Software Engineering

Prof. Dr. Christoph Schober

Department of Applied Computer Science,
Deggendorf Institute of Technology

christoph.schober@th-deg.de

Infrastructure As Code

Infrastructure As Code

1. Challenges and Principles

Building infrastructure is an evolving and complex art, which demands repetitive improvements involving aspects such as maintainability, reliability, scalability, fault-tolerance, and performance.

Configuration drift: System configuration, over time, diverges from its established known-good baseline or industry-standard benchmark.

Causes:

- Ad-hoc configuration and troubleshooting
- Poor communication
- Poor documentation

→ Configuration drift leads to snowflake infrastructure.

Snowflakes: Devices that are kept operational by manual configuration changes.

Good for a ski resort, bad for a data center.

Problems:

- Difficult to reproduce the system configuration in case of hardware outages.
- Hard to understand and modify. Upgrades may cause unpredictable effects.
- Their fragility lead to long, stressful bouts of debugging.

Erosion: Even without a new requirement, infrastructure will decay over time.

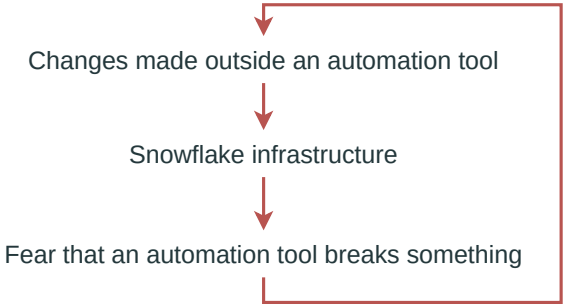
Examples:

- Operating system and infrastructure software patches.
- Logfiles filling up disk space.
- An application crashes and requires someone to login and restart it.
- Failure of underlying hardware causing infrastructure to fail.

→ Configuration drift and erosion lead to **automation fear**.

Automation fear: Lack of confidence that configuration can be reliably applied and problems caught quickly.

Vicious Circle:



Example: Installation of a Simulation Software

GPAW is a software for quantum chemistry, written in Python and available via pypi.

Installation well documented:
`https://wiki.fysik.dtu.dk/gpaw/install.html`

Nevertheless, command history (real example) on a server after installation might look like this:

```
200  sudo -E pip install ase gpaw
201  sudo apt-get install build-essential
202  sudo apt-get install libxc
203  sudo apt-get install libxc9
204  sudo -E pip install ase gpaw
205  gcc
206  cat /etc/debian_version
207  sudo apt-get install libxc-dev
208  sudo -E pip install ase gpaw
```

List of required steps:

1. Installation of explicit dependencies (libxc, blas, ...) via package manager (apt)
2. Installation of implicit dependencies (build-essential) via package manager (apt)
3. Installation of Python package for GPAW (pip)
4. Download of auxiliary files not bundled with GPAW (basis sets)
5. Unpacking of files in a directory
6. Configuring an environment variable (GPAW_SETUP_PATH) persistently in `/etc/profile`

→ What if same setup is required again on a different server? What if the server must be migrated or re-created?

How to overcome these challenges?

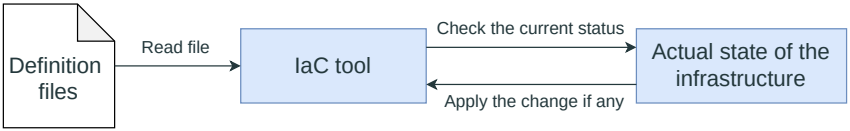
- Related principles and challenges.
  → Software engineering practises.
- Systematic change management.
  → Testing infrastructure changes.
- Chosing the right tools.
  → Pipelines.

Warning: There is no such thing as a silver bullet to magically solve all problems.

Infrastructure As Code (IaC)

Approach to define and deploy infrastructure, such as networks, virtual machines, load balancers, and connection topologies with a descriptive model.

