Infrastructure as Code (IaC)

bjhuang (2026)
tsaimh (2025)
wangth(2023-2024)

Outline

- What is IaC?
- Ansible
- OpenTofu

What is IaC?

Example scenario

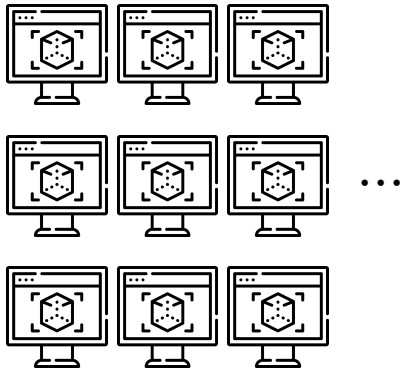
- You need to set up 3 computers
  - Every computer requires compilers, VScode and mounts specific NFS

```
$ ssh host1
$ echo "nfs_server:/exam_files /mnt/exam nfs ro 0 0" >> /etc/fstab
$ sudo apt install build-essential code
$ exit

$ ssh host2
$ echo "nfs_server:/exam_files /mnt/exam nfs ro 0 0" >> /etc/fstab
$ sudo apt install build-essential code
$ exit

$ ssh host3
$ echo "nfs_server:/exam_files /mnt/exam nfs ro 0 0" >> /etc/fstab
$ sudo apt install build-essential code
$ exit
```

Example scenario (cont.)

- What if you have 100 computers to set up?
  - Manually configure costs too much time
  - Human may have error

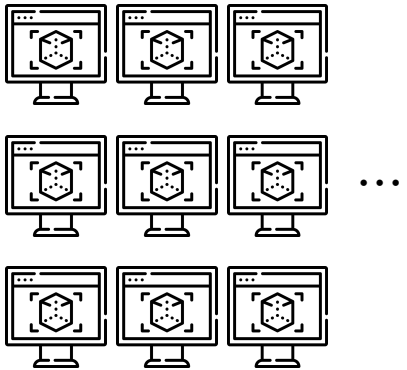
```
$ ssh host1
$ echo "nfs_server:/exam_files /mnt/exam nfs ro 0 0" >> /etc/fstab
$ sudo apt install build-essential code
$ exit
$ ssh host2
$ echo "nfs_server:/exam_files /mnt/exam nfs ro 0 0" >> /etc/fstab
$ sudo apt install build-essential code
$ exit
$ ssh host3
$ echo "nfs_server:/exam_files /mnt/exam nfs ro 0 0" >> /etc/fstab
$ sudo apt install build-essential code
$ exit
$ ssh host4
$ echo "nfs_server:/exam_files /mnt/exam nfs ro 0 0" >> /etc/fstab
$ sudo apt install build-essential code
$ exit
...
```



Example scenario (cont.)

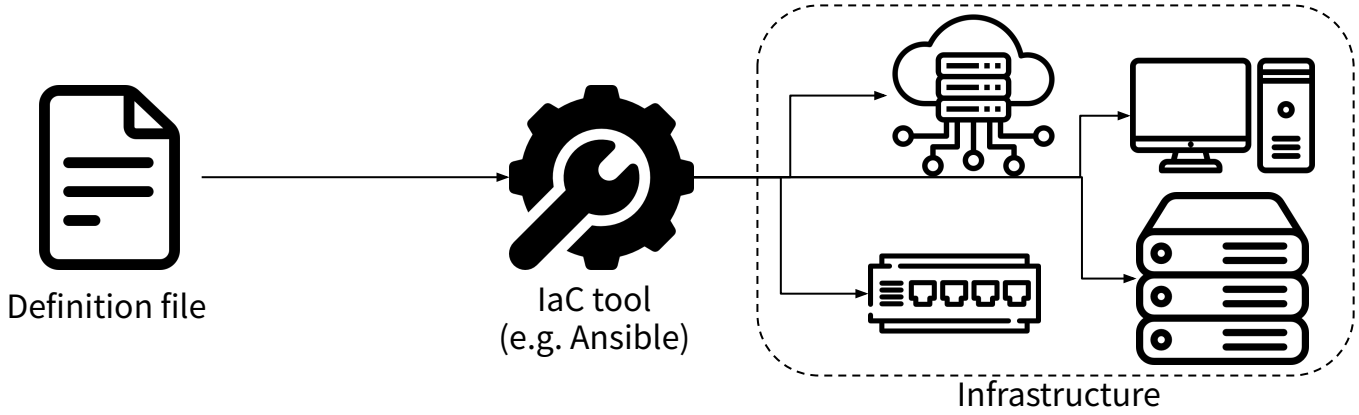
- What if you have 100 computers to set up?
  - What if using shell script?
    - Repeated runs create duplicate `/etc/fstab` entries. => Something we don't want

```
#!/bin/bash

for i in {1..100}
do
    ssh user@host$i "echo 'nfs_server:/exam_files /mnt/exam nfs ro 0 0' >> /etc/fstab"

    ssh user@host$i "sudo apt update && sudo apt install -y build-essential code"

    echo "Host$i Complete."
done

echo "Finished! (Hopefully nothing broke...)"
```



What is IaC?

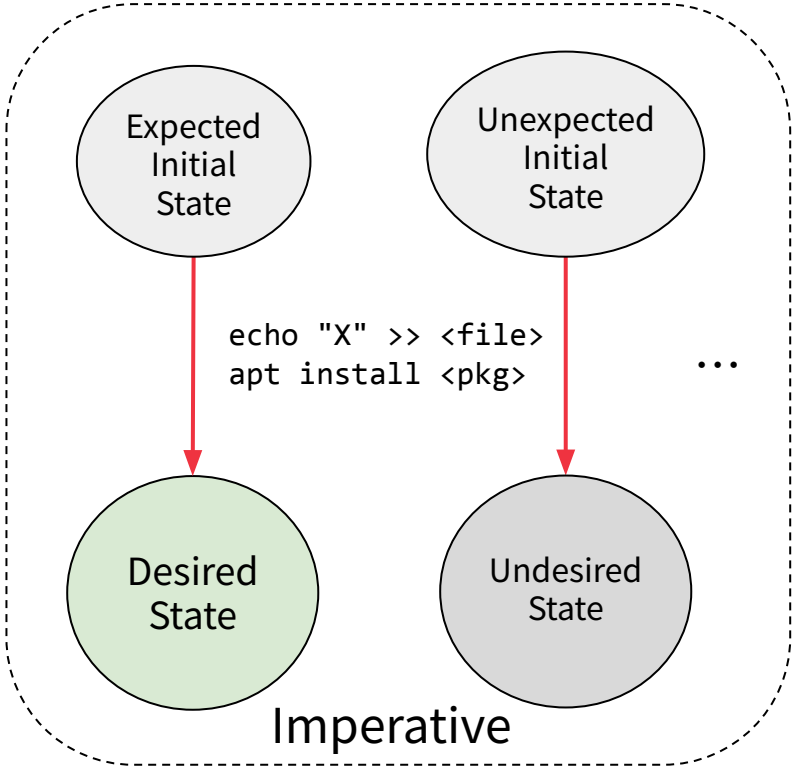
- Writing readable definition files to define the **desired state**.
- IaC tools reconcile infrastructure to match the files.

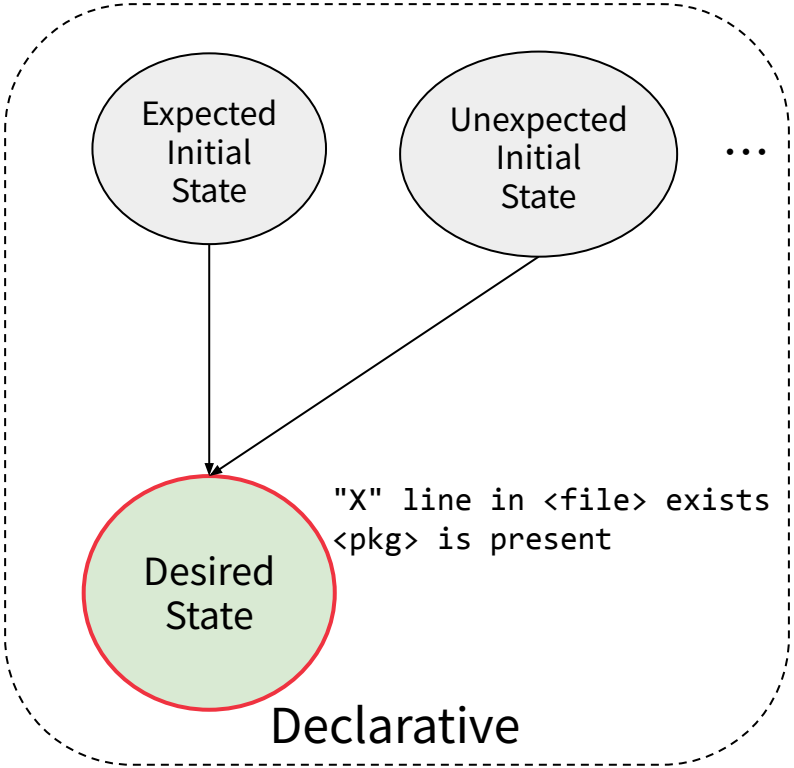


Initial state & Desired state

- Initial state
  - The current condition of your infrastructure (Reality)
  - e.g. VScode **not yet** installed, specific line is **absent** in the file
- Desired state
  - The desired condition defined in the definition file (Goal)
  - e.g. VScode **already** installed, specific line is **present** in the file

