



Infrastructure as Code

Learning objectives

- ▶ Upon successfully completing this work, you should be able to:
 - ▶ Explain the term "Infrastructure as Code" and its benefits
 - ▶ Demonstrate an awareness of various IaC approaches
 - ▶ Explain imperative and declarative programming and when you may use one over the other
 - ▶ Explain the key benefits of Terraform over other approaches
 - ▶ Explain the key components in Terraform such as resources and provisioners
 - ▶ Explain a given piece of terraform code
 - ▶ Explain the concept and benefits of a CI/CD pipeline and general code deployment automation
 - ▶ Deploy infrastructure using Terraform

Basic infrastructure

- ▶ Creating a VM / EC2 from the portal seems so easy. But a lot goes on in the background and various defaults are used; this is ok for experimenting but not in real design:
 - ▶ A VM type and size needs to be specified
 - ▶ A VM OS image needs to be specified
 - ▶ We need a system disk for the VM image
 - ▶ We need a network within which our VM can communicate
 - ▶ VM needs a network Interface Card for network connection
 - ▶ We need firewall to allow only ports we are using
 - ▶ We need a sub-networks for efficiency and security
 - ▶ VM needs a private and public IP address
- ▶ All this gets created using defaults and you can see them in the resource group, but for automated deployment and system design control, **we** need to specify these things

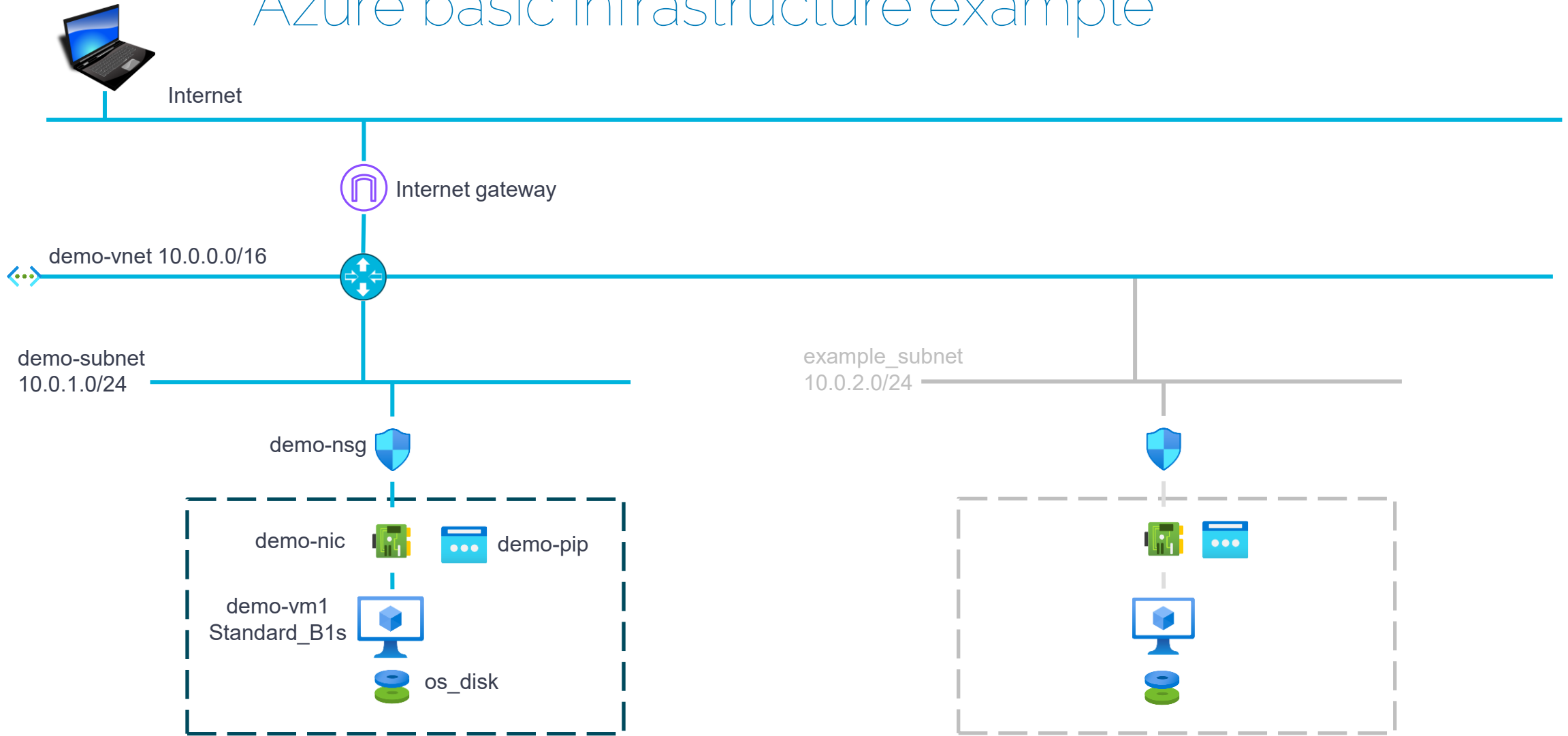
☐ Name ↑↓	Type ↑↓
☐  junk	Virtual machine
☐  junk	Network Interface
☐  junk	Network security group
☐  junk	Public IP address
☐  junk	Virtual network
☐  junk_OsDisk_1_ec2f444defcc4fa281868c12c4be...	Disk