- Understand infrastructure as a software system using software tools.
- Manages and provisions infrastructure through **machine-readible definition files**, rather than physical hardware configuration or interactive tools.



Principles of Infrastructure as Code:

- **Reproducibility:** It should be possible to effortlessly and reliabliy rebuild any element of an infrastructure.
- **Consistency:** Infrastructure elements providing the same service should be configured nearly identical.
- **Repeatability:** Any action that is carried out on the infrastructure should be repeatable. If a task can be scripted, script it!
- **Disposability:** Freely destroy and replace elements whenever needed.
- **Self-documentation:** Steps to carry out a process is captures in definition files, scripts, and tooling. Documentation outside can be minimal.

Versioning: Managing all assets in a version control system is the cornerstone of infrastructure as code. It makes it possible to automate processes around making changes, including tests and auditing.

Reasons why VCS is essential for IaC:

- *Traceability:* Version control systems maintain a history of changes.
- *Rollback:* When a change breaks something, it can be easily reverted.
- *Visibility:* Everyone can see when changes are commited.
- *Actionability*: Automatisms can be triggered when a change is commited.
- *Quality Assurance:* Merge requests undergo the 4/6-eye principle.

Approaches to ”write down” configuration in definition files:

- **Imperative:** Define how the configuration system should setup the infrastructure element. *How* should the system deploy infrastructure?

  Example: *Assign IP address 192.168.1.17 to the VM's Ethernet interface*

- **Declarative:** Define the target state of the infrastucture element. *What* should the eventual infrastructure deployment look like?

  Example: *Main IP address: 192.168.1.17*

Infrastructure As Code

2. Vagrant

Vagrant

Vagrant is a free and open source tool by Hashicorp[1] aimed at building a repeatable development environment inside a virtual machine.

- Based entirely on simple Ruby code.
- Vagrant uses *boxes* to run, which are packaged virtual machines.
- The vagrant infrastructure is described in a file named `Vagrantfile`.

[1] https://www.vagrantup.com/

A basic `Vagrantfile`[2] using the box `debian/bullseye64` in VirtualBox:

```
1 Vagrant.configure("2") do |config|
2     config.vm.box = "debian/bullseye64"
3     config.vm.provider = "virtualbox"
4     # ...other config here
5 end
```

Note: HashiCorp's Vagrant Cloud has a public directory of freely available boxes that run on various platforms and technologies.

[2] https://developer.hashicorp.com/vagrant/docs/vagrantfile

Hypervisor specific configuration[3] is defined as follows :

```
1  Vagrant.configure("2") do |config|
2      config.vm.box = "debian/bullseye64"
3
4      config.vm.provider "virtualbox" do |vb|
5          vb.name = "HPC Cluster Node01"
6          vb.cpus = 2
7          vb.memory = 2048
8          vb.customize ["modifyvm", :id, "--nested-hw-virt", "on"]
9      end
10 end
```

[3] https://developer.hashicorp.com/vagrant/docs/providers/configuration

Vagrant private networks[4] allow to access a guest machine by an IP address that is not publicly accessible:

```
1 Vagrant.configure("2") do |config|
2     config.vm.hostname: "hpc-node01"
3     config.vm.network "private_network", ip: 192.168.56.101
4 end
```

[4] https://developer.hashicorp.com/vagrant/docs/networking

Vagrant is able to define and control multiple guest machines per Vagrantfile. This is known as a *multi-machine* environment[5]:

```
1  Vagrant.configure("2") do |config|
2      # ...other config here
3      config.vm.define "node01" do |node01|
4          node01.vm.network "private_network", ip: 192.168.56.101
5      end
6      config.vm.define "node02" do |node02|
7          node02.vm.network "private_network", ip: 192.168.56.101
8      end
9  end
```

[5] https://developer.hashicorp.com/vagrant/docs/multi-machine

In the directory of the `Vagrantfile`, use the following CLI commands[6]:

- Bring up a virtual machine:

```
$ vagrant up
```

- SSH into the machine:

```
$ vagrant ssh
```

- Destroy the machine:

```
$ vagrant destroy
```

[6] https://developer.hashicorp.com/vagrant/tutorials/getting-started/getting-started-up

Vagrant supports **plugins** to add or change existing functionality.