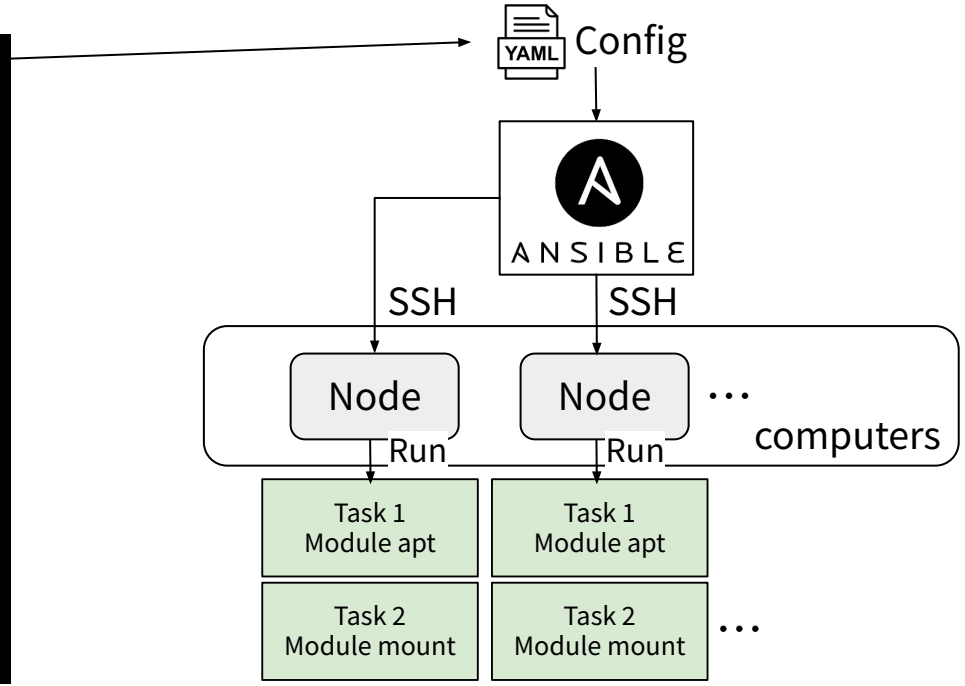
Imperative vs. Declarative





IaC example with Ansible

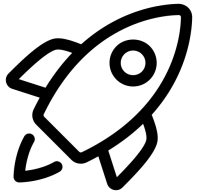
```yaml
---
- name: Setup environment
  hosts: computers
  become: true

  tasks:
    - name: Install compiler/vscode package
      ansible.builtin.apt:
        name:
          - build-essential
          - code
        state: present
        update_cache: true

    - name: Mount exam NFS on computers
      ansible.posix.mount:
        src: nfs_server:/exam_files
        path: /mnt/exam
        opts: ro
        fstype: nfs
        state: mounted
```



Why IaC?

