp
Display	Auto detect

Device status
[ ] Connected
[✓] Connect at power on

Connection
( ) Use default host sound card
(●) Specify host sound card:
Headset Earphone (GameDAC Chat)

Echo cancellation
[ ] Enable echo cancellation

We lower the allocated RAM to 2GB and remove all the unnecessary peripherals. These can be sound cards, USB controllers, …. We can later add peripherals in our Vagrantfile when we need them.

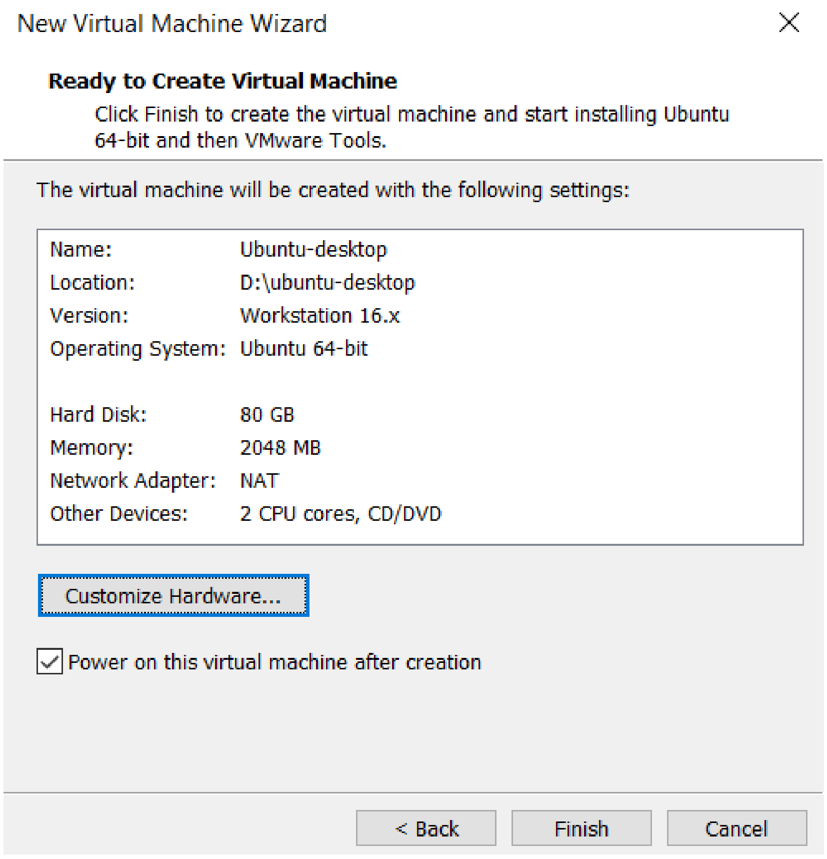


Device	Summary
Memory	2 GB
Processors	2
New CD/DVD (SATA)	Using file D:\ISO_files\ubunt...
Network Adapter	NAT
Display	Auto detect

Memory
Specify the amount of memory allocated to this virtual machine. The memory size must be a multiple of 4 MB.

Memory for this virtual machine: 2048 MB

This leaves us with the configuration that can be seen above.



New Virtual Machine Wizard

Ready to Create Virtual Machine
Click Finish to create the virtual machine and start installing Ubuntu 64-bit and then VMware Tools.

The virtual machine will be created with the following settings:

Name:	Ubuntu-desktop
Location:	D:\ubuntu-desktop
Version:	Workstation 16.x
Operating System:	Ubuntu 64-bit
Hard Disk:	80 GB
Memory:	2048 MB
Network Adapter:	NAT
Other Devices:	2 CPU cores, CD/DVD

[✓] Power on this virtual machine after creation

We then finish the creation of our Virtual Machine.

10.3.1.2 Software installation and configuration

After our Virtual Machine is created, we will configure it.
Some important configurations that we need to do are:

- Allowing SSH through our firewall
- Allowing passwordless sudo
- Adding the Vagrant public key to the authorised keys and changing the keys permissions
- Disable reverse DNS lookup

All of these configurations can be done by executing the following commands or adding these commands to a script and executing this script:
Note: the script only works on systems that use the apt package manager and is meant for making boxes that use the VMware desktop provider.

```
#!/bin/bash
# upgrade and update
sudo apt upgrade -y
sudo apt update -y
# Install opensss-server and vmware tools
sudo apt install -y openssh-server open-vm-tools open-vm-tools-desktop
# allow ssh firewall rule
sudo ufw allow ssh
# allow password-less sudo
echo "vagrant ALL=(ALL) NOPASSWD:ALL" | sudo tee -a /etc/sudoers
# Add Vagrant public key to authorized keys
mkdir ~/.ssh
wget
https://raw.githubusercontent.com/hashicorp/vagrant/master/keys/vagrant.
pub --output-document ./.ssh/authorized_keys
# Assign permissions to the public Vagrant key
chmod 0700 ~/.ssh && chmod 0600 ~/.ssh/authorized_keys
# disable reverse dns lookup
sudo sed -i "s/#UseDNS/UseDns/" /etc/ssh/sshd_config
# Restart the SSH service
sudo systemctl restart ssh
```

These commands are the base of necessary commands that need to be executed to allow the Vagrant box to work. Extra software can be installed but we won't be doing that in this tutorial.