Deleting resources

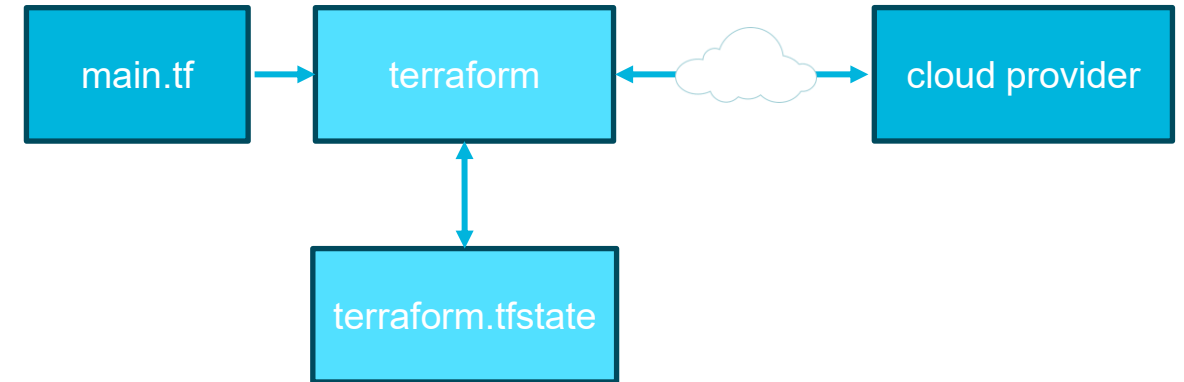
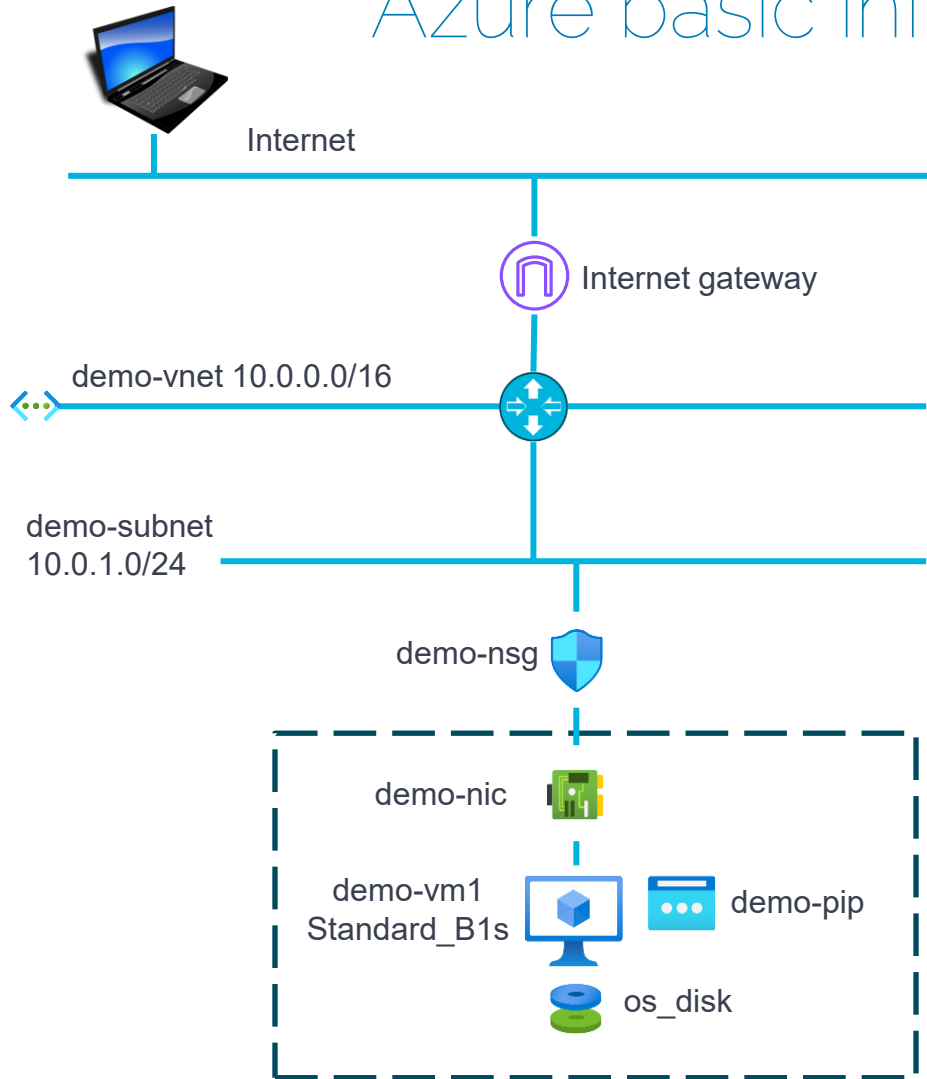
- ▶ If you choose to delete all the resources, say to save cost, then you want to recreate them again
 - ▶ Azure: delete the resource group - simply deletes all resources in the group. That may not be what you want.
 - ▶ AWS: you need to remember all the resources and manually delete each one
 - ▶ To recreate, you need to go through all the console steps again - imagine a more complex architecture
- ▶ Organisations need to automate the creation and destruction of their architecture and code deployment
- ▶ Cloud providers enable various ways of creating, managing and destroying resources programmatically
 - ▶ Software Development Kit: use your favourite programming language - e.g. call a function to create a VM
 - ▶ CLI: install the command line interface which provides various functions - e.g. to create a VM
 - ▶ Use a shell / batch script to execute command line commands imperatively
 - ▶ Use the cloud provider's built-in shell e.g. cloud shell to issue command line commands
 - ▶ Azure Resource Manager (ARM) templates for declarative IaC
 - e.g., In resource manager | Export template creates a reusable template of your infrastructure
 - ▶ AWS CloudFormation templates for declarative IaC
 - ▶ Use third party software such as Terraform which is cloud provider agnostic **declarative** approach
- ▶ I've posted a couple of videos showing CLI and template approach. We will stick to industry approach of Terraform

Videos: Azure CLI IaC and Azure templates

Azure basic infrastructure example



Azure basic infrastructure example



- ▶ **terraform init** - check which provider then download plugin
- ▶ **terraform apply**
 - ▶ Builds dependency graph
 - ▶ Checks for changes in state
 - ▶ Deletes and creates resources as required
- ▶ **terraform destroy** - all resources in state destroyed