- Three measurable categories for the value of IaC

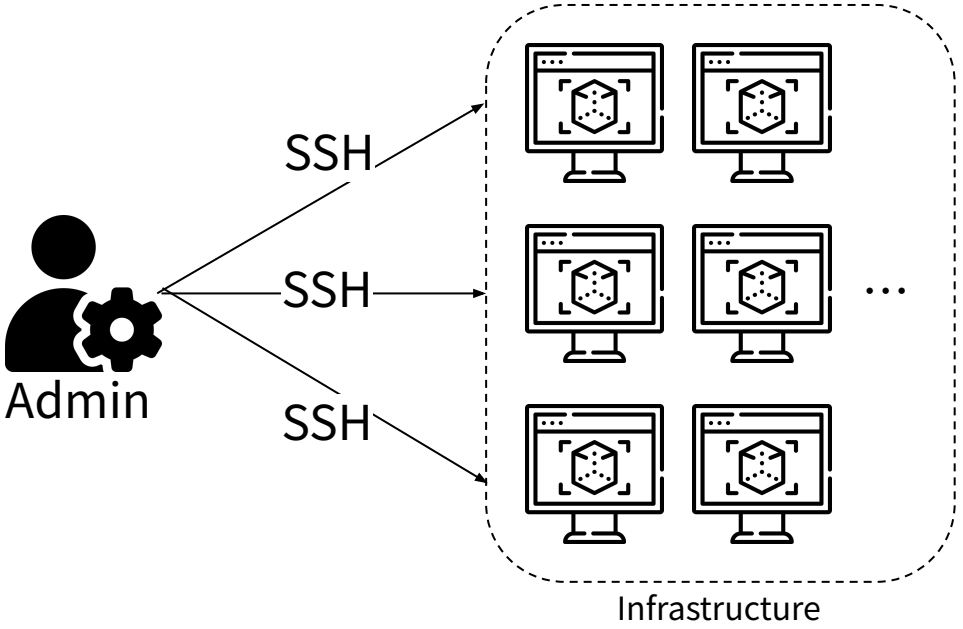
Cost optimization



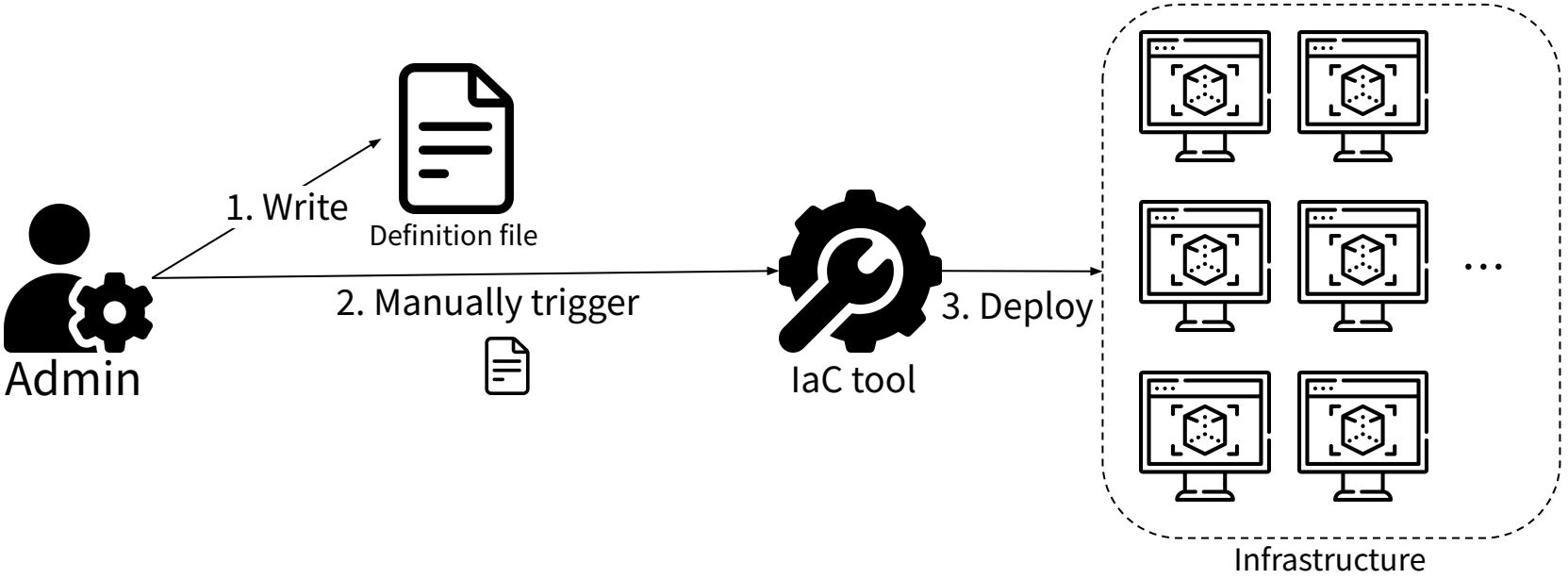
Speedup

Risk reduction

Before IaC

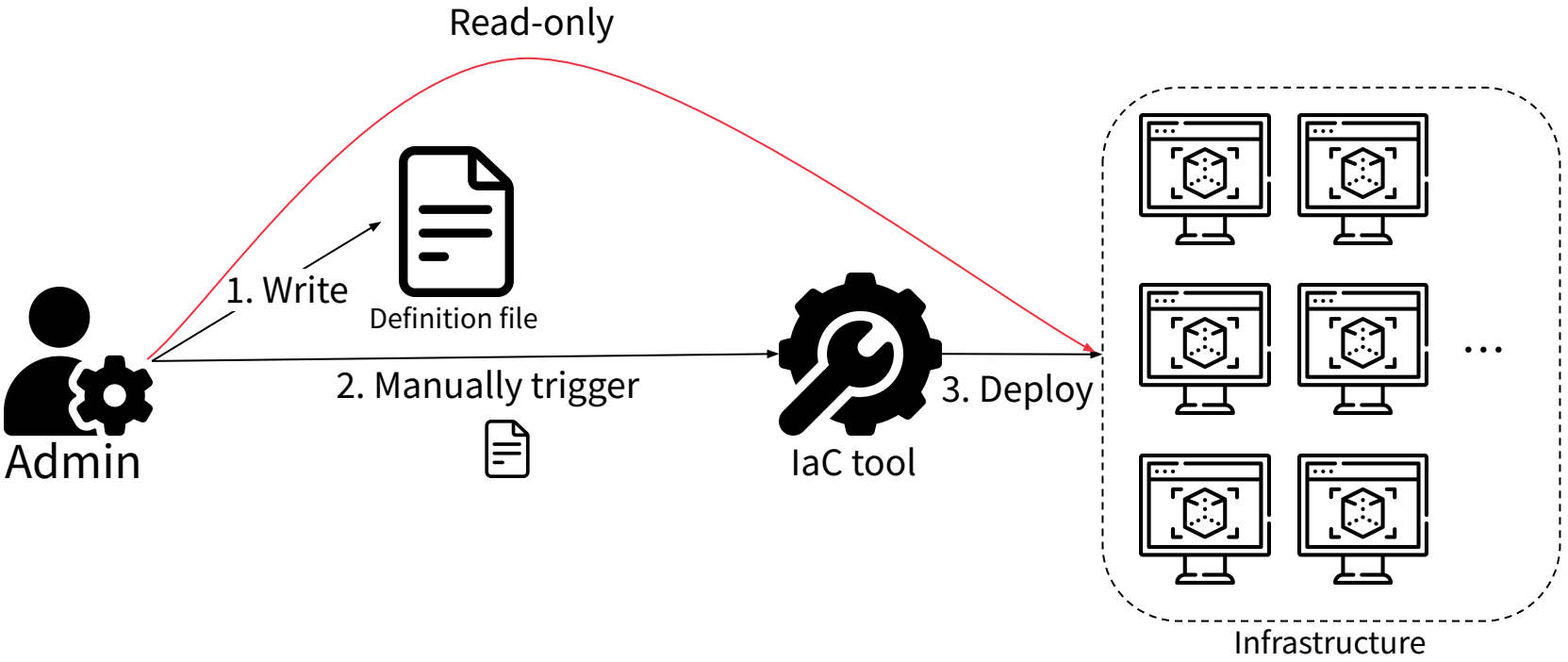


After IaC

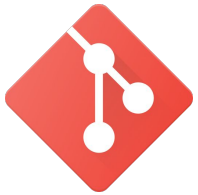

1. Write
Definition file
Admin
2. Manually trigger
IaC tool
3. Deploy
...
Infrastructure

IaC Best Practice

Single source of truth

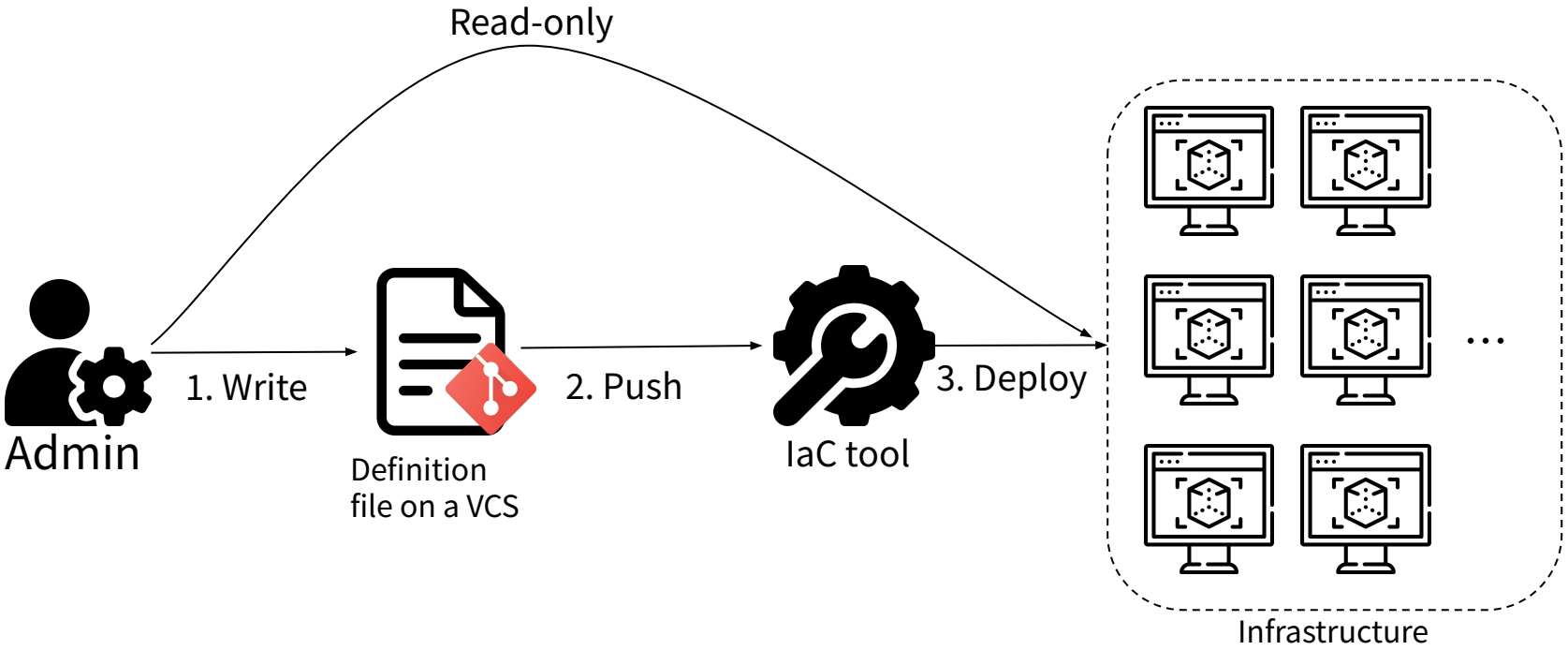


IaC with Version Control

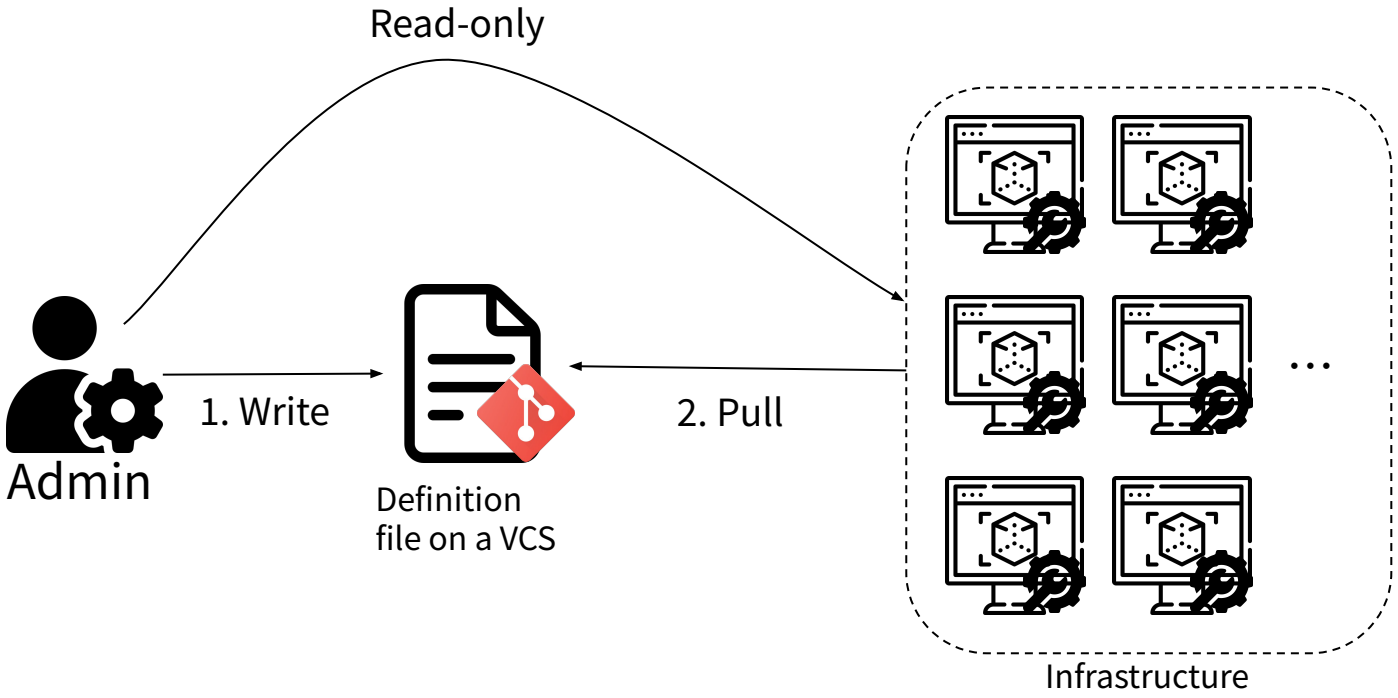


- Put definition files on a **version control system (VCS)**.
  - Auditability
    - Changes are tracked, reviewable, and reversible.
  - Collaboration
    - Improve team efficiency through peer-reviewed code changes.
  - GitOps
    - Use the data and code within Git to automatically manage infrastructure operations.
  - Single source of truth
    - The repository is the definitive authority on the current state of the environment.

After IaC and VCS (push model)

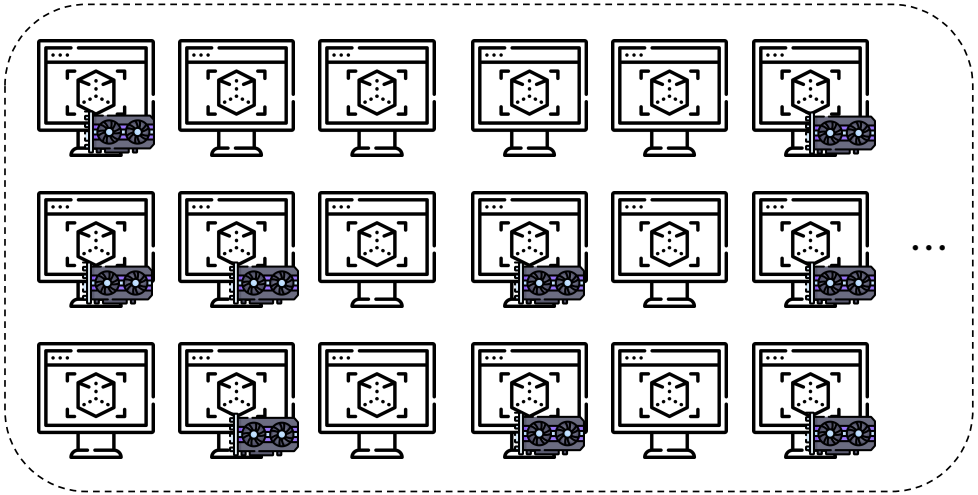


After IaC and VCS (pull model)



Expand previous scenario

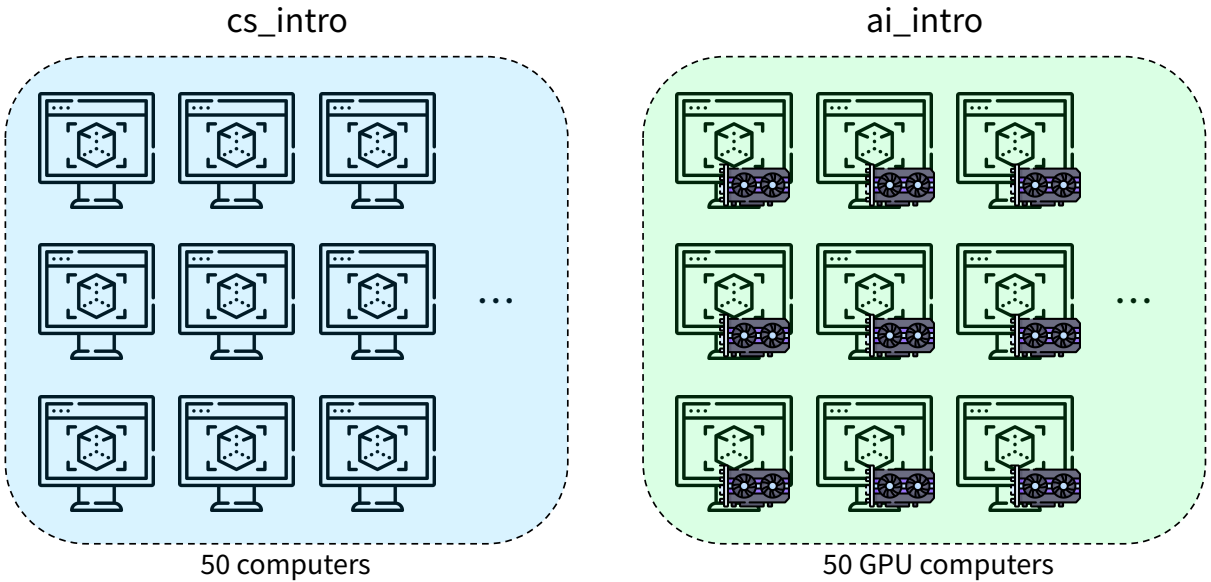
- What if 100 computers are for different purposes?
  - 50 computers for CS intro, 50 GPU computers for AI intro



100 computers

Expand previous scenario (cont.)