10.3.2 Based on a base box (forking)

While creating your own Vagrant box from scratch gives you a lot of customization options, creating a base box isn't that easy and it might take a lot of time.

We will now create our own box based on another base box. We can do this very fast by creating a new Vagrantfile and specifying the base box on which we want to build our own box. We can then install the necessary software on our Virtual Machine by using an inline shell script or other provisioning tools such as Ansible, Chef, ... You can also SSH into your Virtual Machine and configure it manually.

You can customise your box as you like but we chose to install minikube, kubectl and Docker on our box. The example below shows a Vagrantfile in which we use an Ubuntu 20.10 box as a base box. We then install kubectl, minikube v1.25.2 and Docker on our Virtual Machine.

```
Vagrant.configure("2") do |config|
  config.vm.box = "generic/ubuntu2010"
  config.vm.provider "vmware_desktop" do |v|
    v.vmx["memsize"] = "4096"
    v.vmx["numvcpus"] = "3"
  end
  config.vm.provision "shell", inline: <<-SHELL
    apt-get update -y && apt-get upgrade -y
    # Install docker.io
    apt-get install docker.io -y
    usermod -aG docker vagrant
    # Install kubectl
    apt-get install -y apt-transport-https ca-certificates curl
    curl -fsSLo /usr/share/keyrings/kubernetes-archive-keyring.gpg
https://packages.cloud.google.com/apt/doc/apt-key.gpg
    echo "deb
[signed-by=/usr/share/keyrings/kubernetes-archive-keyring.gpg]
https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee
/etc/apt/sources.list.d/kubernetes.list
    apt-get update -y
    apt-get install -y kubectl
    # Install minikube and remove the deb package
    curl -LO
https://storage.googleapis.com/minikube/releases/latest/minikube_latest_
amd64.deb
    dpkg -i minikube_latest_amd64.deb
    rm -f minikube_latest_amd64.deb
    # Reboot to re-evaluate group membership (docker group)
    reboot
  SHELL
end
```

10.3.3 Packaging boxes

10.3.3.1 Self made base box

We need to add a metadata file to the directory that contains all the files of our Virtual Machine of which we want to create a Vagrant box. The "metadata.json" must at least contain information about the provider. Our metadata.json file contains the following content:

```
{"provider": "vmware_desktop"}
```

Note: The content of the metadata.json file may differ. This metadata.json file is specifically designed for boxes that make use of the VMware Desktop provider.

We will now create an archive and compress this archive using the tar command. The second parameter is the name of our box file. The example below creates a box file that is named "basebox_ubuntu_server_20.04.box".

```
tar cvzf basebox_ubuntu_server_20.04.box ./*
```

10.3.3.2 Based on a base box (forking)

We need a running Vagrant Virtual Machine when we want to create our own fork of a box. We can use the following command to start our Virtual Machine.

```
vagrant up
```

After the Virtual Machine has booted and installed all the necessary software, we use the following command to package our box:

```
vagrant package
```

This will package our custom box and this box will be named “package.box”.

We can also specify our own name by using the following command:

```
vagrant package --output [insert name].box
```

10.3.4 Adding custom boxes to the list of local boxes

We can add our custom box to the list of local boxes by using the following command:

```
vagrant box add [name of box] [box file location]
```

In our case the command is:

```
vagrant box add my-box .\basebox_ubuntu_server_20.04.box
```

The output of this command should say something like the following this

```
==> box: Box file was not detected as metadata. Adding it directly...
==> box: Adding box 'my-box-test' (v0) for provider:
    box: Unpacking necessary files from:
file://D:/ubu-desktop/basebox_ubuntu_server_20.04.box
    box:
==> box: Successfully added box 'my-box-test' (v0) for 'vmware_desktop'!
```

We can then verify if our box was successfully added by using the following command:

```
vagrant box list
```

This gives us the following output

```
generic/ubuntu2010   (vmware_desktop, 3.6.12)
my-box               (vmware_desktop, 0)
my-box-test          (vmware_desktop, 0)
thelab/ubuserver2004 (vmware_desktop, 0)
```