[demos/terraform/Azure/minimal_vm/main.tf](#)

Terraform

- ▶ Just to do something relatively simple such as create a vm is messy using command line and scripts
- ▶ In industry we need more control of the infrastructure than using defaults
- ▶ We need reliable repeatable and maintainable automation
- ▶ Use something like Terraform - which is probably the most popular approach (not perfect but what is)
 - ▶ Declarative so you don't need to worry about the creation order (usually)
 - ▶ Don't need to wait for resource creation to move on e.g. to get IP as it is declarative - it will do it
 - ▶ Only one language to learn for any cloud provider - HashiCorp Configuration Language (HCL)
- ▶ Terraform is a module in its own right, so just an intro here due to its importance in industry
- ▶ None of us can remember all the commands of all the tools we use so use documentation
 - ▶ AWS: <https://registry.terraform.io/providers/hashicorp/aws/latest/docs>
 - ▶ Azure: <https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs>
- ▶ Terraform app is available at:
 - ▶ <https://developer.hashicorp.com/terraform/install>
 - ▶ Choose AMD64 - latest version for Windows

Initial key components

- ▶ **Configuration file**
 - ▶ One or more .tf files containing Terraform code. Primary file typically called main.tf
 - ▶ Used to define terraform "things" like vm, nsg or whatever
 - ▶ This is a declarative as opposed to imperative approach - we declare the outcome and let it work out how to do it
- ▶ **Provider**
 - ▶ Defines plugin to connect to provider's API to construct & manage resources based on the declarative Terraform code
- ▶ **Resource (thing)**
 - ▶ Represents resource components such as VM, Storage, Network etc
 - ▶ Consists of:
 - ▶ Resource type: e.g. "aws_instance" for an AWS ec2, "azurerm_virtual_machine" etc. These are defined by the provider's API
 - ▶ Resource name: a unique name for you to refer to in code. i.e., like a constant
 - ▶ Args and Attributes: configuration detail for a specific resource. e.g. instance_type = "t2.micro",
- ▶ **Variable**
 - ▶ Like parameters passed into the file or between blocks within the file similarish to function parameters
 - ▶ Input variables: pass in values from cli, env vars or a variables file to, say, config a resource - e.g. VM type
 - ▶ Local values: used within the code to reduce repetition - like constants
 - ▶ Output values: extract values from configurations for display or pass into other config files - a bit like function return values
 - ▶ Various types: string, number, bool, list (like array), map (like json object)