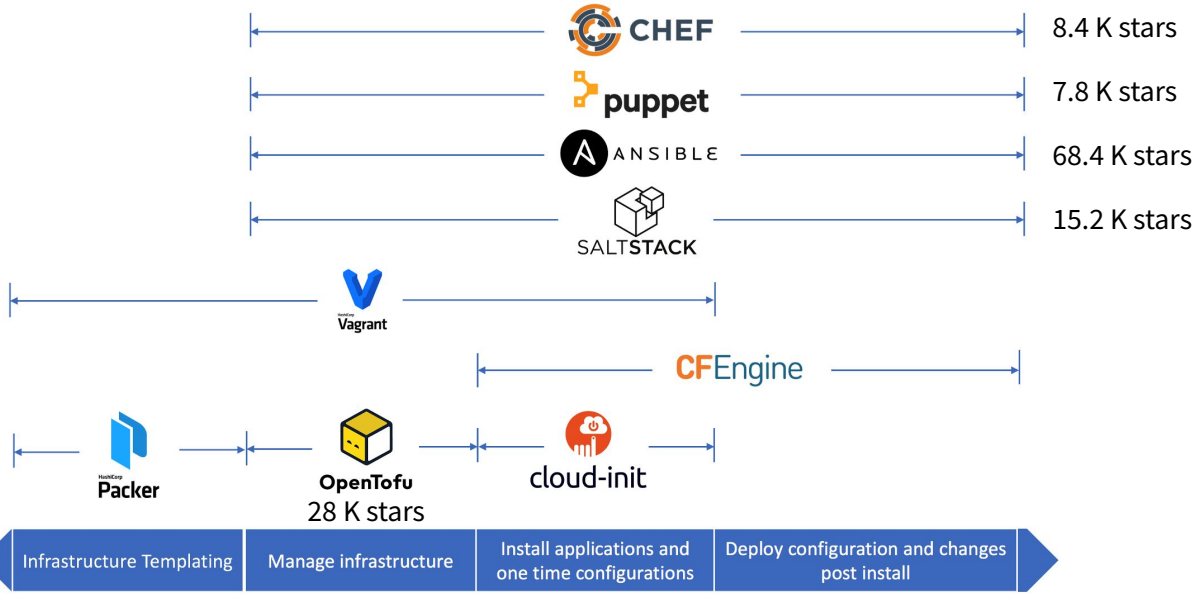
- Split computers into two groups logically
  - Executing specific tasks based on their groups



IaC appendix

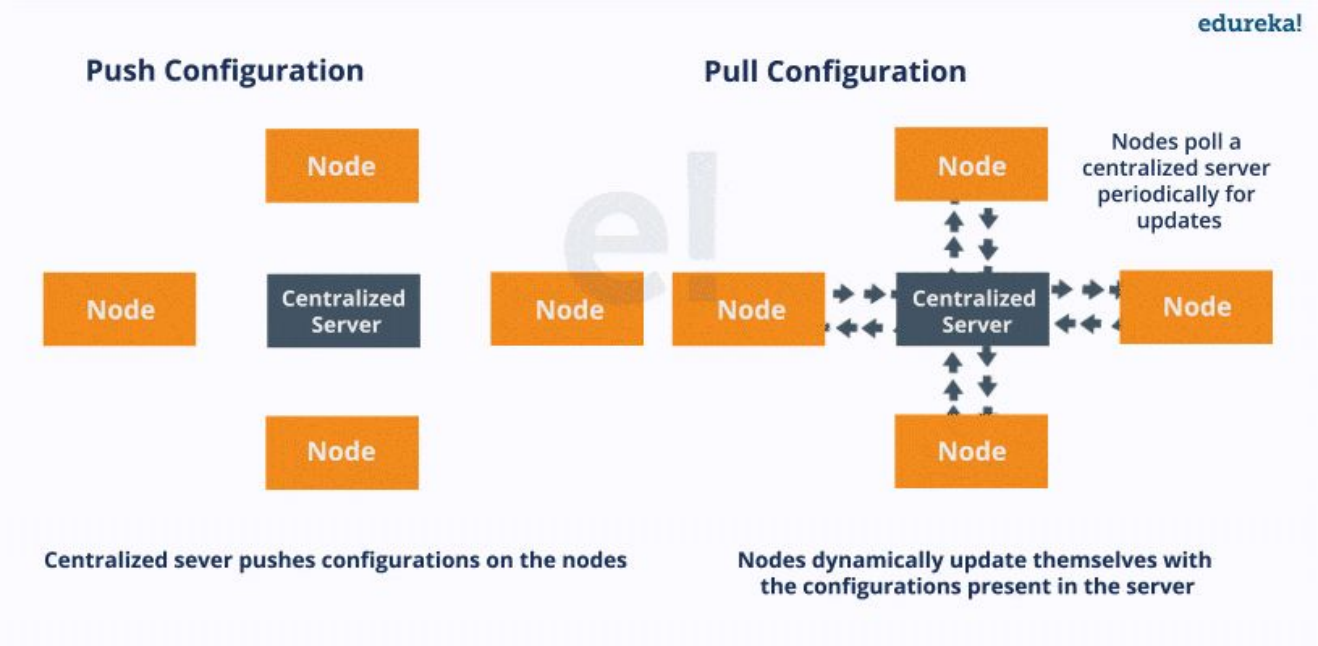
Common IaC tools

- The spectrum of some well-known IaC tools available today



Revised from:
https://medium.com/cloudnativeinfra/when-to-use-which-infrastructure-as-code-tool-665af289fbde

Push Model vs. Pull Model



What Is Chef? – A Tool Used For Configuration Management
https://www.edureka.co/blog/what-is-chef

Ansible

Introduction

- An open-source IaC tool
  - Admin writes YAML(s) to declare the desired state and Ansible ensure the system stay in that state

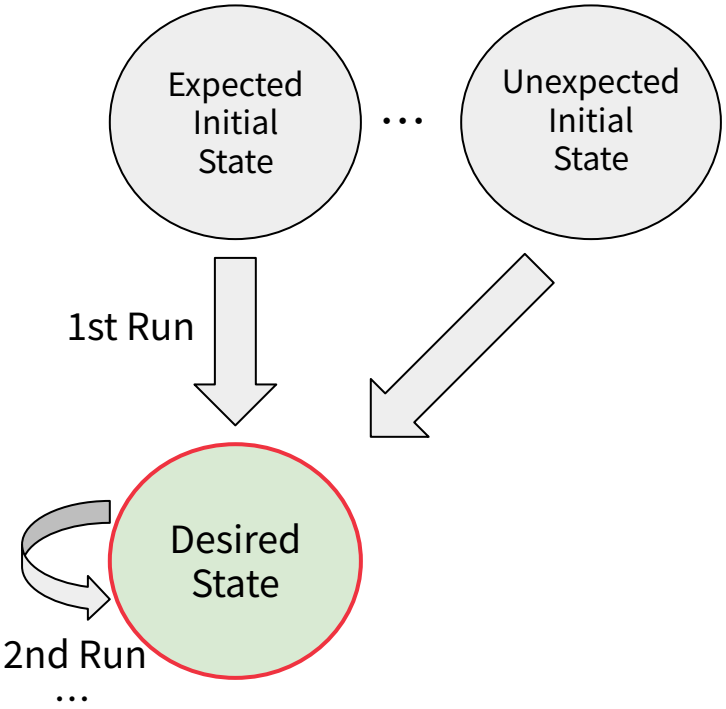
- Agentless — Nothing to install on managed nodes, just SSH and Python
- YAML-based — Readable without learning a new programming language
- Idempotency — Nothing change even if runs multiple times
- Cross platform/OS

YAML Ain't Markup Language (YAML)

- Syntax
  - Indentation Matters: Use Spaces, never Tabs.
  - "Space" Rule: Every colon(:) and dash(-) must be followed by a space.
  - Use --- to start a document and # for comments.
- Two Core Structures
  - Dictionary and List

```
---
# A clean, structured configuration
vm_config:                # Dictionary Key
  name: "node-01"         # Key: Value
  specs:                  # Nested Dictionary
    cpu: 4
    ram: "8GB"
  users:                  # A List of items
    - "user"              # Note the space after the dash
    - "student"
```