10.3.5 Testing local boxes

We can test our newly created Vagrant box by creating a new Vagrantfile with our box as box image. We create a new Vagrantfile by using the following command:

```
vagrant init [my new box]
```

In our case the command is:

```
Vagrant init my-box
```

We then use the following command to start our instance using the VMware Desktop hypervisor:

```
vagrant up --provider vmware_desktop
```

We can then use the following command to check the status of Virtual Machine:

```
vagrant status
```

This gives us the following output:

```
Current machine states:

default                   running (vmware_desktop)

The VM is running. To stop this VM, you can run `vagrant halt` to
shut it down, or you can run `vagrant suspend` to simply suspend
the virtual machine. In either case, to restart it again, run
`vagrant up`.
```

10.3.6 Uploading Vagrant boxes to Vagrant Cloud

We will now go over the steps on how we can publish our own Vagrant boxes on the Vagrant Cloud. We won't go over the steps on how to create your own account because this is pretty straightforward.

We can upload our Vagrant box in 2 ways:

1. Using the CLI
2. Using the Vagrant Cloud website

Note: a Vagrant Cloud account is needed to perform these steps.

10.3.6.1 CLI

Open up your prefered CLI and use the following command to log in on the Vagrant Cloud. We then use the following command to log in onto the Vagrant Cloud:

```
vagrant cloud auth login
```

We now fill in our credentials. When everything went as planned, the next output will be shown:

```
In a moment we will ask for your username and password to HashiCorp's
Vagrant Cloud. After authenticating, we will store an access token
locally on
disk. Your login details will be transmitted over a secure connection,
and
are never stored on disk locally.

If you do not have an Vagrant Cloud account, sign up at
https://www.vagrantcloud.com
Vagrant Cloud username or email: [Your email]
Password (will be hidden):
Token description (Defaults to "Vagrant login from [Your hostname]"):
[My description]
You are now logged in.
```

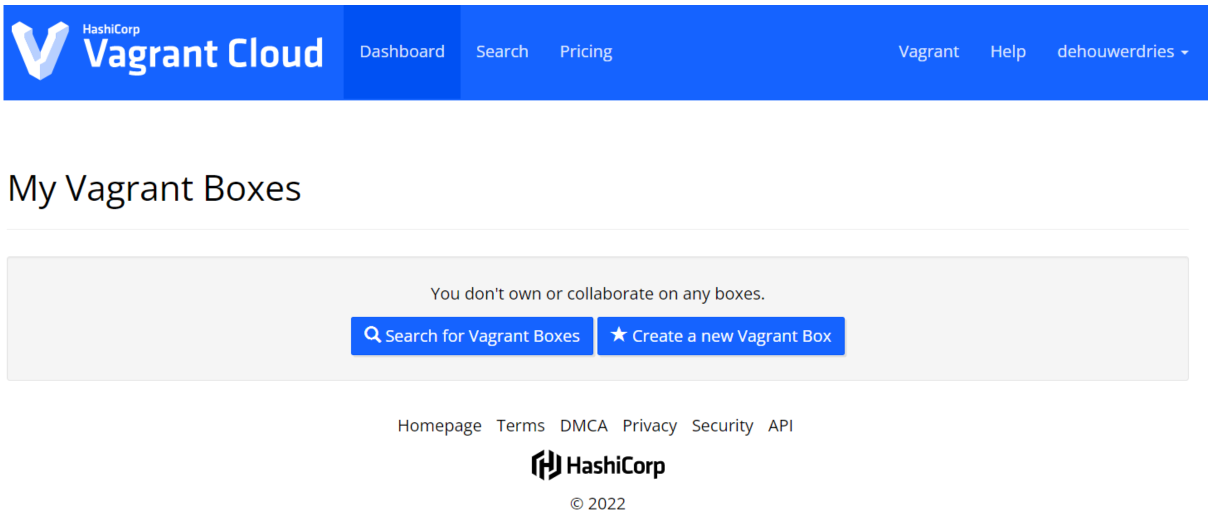
We can then publish our box on the Vagrant Cloud by using the following command:

```
vagrant cloud publish [username]/[box name] [version] [provider] [box
file location] --description [description] --version-description
[version description] --release
```

After our box is done uploading, we can go to the Vagrant Cloud website and check our box out.

