Custom Images for Deployment

Mutability

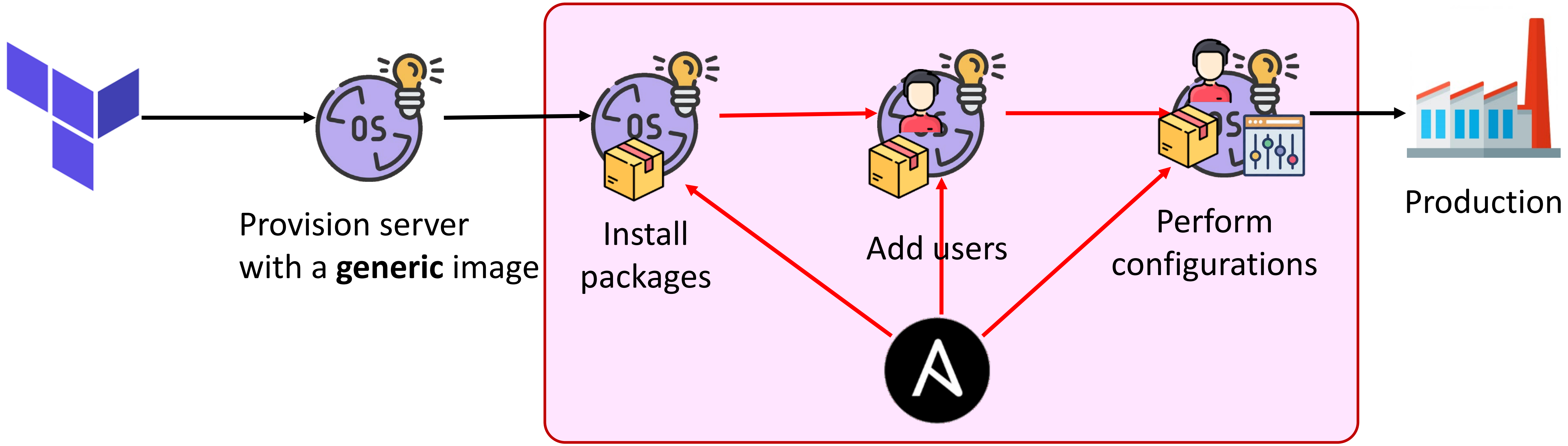


Image is mutated before (or in) production

Immutability

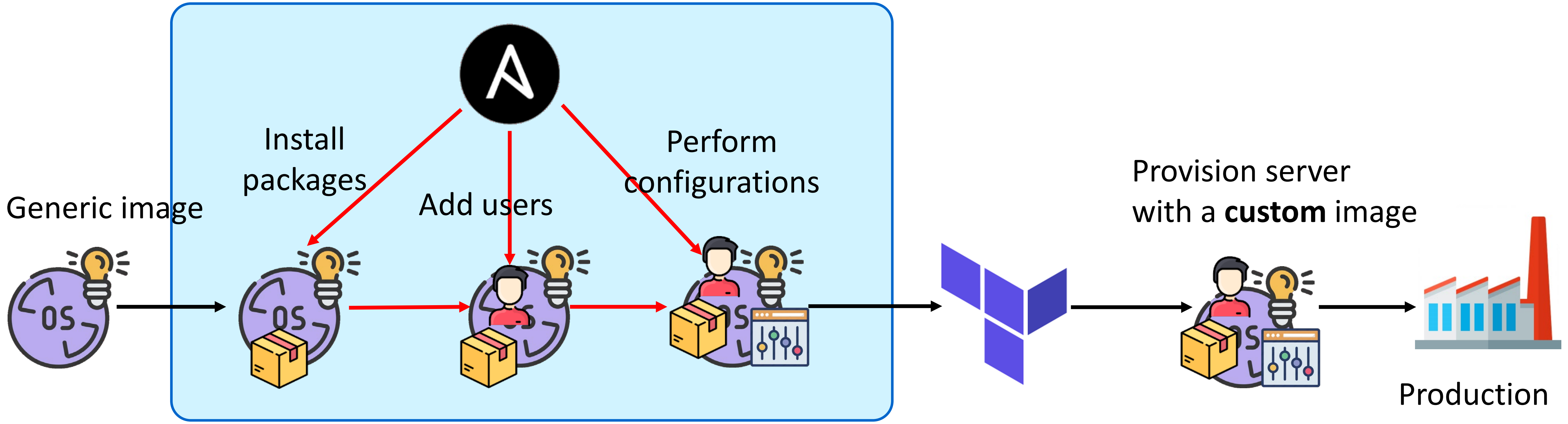
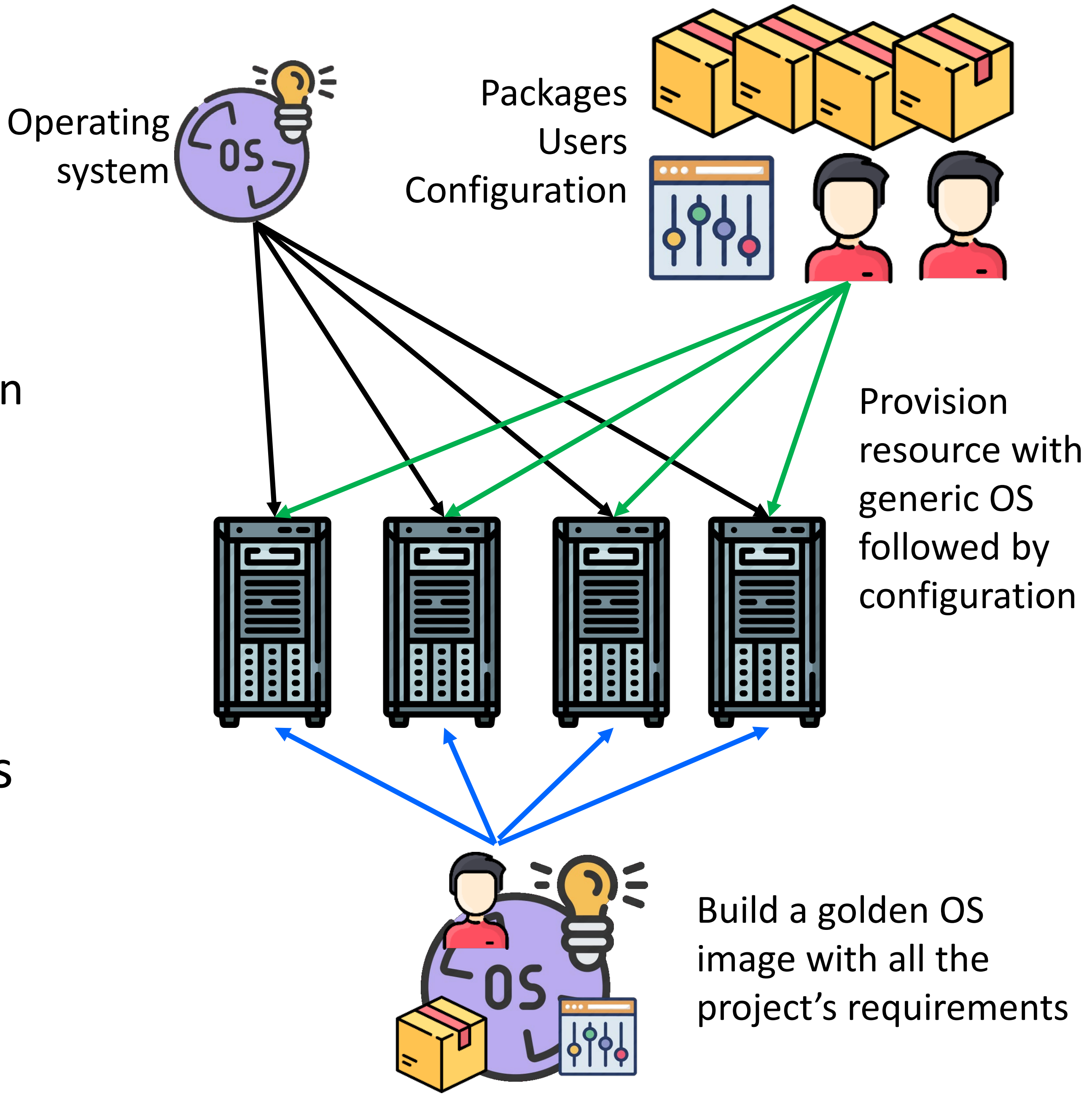


Image is mutated before provisioning

Golden Images

- Custom image can save time
  - Standardize a common set of packages for the project/organization
  - Build the image once, deploy many times vs deploy a generic image and use Ansible to configure image
  - Autoscaling a server group where each server must have the required packages, configurations and software
- Tools for building OS based images
  - Packer, Vagrant, VM platforms (VMWare, VirtualBox)
- Container based images
  - Docker, runc, cri-o