Examples:

- `vagrant-vbguest`[7]: Automatically installs VirtualBox Guest Additions.
- `vagrant-disksize`[8]: Resizes the root disk in VirtualBox
- `virtualbox_WSL2`[9]: Fixes Vagrant CLI commands in WSL2.

Installation:

```
1 vagrant plugin install vagrant-vbguest
```

[7] https://github.com/dotless-de/vagrant-vbguest
[8] https://github.com/sprotheroe/vagrant-disksize
[9] https://github.com/Karandash8/virtualbox_WSL2

Infrastructure As Code

3. Ansible

Ansible

Suite of software tools to automate software provisioning, configuration management, and application deployment.

Ansible aims to be:

- **Clear:** Ansible uses a simple syntax (YAML) and is easy to understand.
- **Fast:** Fast to learn and fast to set up - no extra agents required on servers.
- **Complete:** *Batteries included*, everything needed is already included.
- **Efficient:** No extra software needed on the servers.
- **Secure:** Ansible uses SSH and requires no extra open ports on the servers.

Ansible uses an **inventory file**[10] to communicate with servers:

- Constitutes the list of hosts to manage.
- The most common formats are INI and YAML.
- The headings in brackets are **group names**.
- Hosts may be part of multiple groups.
- Ansible creates two default groups:
  - `all:` Contains every host.
  - `ungrouped:` Contains all hosts that don't have another group aside from `all`.

```
1  mail.example.com
2
3  [webservers]
4  foo.example.com
5  bar.example.com
6
7  [dbservers]
8  one.example.com
9  two.example.com
10 three.example.com
```

[10] https://docs.ansible.com/ansible/latest//inventory_guide/intro_inventory.html

Ansible **playbooks** offer a configuration management and deployment system:

- Expressed in YAML format.
- Runs in order from top to bottom.
- Defines the managed hosts to target.
- Runs at least one **task** to execute.
- Each task executes a **module**.
- A module is a reusable, standalone script that Ansible runs on all target hosts.
- When a task has executed on all target machines, Ansible moves on the next task.

```
1  - name: Update web servers
2    hosts: webservers
3    remote_user: root
4
5    tasks:
6    - name: Ensure latest httpd
7      apt:
8        name: httpd
9        state: latest
10   - name: Write config file
11     template:
12       src: /srv/httpd.j2
13       dest: /etc/httpd.conf
```

A list of *need-to-know* Ansible modules[11]:

Module	Description
apt	Installs package(s) on a system
service	Controls services (systemctl) on the target hosts.
copy	Copies a file from the local host to the target hosts.
debug	Prints statements during execution.
file	Manages files and its properties (also removes files).
lineinfile	Ensures that a line is in a file (also replaces lines).
git	Manages git checkouts of repositories.
shell	Executes shell commands on the target hosts.
template	Render a Jinja2 template on the target hosts.

[11] https://docs.ansible.com/ansible/2.9/modules/list_of_all_modules.html

Idempotence

Property of an operation whereby it can be applied multiple times without changing the result beyond the initial application.

- Ansible modules check whether the desired final state has already been achieved, and exit without performing any actions if it has been achieved.
- Example of the `lineinfile` module:

```
1 - name: Ensure SELinux is set to enforcing mode
2   lineinfile:
3     path: /etc/selinux/config
4     regexp: '^SELINUX='
5     line: SELINUX=enforcing
```

Example task using the template module:

```
- name: Copy a new sudoers file into place, after passing validation with visudo
  template:
    src: files/sudoers
    dest: /etc/sudoers
    validate: /usr/sbin/visudo -cf %s
```