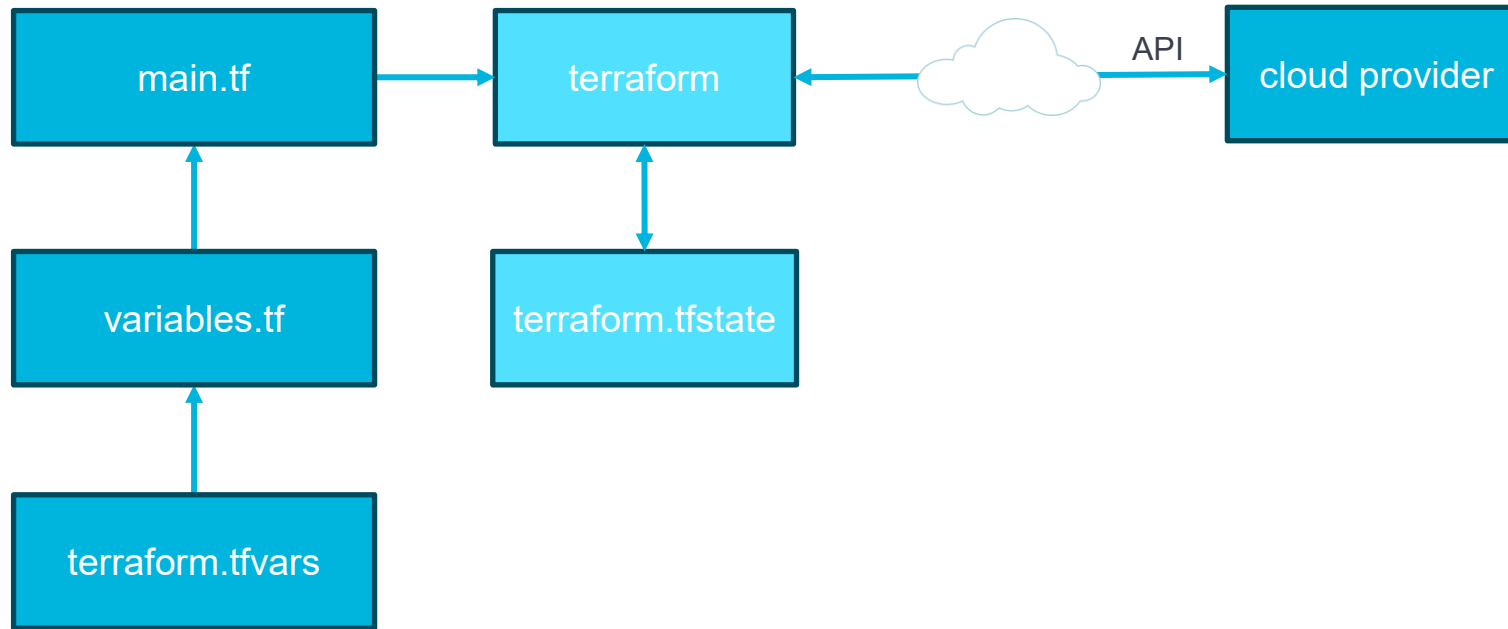
Create a VM

- ▶ It seems complicated but think about it as being like building a computer, connecting components
- ▶ Script is reusable
- ▶ Can build or destroy all the components with a single command
- ▶ Declarative, like in docker compose, so create a target state
- ▶ Terraform creates a state file that holds the exact configuration created so it can be used to delete too
 - ▶ The state file will appear once you build something and is called **terraform.tfstate**
 - ▶ Terraform refers to this when you change something - may not have to rebuild everything
- ▶ Note: For first use, switching providers, switch directories and various other scenarios, you need:
 - ▶ **terraform init**
 - ▶ The environment will usually tell you if you need to do this. It will download a provider .exe file to use as the API integration point to the cloud provider. So you will see a new folder appear: .terraform
 - ▶ A new file appears: .terraform.lock.hcl this is used to ensure that the exact version of the provider used to create infrastructure is used in the future. e.g. if a team works on the same project, this file would be shared. Like the package-lock.json file
- ▶ **terraform apply** will execute code in the main.tf file - in fact all the .tf files in the current directory
- ▶ **terraform destroy** will delete everything defined in the state file