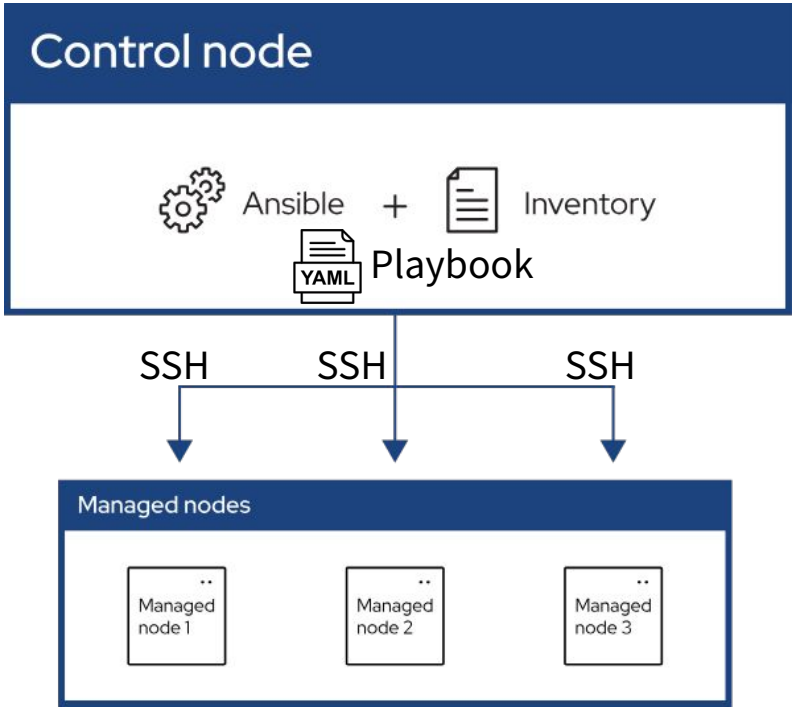
Idempotency



1st Run		2nd Run ...	
build-essential	CHANGED	build-essential	OK
vscode	CHANGED	vscode	OK
Mount exam	CHANGED	Mount exam	OK

Same playbook, same result. Only changes what differs.

Architecture

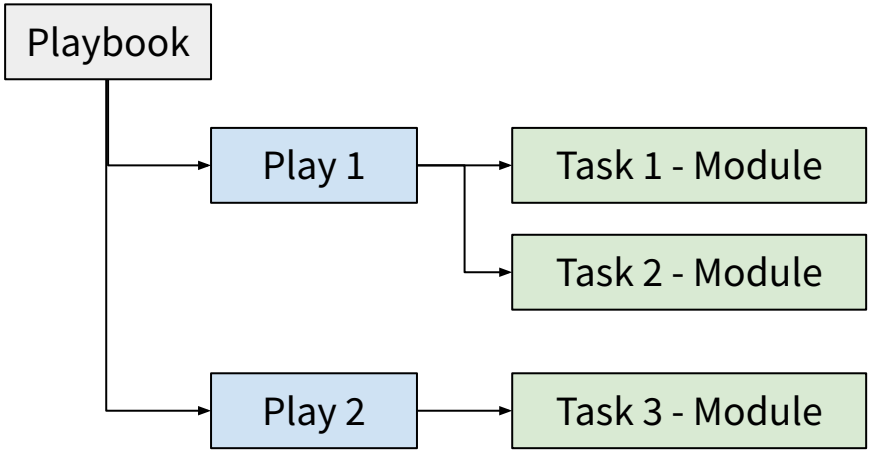


Source: Getting started with Ansible — Ansible Community Documentation

Ansible Concepts

Playbook

- Composed of one or more plays in an ordered list
  - Written in YAML
  - Command: `ansible-playbook <options> <name>.yml`



Playbook

```
---
- name: Setup cs_intro
  gather_facts: false
  hosts: cs_intro
  remote_user: root

  tasks:
  - name: Install vscode and compilers
    ansible.builtin.apt:
      name: [code, build-essential]
      state: present
  - name: Mount exam NFS on computers
    ansible.posix.mount:
      src: nfs_server:/exam_files
      path: /mnt/exam
      opts: ro
      fstype: nfs
      state: mounted
```

Play

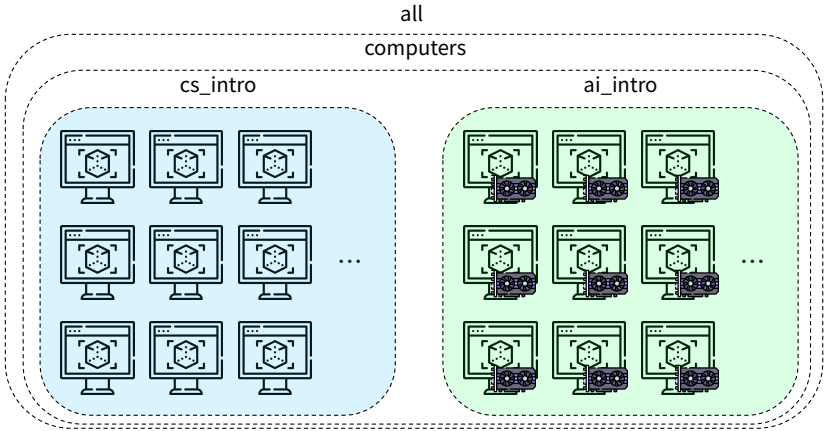
Module

```
- name: Setup ai_intro
  hosts: ai_intro
  remote_user: admin
  become: true

  tasks:
  - name: Install GPU driver
    ansible.builtin.apt:
      name: nvidia-open
      state: present
```

Play

Inventory

- Defines managed nodes and how to connect
  - YAML are common formats
- "Group" can organize nodes with different names
  - "Metagroup" models parent-child relationships between groups
    - `all` is a builtin metagroup that is the parent of all groups




```
$ cat inventory/hosts.yml
ai_intro:
  hosts:
    192.168.1.[1:25]:
    192.168.1.[26:50]:
cs_intro:
  hosts:
    192.168.1.[51:100]:
computers:
  children:
    ai_intro:
    cs_intro:
```

Ansible ad hoc command

- One line command to set up all nodes instead of writing playbooks
  - Used for rarely repeat tasks / non-consistent state
  - Command: `ansible <options> [host pattern] -m [module] -a "[module options]"`

```
# Check SSH connection
$ ansible all -m ping

# Copy file to managed nodes
$ ansible computers -m copy -a "src=/data/exam.pdf dest=/tmp/exam.pdf"

# Check home directory usage
$ ansible all -m shell -a "df -h /home"

# Reboot for GPU driver to activate
$ ansible ai_intro -m reboot

# Update package
$ ansible computers -m apt -a "name=code state=latest update_cache=true"
```

Task status

status	meaning
`ok`	Nothing changed. The system was already in the "desired state".
`changed`	The task ran and modified something.
`failed`	The task crashed or returned an error. By default, Ansible stops running tasks for that host.
`skipped`	The task was ignored because a `when:` condition was not met.
`unreachable`	Ansible couldn't connect to the host. (SSH, network issue, etc.)

Become

- Ansible provides different become methods for privilege escalation
  - Uses **sudo** by default
- Ansible CLI option **`-b`** means become
  - Add option **`-K`** to prompt for become password

Please manage nodes as a **Non-Root** user
Use *become*, *become_user,* and *become_method* keywords to achieve privilege escalation

- Source code of Ansible builtin become plugins
  - You can use `ansible-doc -t become -l` to get all available become methods

Variable - Writing Reusable Playbooks

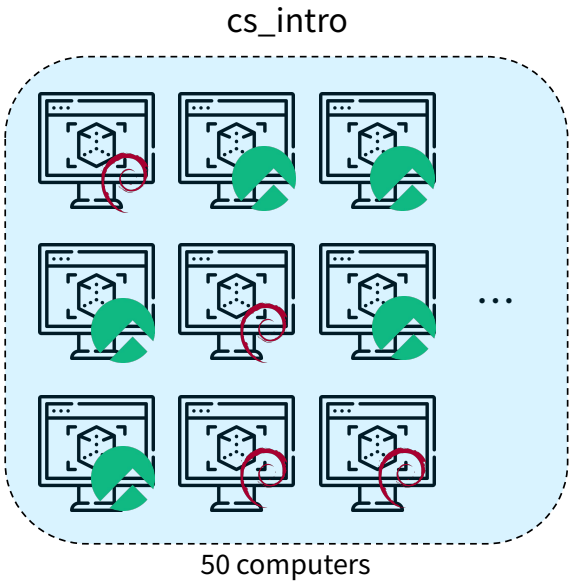
- Define variables with YAML syntax
  - single value, list, dictionary
  - Usually defined in `inventory/group_vars/<group name>.yml`
- Use Jinja2 syntax to reference them in playbook
  - `{{ var_name }}`
- Ansible CLI option **`-e`** can specify extra variables
  - e.g. `ansible-playbook install_py.yml -e "py_ver=3.12"`

```
$ cat inventory/group_vars/ai_intro.yml
---
py_ver: "3.12"


$ cat inventory/group_vars/cs_intro.yml
---
py_ver: "3.14"
```

```
# install_py.yml
...
- name: Install Python
  ansible.builtin.apt:
    name: "python{{ py_ver }}"
    state: present
```

Scenario

- CS intro 50 computers install different Operating systems
  - Debian and RockyLinux use different package manager (apt, dnf)



Fact

- A dictionary variable in Ansible
  - Contains several information about managed nodes
    - Hostname, OS, CPU, Network interfaces, …
    - You can read the example fact here
      - Or run `ansible <hosts> -m ansible.builtin.setup` for your machines
  - Use `{{ ansible_facts['xxx']['xxx']... }}` to reference in playbook

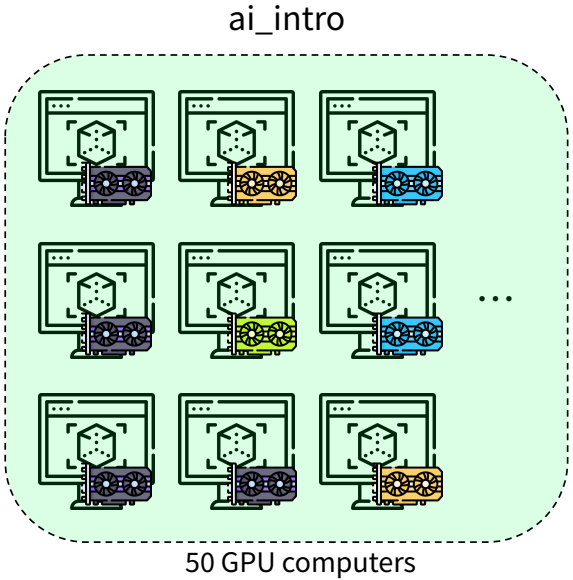
- The Gather Facts task runs implicitly when running a play by default
  - Runs ansible.builtin.setup module
    - To speedup, set `gather_facts: false` and run the module for what you need