Corresponding Jinja2 template:

```
# {{ ansible_managed }}
{% for sudoer in sudoers %}
{{ sudoer.name }} ALL=({{ sudoer.users | join(',') }}) NOPASSWD: ALL
{% endfor %}
```

Ansible uses **variables** to manage differences between systems:

- Variables are referenced using the Jinja2 syntax:

```
1 - name: Copy a new sudoers file into place
2   template:
3     src: files/sudoers
4     dest: '/etc/{{ sudoers_file }}'
5     validate: /usr/sbin/visudo -cf %s
```

- Variables related to remote systems are called **facts**[12]. Example:

```
1 {{ ansible_facts.eth0.ipv4.address }}
```

[12] https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_vars_facts.html

Variables can be defined in a variety of places:

- **At runtime**: Variables can be defined at the command line using the `--extra-vars` (or `-e`) argument in `key=value` format. Example:

```
ansible-playbook site.yml -e sudoers_file="/etc/sudoers"
```

- **In a playbook**: Variables can be defined directly in a playbook:

```
- hosts: webservers
  vars:
    sudoers_file: "/etc/sudoers"
```

- **As host variable**: Variables defined in a file inside the directory `host_vars` that has a filename equal to the name of a host are set for that exact host.

  Examples:

  - `host_vars/foo.example.com`:

```
network_interfaces:
  - name: eth0
    ipv4_address: 192.168.56.101
```

  - `host_vars/bar.example.com`:

```
network_interfaces:
  - name: eth0
    ipv4_address: 192.168.56.102
```

- **As group variable**: Variables defined in files inside the directory `group_vars` that have a filename equal to the name of a group are set for all hosts that belong to that exact group.

  Examples:

  - `group_vars/webservers`:

```
1 http_port: 8080
2 https_port: 8443
```

  - `group_vars/dbservers`:

```
1 sudoers_file: /etc/sudoers
```

- **Precedence order** Command line > playbook > host vars > group vars

Ansible **roles** are reusable automation components that encapsulate artifacts such as tasks, templates, or configuration files.

- Roles are defined in `roles/requirements.yml`:[13]

```
- name: pip
  scm: git
  src: https://github.com/geerlingguy/ansible-role-pip
```

- Roles are included in a playbook:

```
- hosts: webservers:
  roles:
    - pip
```

[13] Installation: `ansible-galaxy install -r roles/requirements.yml`

Infrastructure As Code

4. Terraform

Terraform

Open source Infrastructure-as-code tool that defines and deploys infrastructure elements using a declarative configuration language.

- Developed and mainained by HashiCorp.
- Supported by all major cloud providers and virtualization platforms.
- The configuration language is called *HashiCorp Configuration Lanugage*.
- Configuration files may consist of multiple code blocks, wherein each codeblock corresponds to an infrastructure element.

Elements of the *HashiCorp Configuration Language* (HCL):

- **Blocks**: Containers where the configuration of a resource is kept. Blocks are comprised of block-types, labels, arguments, and nested blocks.
- **Arguments**: Value assignment to a specific name within blocks.
- **Expressions**: Literal or referenced values for arguments.
- **Values**: Can be combined using built-in functions.

A Terraform **provider** is responsible for understanding API interactions and exposing resources. Example configuration for the Proxmox provider[14]:

```
terraform {
    required_providers {
        proxmox = {
            source  = "telmate/proxmox"                # name of the provider
        }
    }
}

provider "proxmox" {                                   # provider specific config
  pm_api_url = "https://192.168.56.101:8006/api2/json" # url to the cluster api
}
```

[14] https://registry.terraform.io/providers/Telmate/proxmox/latest/docs/guides/installation

Example of a virtual machine definition for a Proxmox[15] cluster:

```
resource "proxmox_vm_qemu" "vm01" {
    name        = "vm01"                  # name of the vm
    target_node = "hpc-cluster-node01"    # name of the target cluster node
    clone       = "debian-11"             # name of the template to clone from
    cpu         = "kvm64"                 # type of the cpu (kvm64 compatible)
    cores       = 1                       # number of cpu cores
    memory      = 512                     # size of RAM in MB
}
```

Note: A Proxmox template is a fully pre-configured operating system image that can be used to create new KVM virtual machines by cloning.