Variables

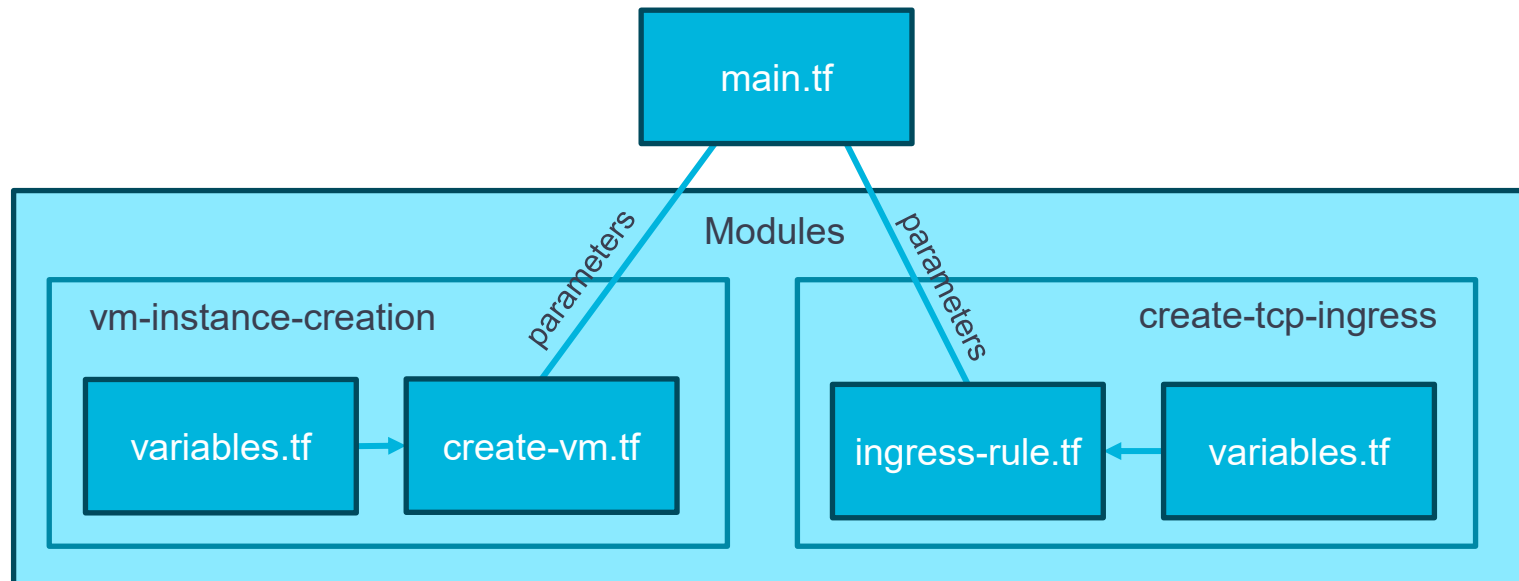
- ▶ Don't hard-code properties such as image and instance type, use in and out variables. Generally, store in variables.tf
- ▶

```
variable "machine_type" {  
  description = "Azure Virtual machine"  
  type        = string  
  default     = "Standard_B1s"  
}
```
- ▶ Variable name **machine_type**. This is initialised to Standard_B1s unless a non-default value is passed
- ▶ A variable value can be passed in from a file with terraform.tfvars extension or at the command line during apply
- ▶ This approach promotes code reuse. In terraform.tfvars:
 - ▶ `machine_type = "Standard_B2s"` This overrides the default B1s
- ▶ Here, the var instance_type is not fixed but assigned the value of machine type. If not provided, it defaults to Standard_B1s. Defaults are optional and often not desirable. Generally best to specify what is used
- ▶ You could have different .tfvars files say for dev and test, e.g. dev.tfvars, test.tfvars set to different values. Just provide the name of the file on the command line during apply
- ▶ Output variables can be added to a default file **output.tf** if required or left in the main.tf file - they all get combined during an apply



- ▶ Terraform apply: values in .tfvars are copied into variables in variables.tf
- ▶ main.tf uses the values in the variables to create the required resources
- ▶ Terraform creates a state file of resources and dependency map of creation order
- ▶ Terraform converts the state file into API calls to the cloud provider to create or destroy resources
 - ▶ Note: if a change is made and some resources are not affected, the unaffected resource definitions are ignored
- ▶ Terraform will read the output generated at the cloud end and display progress on the local machine

- ▶ For common actions, such as VM creation, code can be structured into modules - like functions
- ▶ A main.tf references the module by pointing to its location so is really just a file that references one or more modules to create a more complex infrastructure without code duplication - similar to main program referencing functions by their location
- ▶ A new main is written for the bigger project, and it will reference modules (modules folders)
- ▶ Values are passed from main to module (or module to module) in a similar way to calling functions
- ▶ Main would specify the provider and region which is inherited by modules