10.3.6.2 Vagrant cloud website

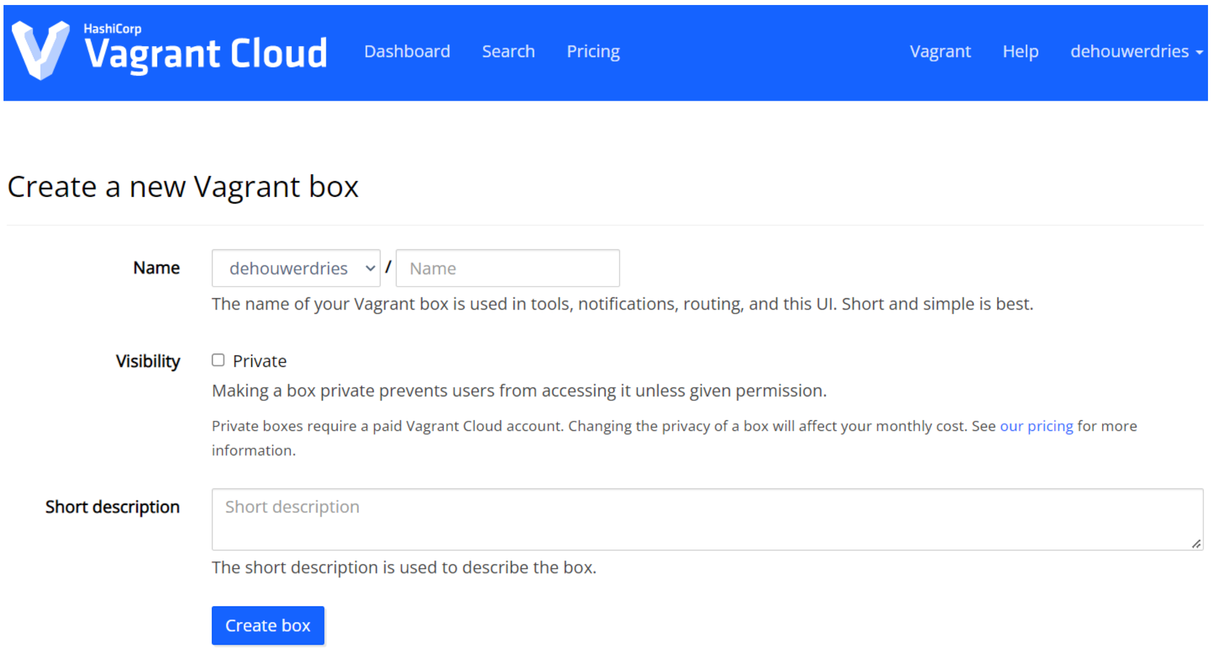
We log in after creating our account.



My Vagrant Boxes

You don't own or collaborate on any boxes.

We click on “Create a new Vagrant Box”



Create a new Vagrant box

Name: dehouwerdries / Name

The name of your Vagrant box is used in tools, notifications, routing, and this UI. Short and simple is best.

Visibility: ☐ Private

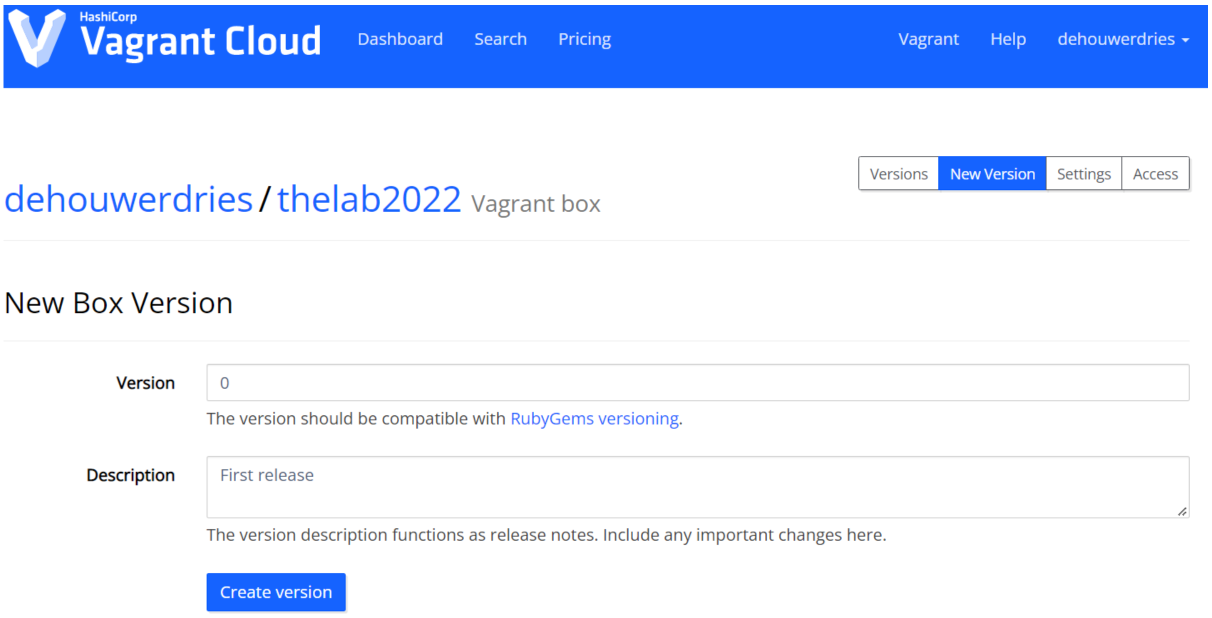
Making a box private prevents users from accessing it unless given permission.