Fact (cont.)

```
- hosts: cs_intro
  gather_facts: false

  tasks:
  - name: Only get OS info with fact
    ansible.builtin.setup:
      filter:
        - 'ansible_os_family'

  - name: Install package with apt
    ansible.builtin.apt:
      name: code
    when: {{ ansible_facts['os_family'] }} == 'Debian'

  - name: Install package with dnf
    ansible.builtin.dnf:
      name: code
    when: {{ ansible_facts['os_family'] }} == 'RedHat'
```

Scenario

- 50 GPU computers have different types of GPUs
  - Need to install different versions of drivers



Register Variable

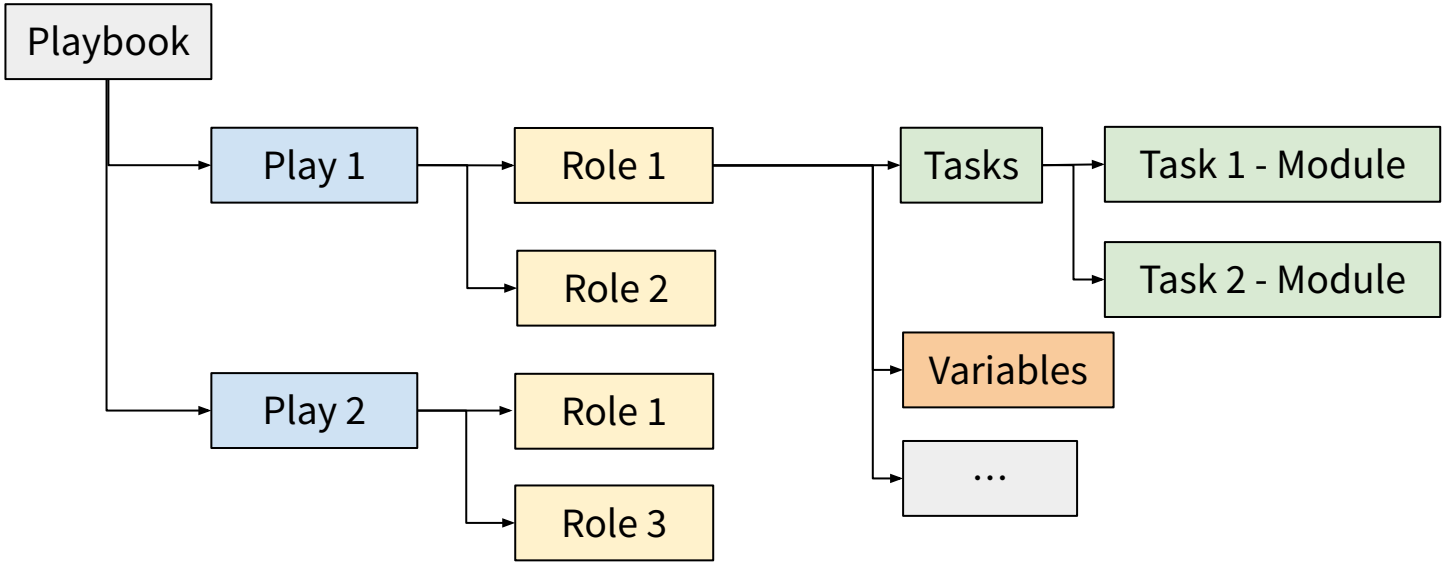
- Create a variable from the return values of an Ansible task
  - Read the module documentation for all available return values
    - e.g. ansible.builtin.shell

```
- hosts: ai_intro

  tasks:
  - name: Register current nvidia GPU pci id
    ansible.builtin.shell:
      cmd: "lspci -d 10de::0300 -n | awk '{split($3,a,":"); print a[2]; exit}'"
    register: gpu_dev
    failed_when: gpu_dev.rc != 0

  - name: Install proprietary driver for 1060
    ansible.builtin.apt:
      name: cuda-drivers
    when: gpu_dev.stdout == "1c03"

  # Handling other situations...
```

Role - Modularize complex playbook

- Reusable Ansible content (tasks, handlers, variables, templates and files) for use inside of a Play.

```
# setup.yml
---
- hosts: cs_intro
  roles:
    - common
    - cs_pkg

- hosts: ai_intro
  roles:
    - common
    - driver
```



Role directory structure

- A defined directory structure under `roles/<role name>/`
  - `tasks/`
  - `templates/`
  - `defaults/`
  - `files/`
  - `…`

```
# setup.yml
---
- hosts: cs_intro
  roles:
    - common
    - cs_pkg

- hosts: ai_intro
  roles:
    - common
    - driver
```

```
roles/
    common/               # this hierarchy represents a "role"
        tasks/            #
            main.yml      #  <-- tasks file can include smaller files if warranted
        handlers/         #
            main.yml      #  <-- handlers file
        templates/        #  <-- files for use with the template resource
            ntp.conf.j2   #  <----- templates end in .j2
        files/            #
            exam.pdf      #  <-- files for use with the copy resource
            foo.sh        #  <-- script files for use with the script resource
        vars/             #
            main.yml      #  <-- variables associated with this role
        defaults/         #
            main.yml      #  <-- default lower priority variables for this role
        meta/             #
            main.yml      #  <-- role dependencies

    driver/               # same kind of structure as "common" was above, done for the driver role
    cs_pkg/               # ""
    fooapp/               # ""
setup.yml
```

Real example

- kubernetes-sigs/kubespray: Deploy a Production Ready Kubernetes Cluster
- kolla-ansible/ansible at master · openstack/kolla-ansible

Ansible Best Practice

Playbook tips

- Use whitespace
  - A blank line before each task, making a playbook easy to scan
- Always add names to plays and tasks
- Always mention the state for modules
- Use fully qualified collection names
  - Use `ansible.builtin.apt` instead of only `apt`
- Some commands help you write better playbook
  - `ansible-playbook --syntax-check`
  - `ansible-lint`

Inventory tips

- Do not use spaces, hyphens, or preceding numbers in group names
  - Use `floor_19`, not `19th-floor`
- Group inventory by function
  - e.g. `webserver`, `dbserver`, ...
- Separate production, staging, test and development inventory
  - Split into different files, e.g. `inventory/prod.yml`, `inventory/staging.yml`, ⋯

Ansible Appendix

Ansible model

- Uses "push" model by default
- Pull model is provided for nodes to check on a particular schedule
  - `ansible-pull`
    - Pulls playbooks from a VCS repo and executes them for the local host

Inventory layout

- Organizing inventory in a directory with multiple files
  - Inventory files are usually stored under `inventory/` path
  - Use `ansible-inventory` to show how Ansible parses inventory
    - Add `-i` option to specify the inventory file

```
$ ansible-inventory -i inventory/prod.yml --graph
@all:
  |--@ungrouped:
  |--@webservers:
  |  |--@prod_web:
  |  |  |--foo.example.com
  |--@dbservers:
  |  |--@prod_db:
  |  |  |--192.168.1.1
  |  |  |--192.168.1.2
  |--@prod:
  |  |--@prod_web:
  |  |  |--foo.example.com
  |  |--@prod_db:
  |  |  |--192.168.1.1
  |  |  |--192.168.1.2
```

```
$ ansible-inventory -i inventory/test.yml --graph
@all:
  |--@ungrouped:
  |--@webservers:
  |  |--@test_web:
  |  |  |--bar.example.com
  |--@dbservers:
  |  |--@test_db:
  |  |  |--192.168.1.3
  |--@test:
  |  |--@test_web:
  |  |  |--bar.example.com
  |  |--@test_db:
  |  |  |--192.168.1.3
```

Connection

- Ansible provides plugins for connecting to managed nodes without agents
  - **SSH**
    - OpenSSH (default), Paramiko
  - Local
  - WinRM (for Windows)
- Some Ansible CLI options related to connection
  - **`-u`** if the username is different on the control node and the managed node(s)
  - **`-k`** for the user password prompt
  - **`--private-key`** to specify the SSH private key path

- Source code of Ansible builtin connection plugins
  - You can use `ansible-doc -t connect -l` to get all available connection plugins

Handler

- Handlers are tasks that only run once when notified
  - Notifications are triggered when the status of the notifying task is **changed**
  - Use `meta: flush_handlers` to trigger handler earlier instead of the end of playbook

```
---
- name: Revise nginx.conf if needed
  hosts: webservers
  tasks:
    - name: Write the nginx config file
      ansible.builtin.template:
        src: /srv/nginx.j2
        dest: /etc/nginx.conf
      notify:
        - Restart nginx

    - name: Ensure nginx is running
      ansible.builtin.service:
        name: nginx
        state: started

  handlers:
    - name: Restart nginx
      ansible.builtin.service:
        name: nginx
        state: restarted
```

Template - dynamic configuration

- Turning Jinja2 templates into customized system files using variables
  - ansible.builtin.template module can help render the jinja2 syntax

```
# hosts.j2
127.0.0.1   localhost
::1         localhost

# --- Lab Nodes Managed by Ansible ---
{% for host in groups['all'] %}
{{ hostvars[host]['ansible_host'] }} {{ host }}
{% endfor %}
```

```
- name: Synchronize Network Hosts File
  hosts: all
  become: true

  tasks:
    - name: Generate and deploy /etc/hosts
      ansible.builtin.template:
        src: hosts.j2
        dest: /etc/hosts
        owner: root
        group: root
        mode: '0644'
        backup: true
```

Comparison of config management(CM) tools

	Ansible	**SaltStack**	**Chef**	**Puppet**
Method	Push, Pull	Pull, Push	Pull	Pull, Push
	Agentless	Agent Agentless (Salt SSH)	Agent	Agent Agentless (Bolt)
Configuration Language	YAML Python	YAML Python	Ruby DSL	Puppet DSL
Implementation Language	Python	Python	Ruby Erlang	Ruby C++ Clojure
Company	Red Hat	VMware	Chef	Perforce