Operating system
OS
Packages
Users
Configuration
Provision resource with generic OS followed by configuration
OS
Build a golden OS image with all the project's requirements

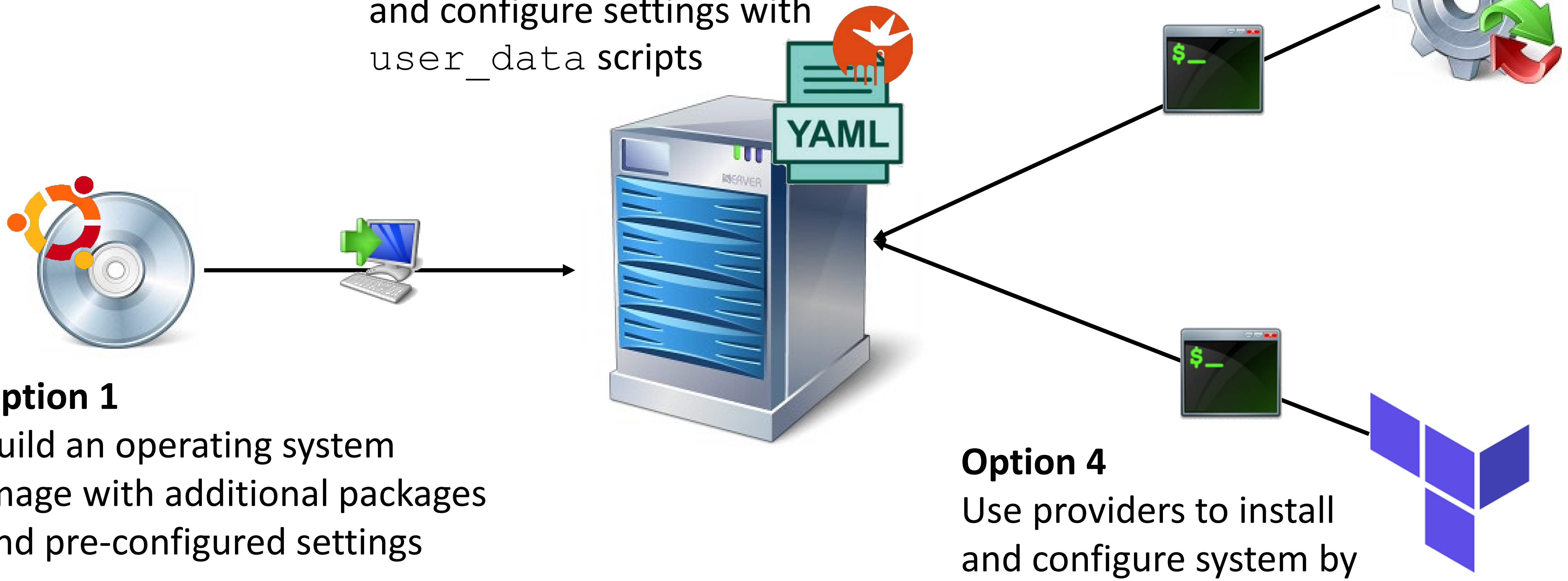
Configuration Options

Option 1
Build an operating system image with additional packages and pre-configured settings

Option 2
Install additional packages and configure settings with `user_data` scripts

Option 3
Use configuration tools to install and configure system

Option 4
Use providers to install and configure system by ‘sshing’ into the system



Packer

- Tool for building golden (custom) OS images
- Builds images for multiple platform and cloud providers
- Use cases
  - Golden image for your project, consistent and immutable
  - Environment parity for development, testing and deployment
  - Speed in launching new instances
- Packer describes, in a configuration file, how to create these golden images
  - From existing image
  - The resulting image is saved on the cloud provider

Initializing a Packer Project

```
config.pkr.hcl
packer {
  required_plugins {
    digitalocean = {
      source = “github.com/hashicorp/digitalocean”
      version = “>= 1.0.0”
    }
  }
}
```

Run the init command once, at the start of the project to download the providers

```
packer init config.pkr.hcl
```

Builder

- Packer file has the same format as Terraform
  - Use HashiCorp Configuration Language (HCL)
- Packer file consist of the following sections
  - Builders
  - Provisioners
  - Post-processors

- Builders are responsible for creating the custom images
- Can have multiple builders in a build process
  - Each builder produces an image for a particular platform
  - Eg. EC2, Droplet, VirtualBox, etc
- List of builders
  - https://www.packer.io/docs/builders