Private boxes require a paid Vagrant Cloud account. Changing the privacy of a box will affect your monthly cost. See our pricing for more information.

Short description: Short description

The short description is used to describe the box.

We now give a name to our box, make it public or private and add a short description of our box.



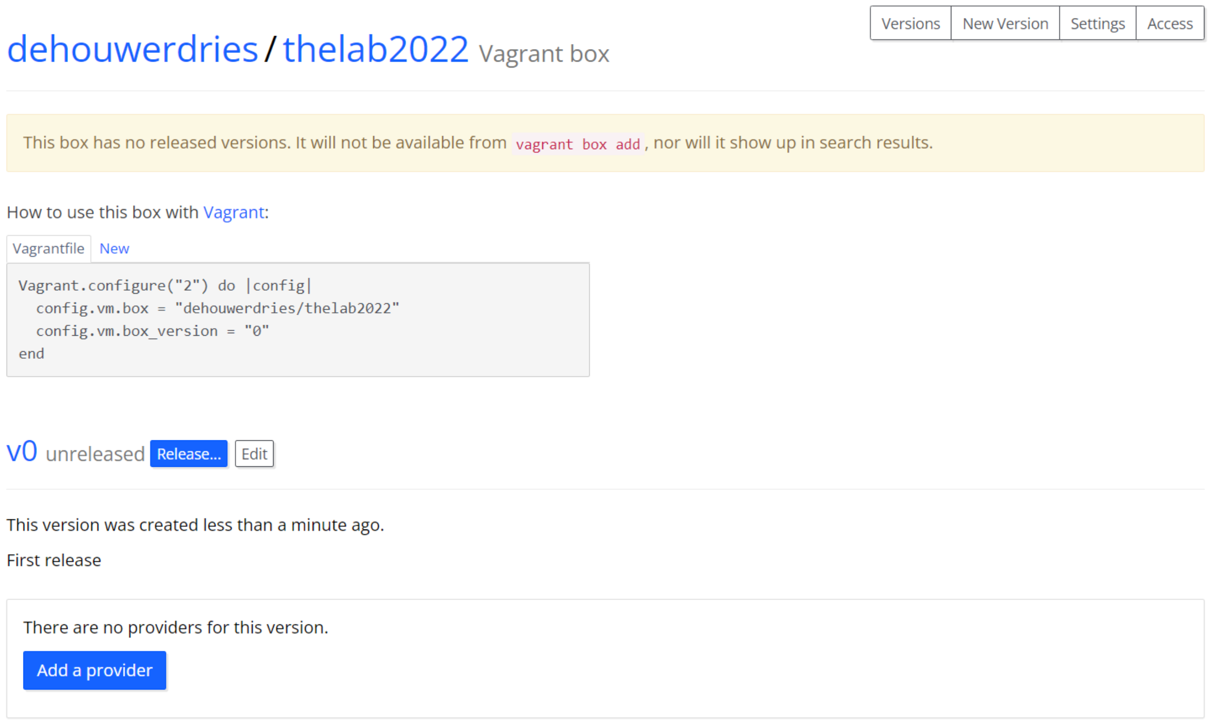
dehouwerdries / thelab2022 Vagrant box

New Box Version

Version	0
	The version should be compatible with RubyGems versioning.
Description	First release
	The version description functions as release notes. Include any important changes here.

Create version

We then add a version number and a release description.



dehouwerdries / thelab2022 Vagrant box

This box has no released versions. It will not be available from `vagrant box add`, nor will it show up in search results.