[15] https://registry.terraform.io/providers/Telmate/proxmox/latest/docs/resources/vm_qemu

Additional blocks[16] define the disk and network configuration:

```
resource "proxmox_vm_qemu" "vm01" {
    # other configuration
    disk {
        type    = "scsi"   # type of disk device to add
        storage = "local"  # name of the storage pool on which to store the disk
        size    = "12G"    # size of the created disk, G represents Gigabytes
    }
    network {
        bridge  = "vmbr0"  # bridge to which the network device is attached
        model   = "e1000"  # network card model
    }
}
```

[16] https://registry.terraform.io/providers/Telmate/proxmox/latest/docs/resources/vm_qemu

Terraform provides a command-line interface[17] to deploy infrastructure:

- Set the login credentials for the Proxmox cluster:

```
$ export PM_USER="root@pam" && export PM_PASS="vagrant"
```

- Prepare the working directory for other commands:

```
$ terraform init
```

- Show changes required by the current configuration:

```
$ terraform plan
```

- Create or update infrastructure

```
$ terraform apply
```

[17] https://developer.hashicorp.com/terraform/cli/commands

Infrastructure As Code

5. Cloud Init

Cloud Init

Infrastructure-as-code tool that automates the configuration of an operating system during boot (often used for cloud server instances, hence the name).

Sample tasks that `cloud-init` can perform:

- Initializing users and groups.
- Installing SSH keys.
- Configuring network interfaces.
- Running scripts.
- Installing packages.

`cloud-init` uses YAML-formatted file instructions[18] to perform tasks:

```yaml
# cloud config
users:
  - name: woelfl
    passwd: my-encrypted-password
    groups: sudoers

groups:
  - admingroup: [root,sys]
  - cloud-users

runcmd:
  - ls -l /root
  - mkdir /run/mydir
```

```yaml
#cloud config
network:
  version: 1
  config:
    - type: physical
      name: eth0
      subnets:
        - type: static
          ipv4: 192.168.57.2
          netmask: 255.255.255.0
          gateway: 192.168.57.1
    - type: nameserver
      address: 1.1.1.1
```

[18] https://cloudinit.readthedocs.io/en/latest/topics/examples.html

The configuration data of is separated into:

- `user-data`: configuration data that comes from the user.
- `meta-data`: configuration data that comes from the VM provider.

The location of configuration data is referred to as **datasource**:

- Preconfigured in the file `/etc/cloud/cloud.cfg`[19].
- Loaded from any filesystem volume that is labeled `cidata` or `CIDATA`. The configuration data is expected to be in the following files:

```
1 /user-data
2 /meta-data
```

[19] The image `debian-11-ec2-amd64` is built to fetch the data from a preconfigured AWS location.

The Proxmox terraform provider has integrated support for `cloud-init`:

```
resource "proxmox_vm_qemu" "vm01" {
    # other configuration here
    ciuser      = "root"
    cipassword  = "vagrant"
    ipconfig0   = "ip=192.168.57.2/24,gw=192.168.57.1"
}
```

The `cicustom` argument is suitable for complex `cloud-init` configuration:

```
resource "proxmox_vm_qemu" "vm01" {
    # other configuration here
    cicustom    = "/path/to/user-data"
}
```

Infrastructure As Code

6. Summary

Summary

You should have accquired the following competencies:

- Understand the challenges in building complex infrastructures.
- Define Infrastructure-as-Code and describe its purpose.
- Create a cluster of virtual machines using Vagrant.
- Automate the software and network configuration of a system using Ansible.
- Define and deploy virtual infrastructure elements using Terraform.
- Automate the initial operating system configuration using Cloud Init.