Declaring Variables

```
variables.pkr.hcl

variable DO_token {
  type = string
  sensitive = true
  value = env("MY_TOKEN")
}

variable droplet {
  type = object({...})
  description = "Droplet spec"
}
```

- Uses the same syntax as Terraform to define variables
  - Variables have to be declared before they are used

Get value for environment variable

Example - Building an Image

droplet.pkr.hcl

```
source digitalocean mydroplet {
  api_token = var.DO_TOKEN
  region = var.region
  size = var.droplet.size
  image = var.droplet.image
  snapshot_name = “mydroplet”
  ssh_username = “root”
}

source amazon-ebs myami { ... }

build {
  sources = [
    “source.digitalocean.mydroplet”,
    “source.amazon-ebs.myami”
  ]
}
```

Source image configurations

What sources to build

Note: this image has no customization

Sourcing Variables

Use Packer convention to set variable

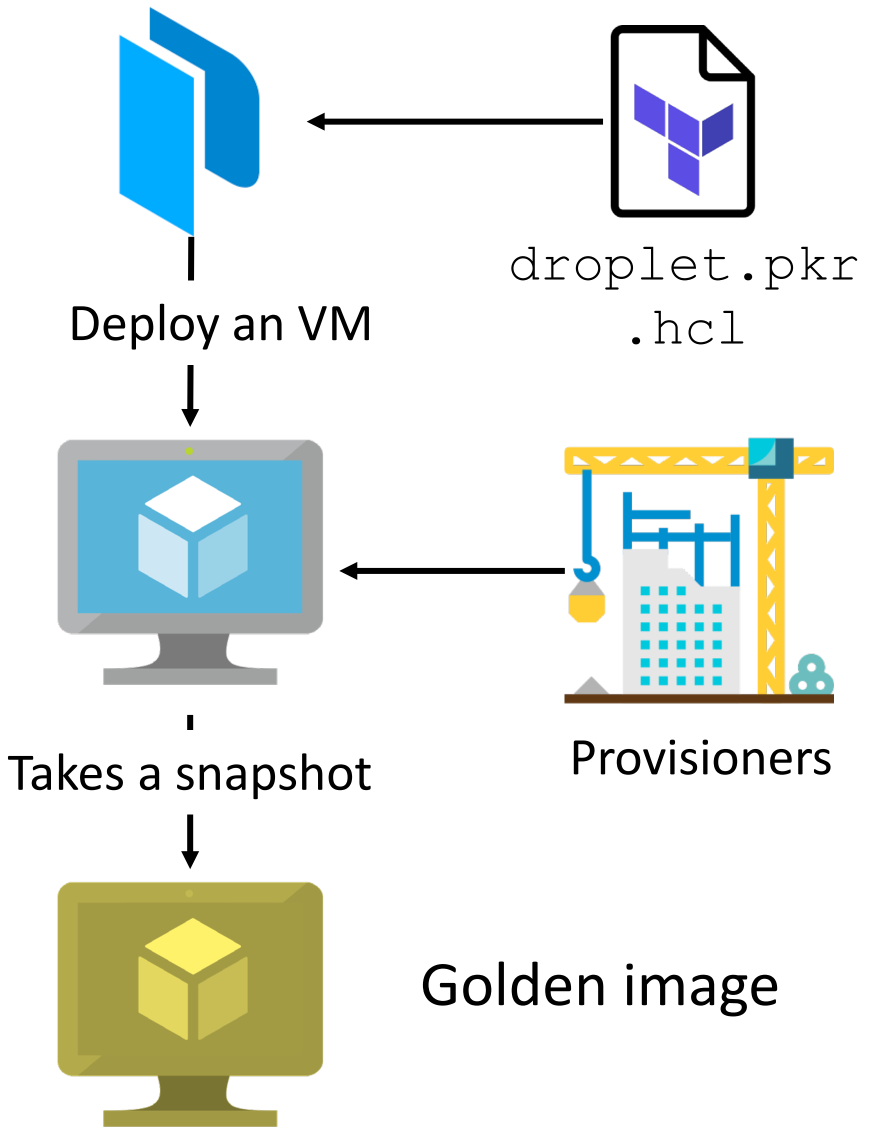
Get value for environment variable

```
export PKR_VAR_region="sgp1"

variables.pkrvars.hcl

DO_TOKEN = env("MY_TOKEN")
droplet = {
  image = "ubuntu-20-04-x64"
  size = "s-1vcpu-1gb"
}

packer build \
  -var-file=variables.pkrvars.hcl \
  droplet.pkr.hcl
packer build \
  -var-file=variables.pkrvars.hcl .
```