DSL: Domain Specific Language

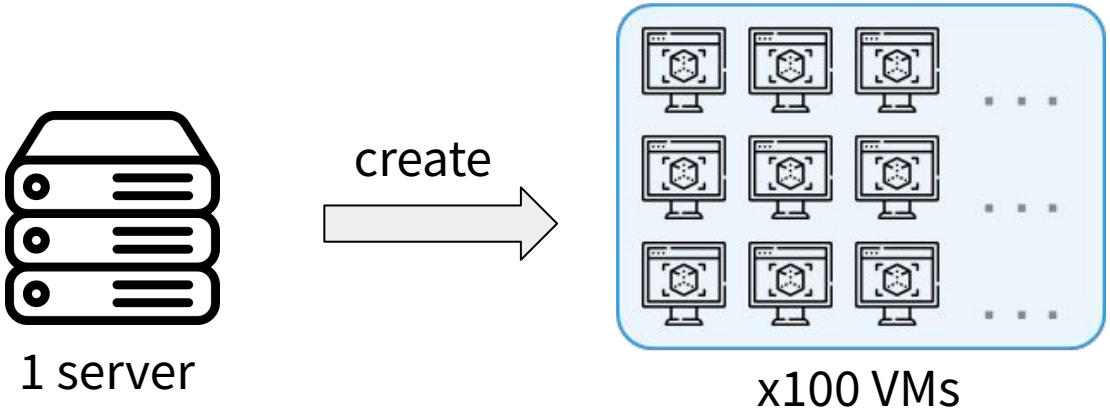
Terms used by each CM tool

Our term	Ansible	Salt	Puppet	Chef
operation	task	state	resource	resource
op type	module	function	resource type, provider	provider
op list	tasks	states	class, manifest	recipe
parameter binding	parameter play(book)	parameter top file	property, attribute classification, declaration	attribute run list
master host	control	master	master	server
client host	host	minion	agent, node	node
client group	group	nodegroup	node group	role
variable	variable	variable	parameter, variable	attribute
fact	fact	grain	fact	automatic attribute
notification	notification	requisite	notify	notifies
handler	handler	state	subscribe	subscribes
bundle	role	formula	module	cookbook
bundle repo	galaxy	GitHub	forge	supermarket

OpenTofu

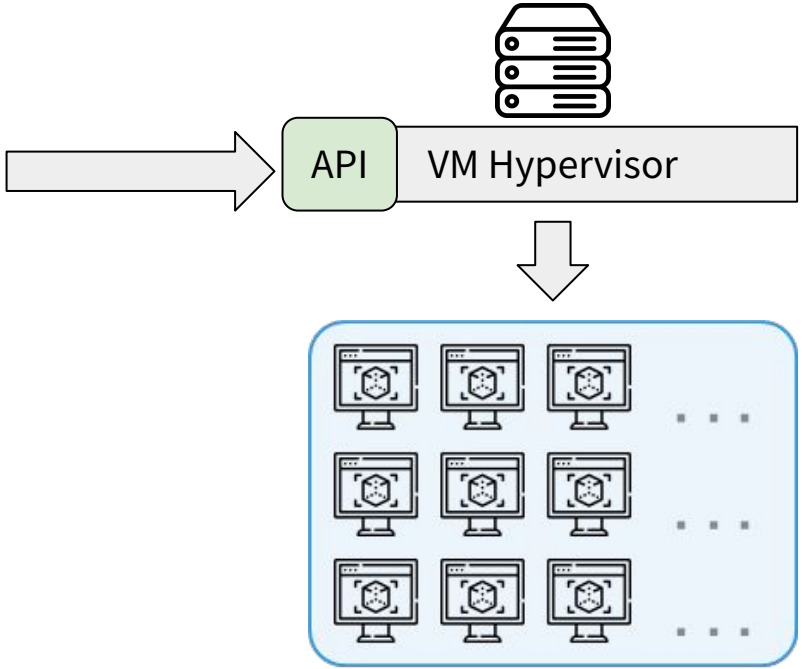
Scenario

- You don't have 100 computers, but a server with many hardware resources
  - Your boss ask you to create 100 VMs from scratch



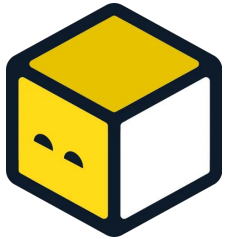
Virtual machine example

```
#!/bin/bash

for i in {1..100}
do
  curl -X POST "http://hypervisor-api/v1/domains" \
    -H "Content-Type: application/json" \
    -d "{
      \"name\": \"student-vm-$i\",
      \"memory\": 2048,
      \"vcpu\": 2,
      \"disk\": \"/var/lib/libvirt/images/disk-$i.qcow2\",
      \"network\": \"default\"
    }"
done

echo "Done."
```



x100 VMs created?

Introduction

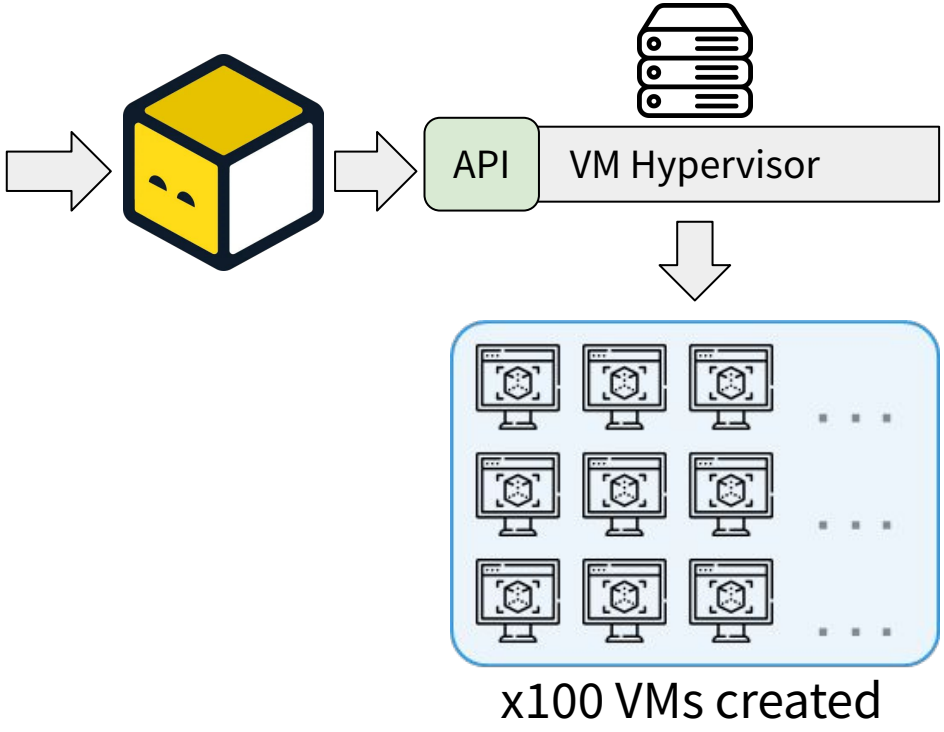


- An IaC tool used for provisioning anything if target **API**s support
  - e.g. virtual machines, S3 storage bucket, DNS records, Domino pizzas …
- The open-source fork of HashiCorp Terraform

- Ansible configures what is on a node; OpenTofu builds the node itself
- Ansible: Declarative tasks (Step by step)
- OpenTofu: Declarative resources (More abstract)

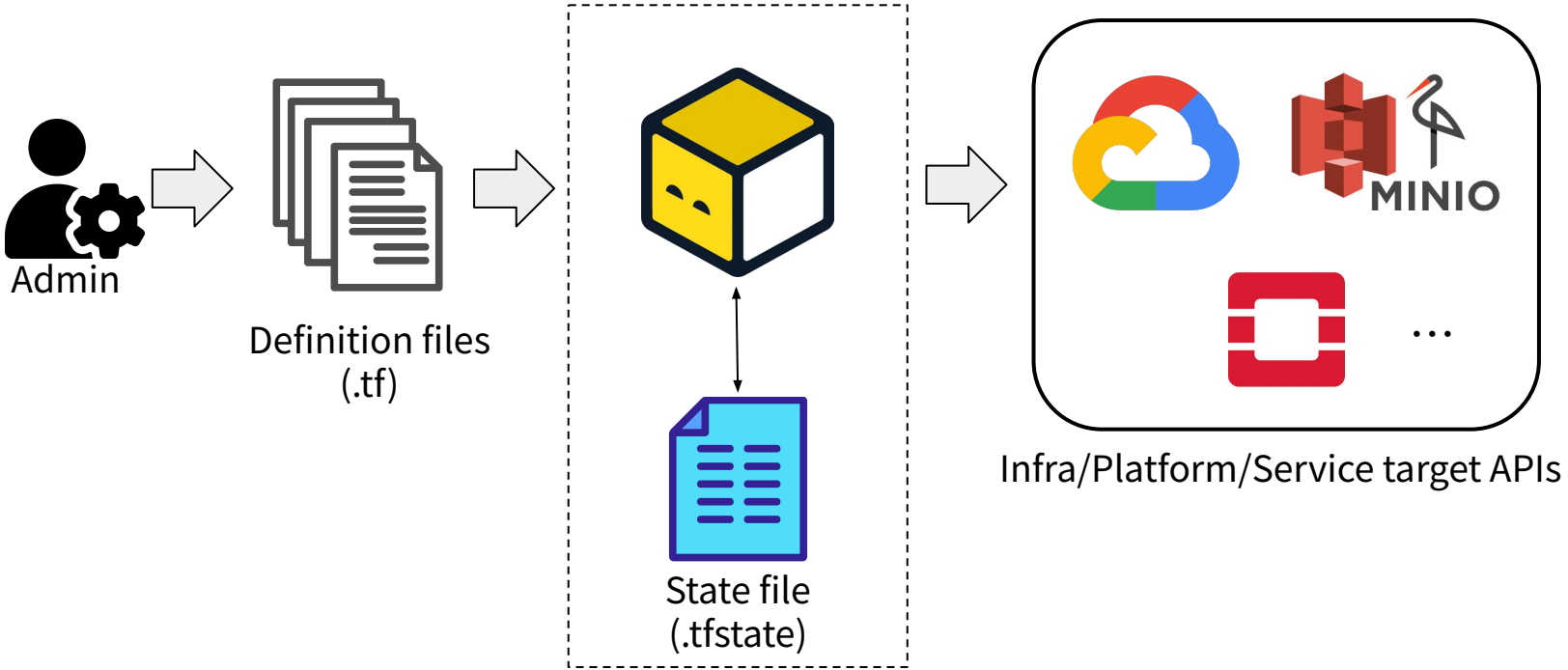
Virtual machine example

```
resource "libvirt_domain" "course_vms" {
  count  = 100

  name   = "student-vm-${count.index}"

  memory = 2048
  vcpu   = 2

  disk {
    file = "/var/lib/libvirt/images/disk-${count.index}.qcow2"
  }

  network_interface {
    network_name = "default"
  }
}
```