Terraform Provisioners

- ▶ Terraform is an infrastructure deployment application but it is also capable of deploying code
- ▶ A provisioner enables OS command execution on the client or remote machine and are added to a resource block
- ▶ local-exec provisioner
 - ▶ Execute commands on the client machine - e.g. your laptop
 - ▶ For example, pipe terraform output to a file instead of cluttering the console
- ▶ remote-exec provisioner
 - ▶ Execute system commands on the remote machine - e.g., Azure VM, AWS ec2
 - ▶ e.g. can use to install node, or docker or whatever
- ▶ file provisioner
 - ▶ Securely transfer files from the host to the remote machine - e.g. rather than using scp or winscp manually
- ▶ Note: as resource blocks are only called when creating the resource, the provisioners are only executed during resource creation. e.g. Maybe during a VM creation we want to copy a Docker install script to the remote vm and run it to install Docker. That only happens once during VM creation
- ▶ This is a useful config feature for smaller or test projects, In industry you would more likely use:
 - ▶ Ansible, Chef or Puppet

Terraform null-resource

- ▶ Provisioners are really handy in executing local or remote commands but they are only executed when the resource is being created - i.e. generally once. That's fine for installing Docker
- ▶ But say we want to automate a file copy if the file changes. E.g. a source code file or compose.yaml file, we can't unless we delete a resource and recreate it
 - ▶ Resource blocks are only executed when they are created and the provisioners are in there
 - ▶ Terraform provides a special "fudge" resource block called a **null-resource** that doesn't create a resource but acts like it does. i.e. it is called once when trying to create a resource, but as it is a null-resource, it creates nothing so doesn't affect the infrastructure, but its provisioners are executed

Terraform null resource

- ▶ When terraform apply is executed, terraform compares the current state, held in the state file, to the desired state held in .tf files and creates a plan indicating any changes it needs to make
- ▶ Changing the content of a file you want to copy to the vm is not a resource change so will not trigger the file-deployment within the resource. A null-resource can deal with this using a trigger

```
resource "null_resource" "copy_file" {
  triggers = {
    file_md5 = filemd5("path/to/local/file.txt")
  }

  provisioner "file" {
    source      = "path/to/local/file.txt"
    destination = "/home/ubuntu/file.txt"

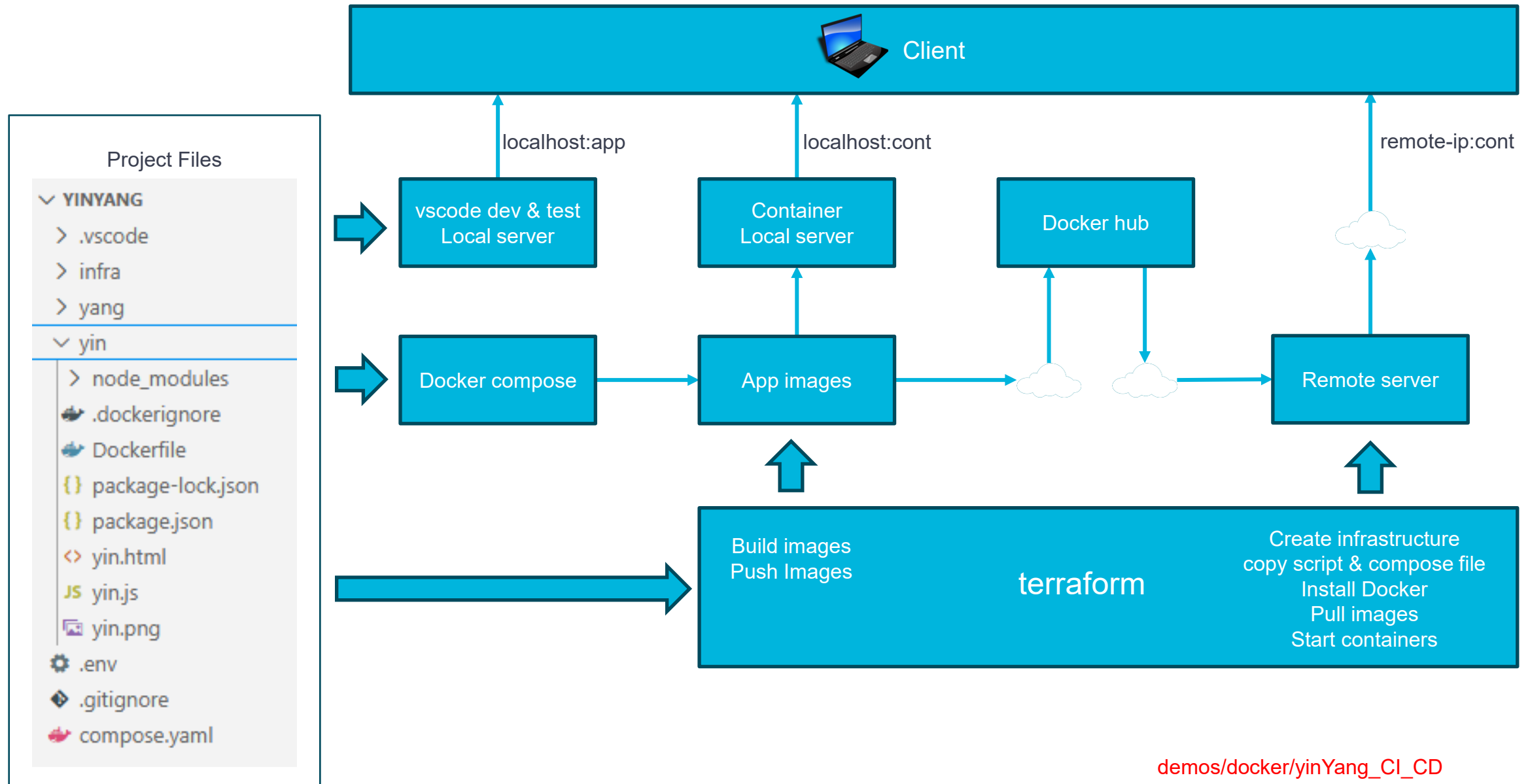
    connection {
      type      = "ssh"
      user      = "tony"
      private_key = file("path/to/private_key.pem")
      host      = azurerm_linux_virtual_machine.demo-vm.public_ip_address
    }
  }
}
```