How to use this box with Vagrant:

Vagrantfile New

```
Vagrant.configure("2") do |config|
  config.vm.box = "dehouwerdries/thelab2022"
  config.vm.box_version = "0"
end
```

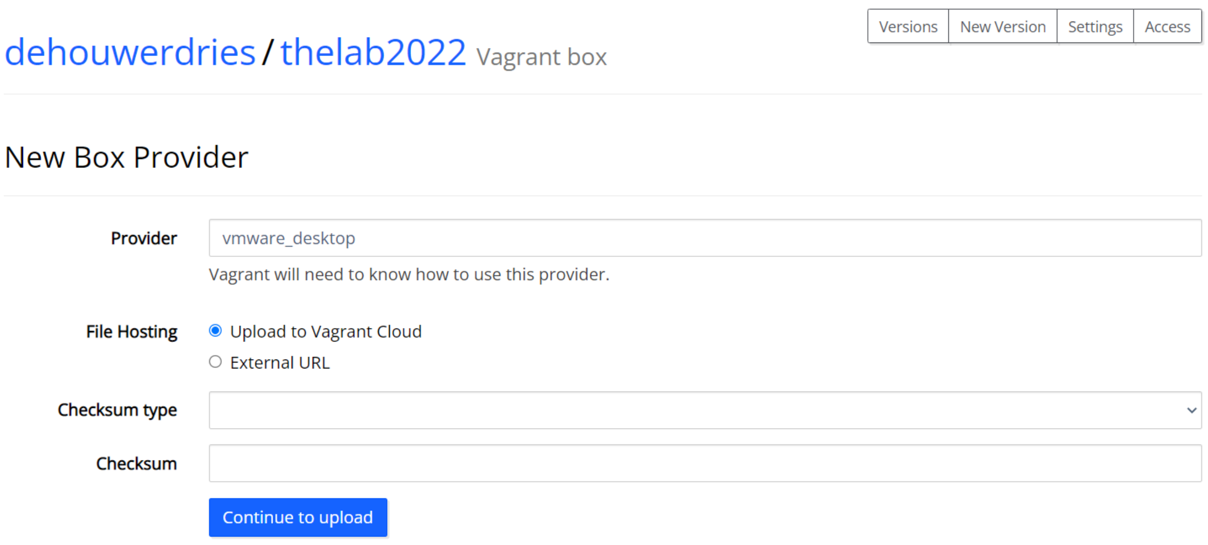
v0 unreleased

This version was created less than a minute ago.

First release

There are no providers for this version.

We then add a provider.

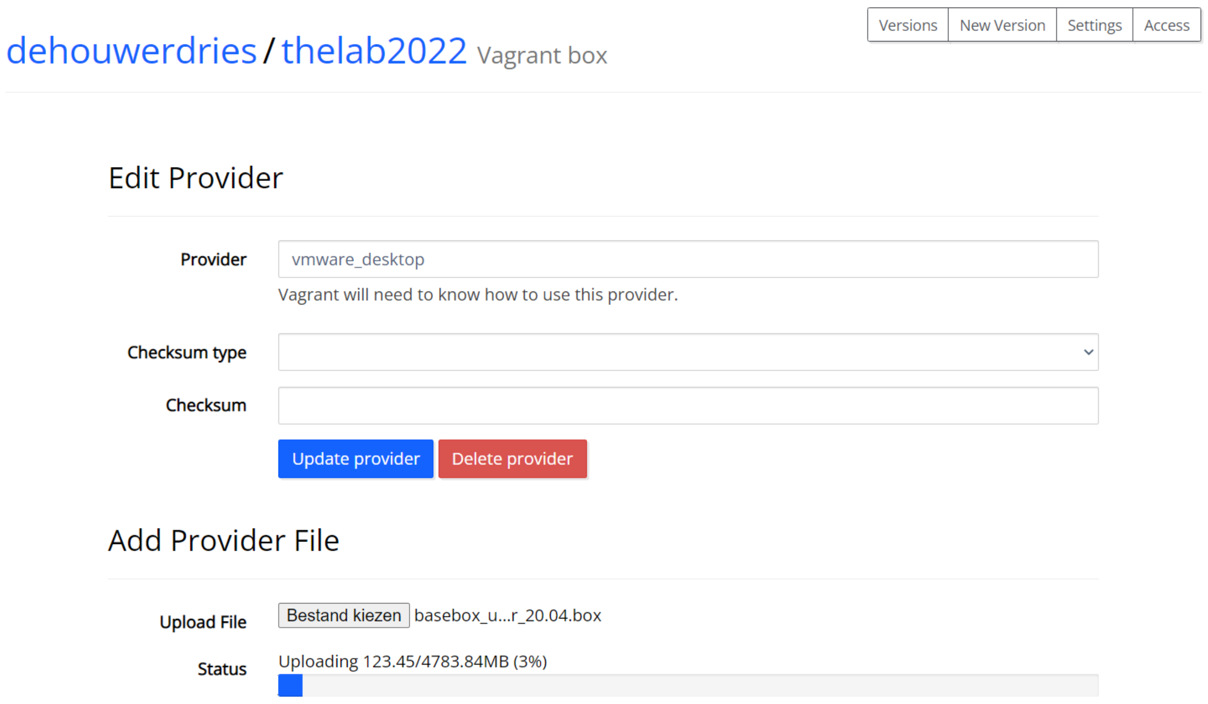


dehouwerdries / thelab2022 Vagrant box

New Box Provider

Provider	vmware_desktop Vagrant will need to know how to use this provider.
File Hosting	● Upload to Vagrant Cloud ○ External URL
Checksum type	
Checksum	