x100 VMs created

Architecture



Workflow

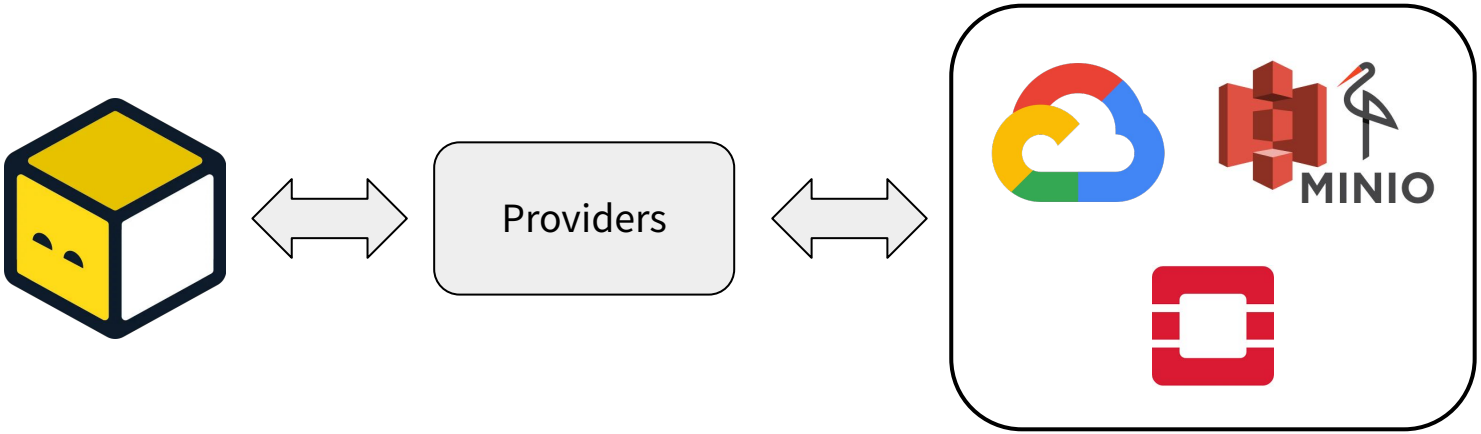
- Write
  - Admin write definition files(*.tf) in HashiCorp Configuration Language (HCL).
- Plan (`tofu plan`)
  - OpenTofu compares current state file and reality to desired state and shows a diff
  - No changes are made during plan, a safe preview
- Apply (`tofu apply`)
  - OpenTofu executes the plan; creates, modifies, or destroys resources as needed
  - Respects resource dependencies; updates the state file on completion

State

- OpenTofu records all managed resources in a state file
  - Stores mappings between configured resource instances and remote objects
    - e.g. mapping real VM ID to your resource in your config
- A file named "`terraform.tfstate`" by default
  - JSON format
  - Required for every plan and apply (Backup is important for this file!)

Provider

- A plugin that translates OpenTofu resource definitions into API calls
- e.g. Domino pizza, LibVirt, MinIO, DNS record ...



OpenTofu configuration

- Written in HCL (`.tf`)
  - The syntax looks like below:

```
<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {
  # Block body
  <IDENTIFIER> = <EXPRESSION> # Argument
}

resource "minio_s3_bucket" "main" {
  bucket         = "www-drive"
  quota          = 64 * 1024 * 1024 * 1024 # 64G
}
```

Common blocks in OpenTofu configuration

Block syntax	Purpose
`terraform {}`	Settings & provider requirements
`provider "<name>" {}`	Authentication information for the target API
`resource "<type>" "<name>" {}`	Actual things you want to build
`variable "<name>" {}`	Inputs to make your code be reusable module
`data "<type>" "<name>" {}`	Data that you need to fetch from external systems
`locals {}`	Internal constants that remaining code can refer to
`output "<name>" {}`	Print out information after resource is built

Virtual machine example (settings)

```
terraform {
  required_providers {
    libvirt = {
      source = "dmacvicar/libvirt"
    }
  }
}

provider "libvirt" {
  uri = "qemu:///system"
}
```

Virtual machine example (volume)

```
resource "libvirt_volume" "ubuntu_base" {
  name   = "ubuntu-22.04.qcow2"
  pool   = "default"
  target = {
    format = {
      type = "qcow2"
    }
  }

  create = {
    content = {
      url = "https://cloud-images.ubuntu.com/releases/22.04/release/ubuntu-22.04-server-cloudimg-amd64.img"
    }
  }
}

resource "libvirt_volume" "student_disks" {
  count          = 100
  name           = "student-disk-${count.index}.qcow2"
  base_volume_id = libvirt_volume.ubuntu_base.id
  pool           = "default"
  size           = 10737418240 # 10GB in bytes
}
```

Virtual machine example (network)

```
resource "libvirt_network" "lab_network" {
  name      = "course-net"
  mode      = "nat"      # NAT allows VMs to access the internet
  domain    = "lab.local"
  addresses = ["10.100.0.0/24"] # The IP range for your 100 VMs

  dns {
    enabled = true
    local_only = true
  }

  dhcp {
    enabled = true
  }
}
```

Full virtual machine example (VM)

```
resource "libvirt_domain" "course_vms" {
  count  = 100
  name   = "student-vm-${count.index}"
  memory = 2048
  vcpu   = 2

  # Reference the volume created in step 4
  disk {
    volume_id = libvirt_volume.student_disks[count.index].id
  }

  network_interface {
    network_id = libvirt_network.lab_network.id
    wait_for_lease = true
  }
}

output "vm_ips" {
  value = {
    for vm in libvirt_domain.course_vms :
    vm.name => vm.network_interface[0].addresses[0]
  }
}
```

Domino Pizza example

```
terraform {
  required_providers {
    dominos = {
      source = "MNThomson/dominos"
    }
  }
}

provider "dominos" {
  first_name    = "My"
  last_name     = "Name"
  email_address = "my@name.com"
  phone_number  = "15555555555"

  credit_card = {
    number      = 123456789101112
    cvv         = 1314
    date        = "15/16"
    postal_code = "18192"
  }
}
```

```
data "dominos_address" "addr" {
  street      = "123 Main St"
  city        = "Anytown"
  region      = "WA"
  postal_code = "02122"
}

data "dominos_store" "store" {
  address_url_object = data.dominos_address.addr.url_object
}

data "dominos_menu_item" "item" {
  store_id     = data.dominos_store.store.store_id
  query_string = ["philly", "medium"]
}

resource "dominos_order" "order" {
  api_object = data.dominos_address.addr.api_object
  item_codes = data.dominos_menu_item.item.matches[*].code
  store_id   = data.dominos_store.store.store_id
}

output "OrderOutput" {
  value = data.dominos_menu_item.item.matches[*]
}
```

OpenTofu Appendix

Backend configuration

- Let multiple people in the same team share state and work together
  - Default backend (local path) is not a good idea when collaborating
- Decide how to manage the state file
  - lock, unlock

```
# example: http backend with gitlab
PROJECT_ID="<gitlab-project-id>"
TF_ADDRESS="https://gitlab.com/api/v4/projects/${PROJECT_ID}/terraform/state/<state name>"

tofu init \
  -backend-config=address=${TF_ADDRESS} \
  -backend-config=lock_address=${TF_ADDRESS}/lock \
  -backend-config=unlock_address=${TF_ADDRESS}/lock \
  -backend-config=username=<gitlab-username> \
  -backend-config=password=<gitlab-personal-access-token> \
  -backend-config=lock_method=POST \
  -backend-config=unlock_method=DELETE \
  -backend-config=retry_wait_min=5
```

Module - Reusable Infrastructure Patterns

- Allowing resource configurations to be packaged and reused
  - e.g. When constructing a virtual machine, it is often necessary to set its network and combine them together.
- Module Sources | OpenTofu
  - Local path
    - relative path (starts with `./`, `../`)
  - Public registry, e.g. official registry, git, ...

```
# module syntax
module "<Name>" {
  source = "<module source>"

  <varible key> = <variable value>
  ...
}
```

```
# standard structure of a module
$ tree minimal-module/
.
├── README.md
├── main.tf
├── variables.tf
├── outputs.tf
```

Reference

Ansible-related reference

- Introduction to ad hoc commands — Ansible Community Documentation
- ansible(1) — Arch manual pages
- YAML Syntax — Ansible Community Documentation
- Understanding privilege escalation: become — Ansible Community Documentation
- How to build your inventory — Ansible Community Documentation
- Getting started with Ansible — Ansible Community Documentation
- Getting started with Ansible Playbooks | Red Hat Ansible Automation Platform | 2.4 | Red Hat Documentation
- General tips — Ansible Community Documentation

OpenTofu-related reference

- Getting started | OpenTofu
- Purpose of OpenTofu State | OpenTofu
- Module Sources | OpenTofu
- Standard Module Structure | OpenTofu
- GitLab-managed Terraform/OpenTofu state | GitLab Docs
- Terraform Architecture Overview — Structure and Workflow | by Pradeep Raj | Medium

Reference

- Chef vs Puppet vs Ansible - Whizlabs Blog
- Chef Documentation
- Puppet Core 8.16.0
- Salt Table of Contents