- ▶ The resource block is triggered by some condition and executed. In this case, the file hash is compared to that in the state and if different, triggers this resource block so the file provisioner is executed

CI/CD Pipeline

- ▶ Modern organisations tend to use a CI/CD approach for software development
- ▶ Continuous integration (CI)
 - ▶ teams use gitlab, or similar, to work independently on parts of a project
 - ▶ each would debug and test their code then when ready, merge into a single project ready to test as a whole
 - ▶ tests can be automated and a popular approach is test-driven design
- ▶ Continuous deployment (CD)
 - ▶ Automatically deploy tested code into the runtime environment e.g. dev, test, production or whatever
- ▶ So CI/CD automates testing of new or modified code, and if successful deploys it
 - ▶ this enables regular code releases so enables rapid response to changes in business needs
 - ▶ reduces the potential for human error in manually transferring and configuring files between environments
- ▶ There are various CI/CD tools available such as gitlab, Jenkins, GitHub actions, and cloud services
- ▶ The following demo will use Terraform to:
 - ▶ create the target infrastructure
 - ▶ create a basic CI/CD pipeline to automate deployment of a previous simple basic app

Terraform basic CI/CD



Authentication

- ▶ Terraform needs to login to Azure to do its thing
- ▶ There are various ways but probably the easiest is to use the Azure CLI
- ▶ Terraform doesn't use the CLI to create resources but it can use it to authenticate
- ▶ Download the cli. For Windows I chose the MSI for 64 bit
- ▶ You can now also use the CLI instead of Terraform if you want to for quick test builds
- ▶ **az version** # check it works
- ▶ **az login** # authenticate through browser popup
- ▶ **az account list --output table** # get a list of accounts - make sure yours is set to default
- ▶ **az account set -subscription "*your subscription number*"** # get from portal. Search for subscriptions
- ▶ You should now be able to use Terraform to build Azure infrastructure