We then choose a provider. We can upload our box to the Vagrant Cloud or host it externally. We will host it on the Vagrant Cloud.



dehouwerdries / thelab2022 Vagrant box

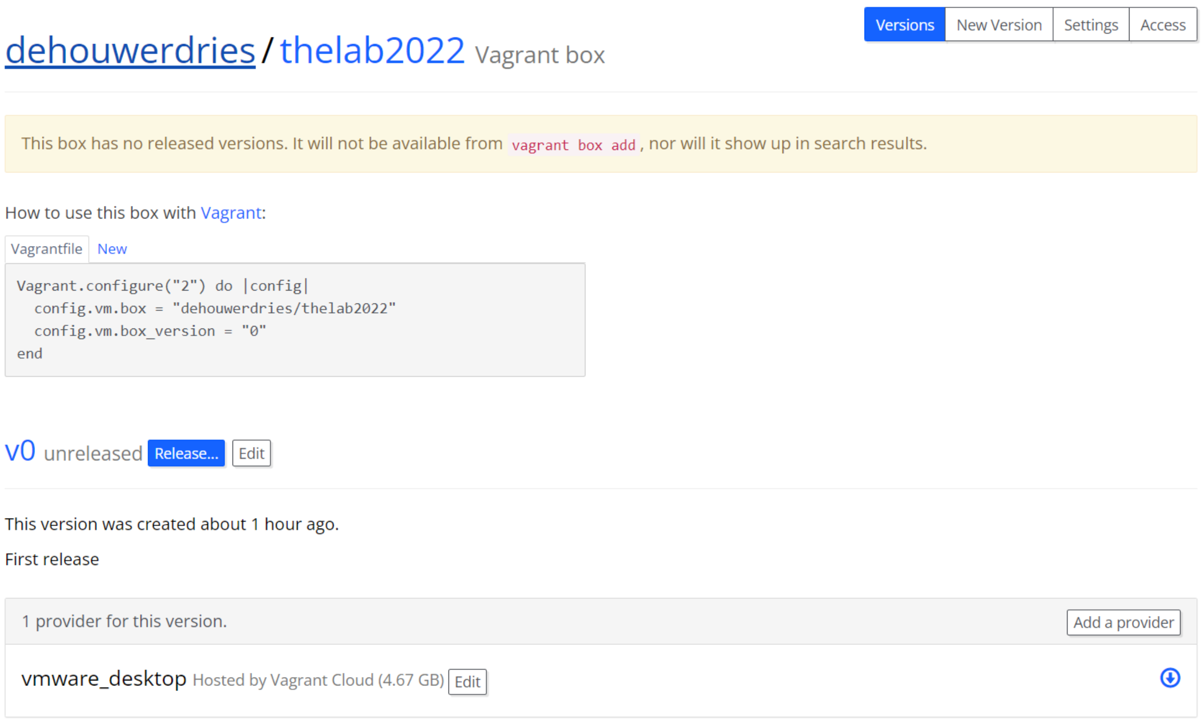
Edit Provider

Provider	vmware_desktop Vagrant will need to know how to use this provider.
Checksum type	
Checksum	

Add Provider File

Upload File	Bestand kiezen basebox_u...r_20.04.box
Status	Uploading 123.45/4783.84MB (3%)

We then add a provider file by clicking on the “Choose file” button. After our upload is completed, we click on our box repository name.


Versions | New Version | Settings | Access

dehouwerdries / thelab2022 Vagrant box

This box has no released versions. It will not be available from `vagrant box add`, nor will it show up in search results.

How to use this box with Vagrant:

Vagrantfile | New

```
Vagrant.configure("2") do |config|
  config.vm.box = "dehouwerdries/thelab2022"
  config.vm.box_version = "0"
end
```