- Uses the same syntax as Terraform to define variables
  - Variables have to be declared before they are used
- Values can be sourced from
  - `default` argument in declaration
  - Variable files passed to the builder with `-var-file` option
    - File must have `.pkrvars.hcl` suffix
  - Environment variables prefixed with `PKR_VAR_`
  - Read with the `env()` function

Creating Golden Images

- Packer provisions a virtual machine on the cloud provider
  - Uses the provided configuration for this purpose
  - Eg. on DigitalOcean uses the API key, droplet, size and region to create a droplet
- Once the droplet is created, Packer will generate a temporary SSH keypair
- Packer will use the SSH keys to customize this image with provisioners
  - Need to configure SSH user
- When the customization is completed, Packer will take a snapshot of this image
  - This is the golden image



Provisioners

- Like Terraform provisioners, used to customize an image
- May different types of provisioners
  - File - uploads files from the local machine into the VM
  - Shell - executes shell commands. There are 2 types of shell provisioner
    - `shell-local` - run shell command on the local machine
    - `shell` - run shell commands on the VM. Script files can be uploaded to the VM with file provisioner before running this provisioner
  - Ansible - executes playbooks. There are 2 types of Ansible provisioner
    - `ansible-local` - run Ansible playbooks in the VM. Ansible must be available
    - `ansible` - run Ansible playbooks on the local machine targeting the VM
    - Ansible provisioner will automatically generate an inventory file
    - Ansible is implemented as a plugin, has to be downloaded during initialisation
- See https://www.packer.io/docs/provisioners

Example - File and Shell Provisioner

```
build {
  sources = [ "source.digitalocean.mydroplet" ]

  provisioner file {
    source = "setup.sh"
    destination = "/tmp/"
  }

  provisioner shell {
    inline = [
      "chmod a+x /tmp/setup.sh",
      "/tmp/setup.sh"
    ]
  }
}
```

setup.sh

```
#!/usr/bin/env bash
apt update
apt install -y nginx
systemctl enable nginx
systemctl start nginx

ufw default allow outgoing
ufw allow ssh
ufw allow http
ufw allow https
ufw enable
```

Install Ansible Plugin

```
config.pkr.hcl
packer {
  required_plugins {
    digitalocean = {
      source = “github.com/hashicorp/digitalocean”
      version = “>= 1.0.0”
    }
    ansible = {
      version = "~> 1"
      source = "github.com/hashicorp/ansible"
    }
  }
}
```

Example - Ansible

```
build {
  sources = [ “source.digitalocean.mydroplet” ]

  provisioner ansible {
    playbook_file = “playbook.yaml”
    extra_arguments = [
      “-e”, “db_password=${var.db_password}“,
    ],
    ansible_ssh_extra_args = [
      "-oHostKeyAlgorithms=+ssh-rsa -oPubkeyAcceptedKeyTypes=+ssh-rsa"
    ]
  }
}
```

Command line arguments for `ansible-playbook`, if any

Use RSA for SSH connection

Temporary Inventory File

inventory.yaml

```
all:
  vars:
    ansible_user: root
    ansible_connection: ssh
    ansible_ssh_private_key_file: /path/to/private/key

  hosts:
    default:
      ansible_host: <IP address>
```

From `ssh_username`

Temporary key pair generated by Packer

playbook.yaml

```
- name: Configure image
  hosts: default
  ...
```

Ansible provisioner creates a host alias called `default` referring to the host that is being provisioned
Using `all` works as well

Running Package Update

- Performing a package update on Ubuntu with Ansible may fail when running the Ansible plugin
  - Auto update may have started preventing the playbook's update to run
  - Cause failure
- Retry the update and install until succeed

playbook.yaml

```
- name: Iinstall
  hosts: default
  tasks:
  - name: Install packages
    apt:
      name: nginx
      update_cache: yes
      state: latest
    register: result
    until: result.stderr == ""
    retries: 10
    delay: 10
```

Retry 10 times until there are no more errors

Using the Image

- Some Terraform providers cannot provision snapshots/VMs directly from images build by Packer
  - Eg. DigitalOcean droplet will fail if you use the image
- Alternative is to use `data` to lookup the image

Data source to lookup the image

```
data digitalocean_image mydroplet {
  name = “mydroplet”
}

resource digitalocean_droplet app {
  name = “app”
  image = data.digitalocean_image.mydroplet.id
  ...
}
```

Image name created by Packer. For DigitalOcean this is `snapshot_nane`

Set the id of image from data as the image