v0 unreleased Release... Edit

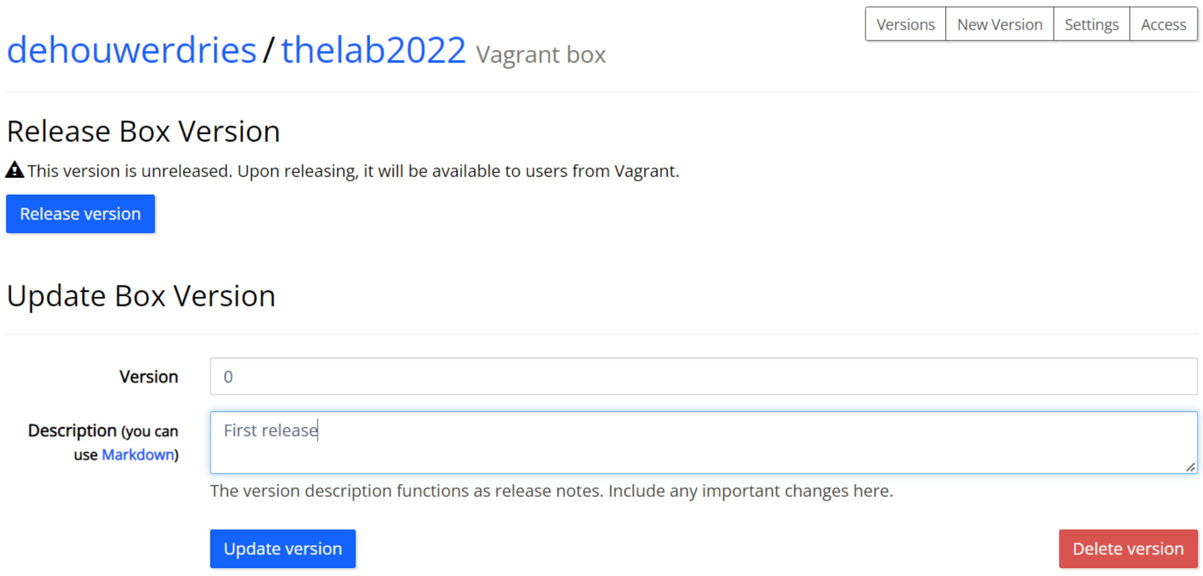
This version was created about 1 hour ago.

First release

1 provider for this version. Add a provider

vmware_desktop Hosted by Vagrant Cloud (4.67 GB) Edit

We then click on the "release" button.


Versions | New Version | Settings | Access

dehouwerdries / thelab2022 Vagrant box

Release Box Version

⚠ This version is unreleased. Upon releasing, it will be available to users from Vagrant.

Release version

Update Box Version

Version	0
Description (you can use Markdown)	First release

The version description functions as release notes. Include any important changes here.

Update version | Delete version

We then click on the "Release version" button to release our Vagrant box.
We can now check out our Vagrant box on the Vagrant cloud.

11 Conclusion

We've been able to experiment with Vagrant a lot now and the first thing that is apparent to us is that it's simply outdated. It was first released in 2010 and granted, it still gets semi-regular updates (last one having been 4 months ago at the time of writing this) which are probably more security focused than anything.

Locally Vagrant is still useful. With support for the most popular hypervisors such as VirtualBox and VMWare and the ability to provision your instances natively you can easily test your applications locally with a properly configured Vagrantfile and provisioner. With the networking options you can easily see if the network design is up to the task. One negative point about Vagrant is that all Virtual Machine instances are created one by one. This can make the setup process of multiple instances take a long time but it is still faster than setting up all these virtual machines by hand.

There are a lot of pre configured boxes available on the Vagrant Cloud. It is also possible to create your own base boxes but this can be a tedious process and can take a long time. Forking a base box, configuring and installing the necessary software on it and then creating your own box from this fork is easier and faster when we compare it to creating your own base box.

The built-in provisioning tools are really useful and are great for testing purposes. It also takes away the tedious task of setting up these provisioning tools. One negative point of these provisioning tools is that the documentation isn't always great.

One thing that is extremely clear to us is that Vagrant was **never** meant to be used in the cloud. There's no native support for it at all and the third-party plugins are hopelessly out of date - with one exception, the Google Cloud plugin.The AWS plugin's last update was **five years ago** and it doesn't work anymore. However, even the Google Cloud only has basic support because you can only manage Compute Engine instances. As explained in Vagrant for DevOps Engineers you'd still need to create everything else manually which is time consuming and tedious. We can say with certainty that if you're looking to use Vagrant mostly for usage within the Cloud, you're done for and you'd better look for alternatives. We think we can blame that on the fact that Vagrant was released in a time where cloud simply wasn't the standard and on-premise was still the most popular choice.

12 Sources

[Documentation | Vagrant by HashiCorp (vagrantup.com)]
[GitHub - mitchellh/vagrant-google: Vagrant provider for GCE.]
[GitHub - mitchellh/vagrant-aws: Use Vagrant to manage your EC2 and VPC instances.]
[GitHub - Azure/vagrant-azure: Enable Vagrant to manage virtual machines in Microsoft Azure]
[Our Git repository that contains all code]