ADDO

ALL DAY DEVOPS

NOVEMBER 12, 2020

Matteo Valentini, Nethesis

Immutable deployments: the new classic way for service deployment

Warning!

The events depicted in this talk are real. Any similarity to any technology living or dead is not a merely coincidental.

The illustrated approach is based on lessons learned in more than two years of using the methodology on a production service.

The Problems

SnowflakeServer

SnowflakeServer

A server that is unique[1]:

- Manual installation
- Manual configuration
- Manual maintenance

[1]https://martinfowler.com/bliki/SnowflakeServer.html

It Is **your** server and **you** take care of it, as you do with your pet.

Configuration Drift

The drift from a well know start state, even if automated configuration tool are used[2]:

- Automated configuration tools manage a subset of a machine's state
- Writing and maintaining manifests/recipes/scripts is time consuming

[2]http://kief.com/configuration-drift.html

The path of least resistance of services management

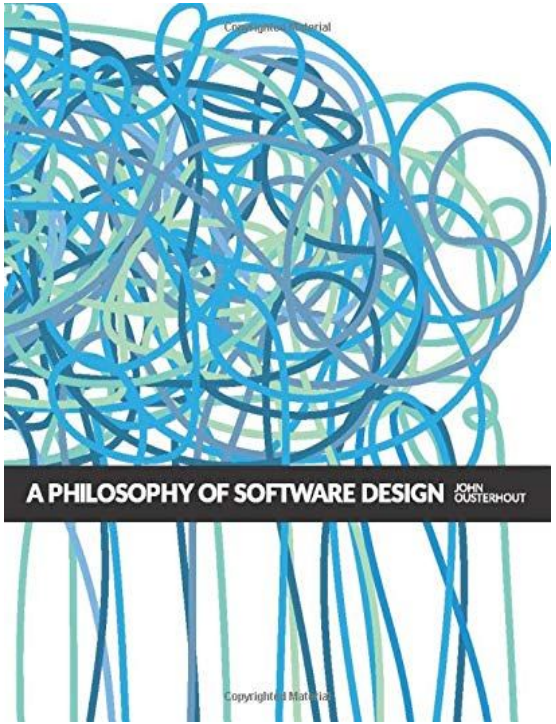
Every developer or operator will always follow the simple, less costly and quick way to fix a production problem.

And then he/she will forget about it.

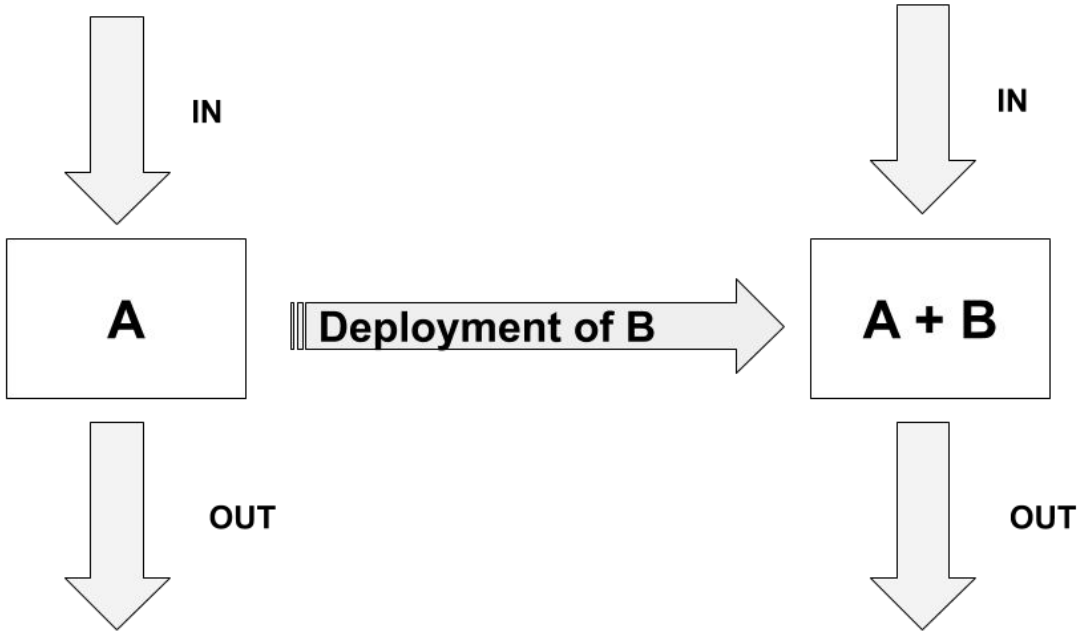
Unknown Unknowns

"An unknown unknown means that there is something you need to know, but there is no way for you to find out what it is, or even whether there is an issue."

John Outsterhout, "A Philosophy of Software Design", p. 9



Not Deterministic Deployment



The Solution

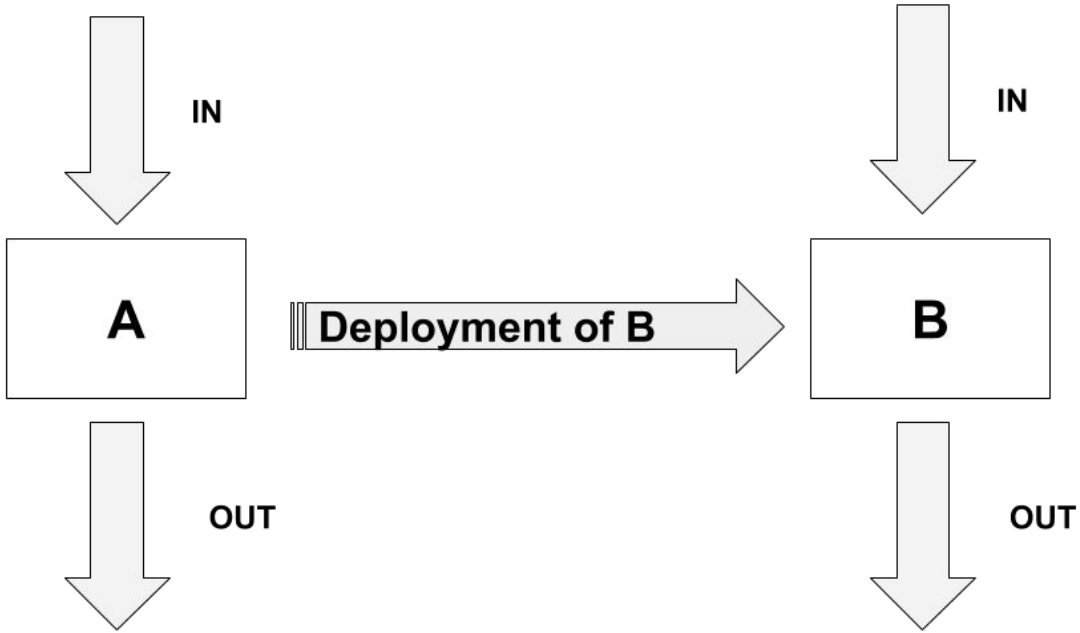
Immutable Infrastructure

"If you absolutely know a system has been created via automation and never changed since the moment of creation, most of the problems I describe above disappear. Need to upgrade? No problem. Build a new, upgraded system and throw the old one away. New app revision? Same thing. Build a server (or image) with a new revision and throw away the old ones."

Chad Fowler, "Trash Your Servers and Burn Your Code: Immutable Infrastructure and Disposable Components"

http://chadfowler.com/2013/06/23/immutable-deployments.html

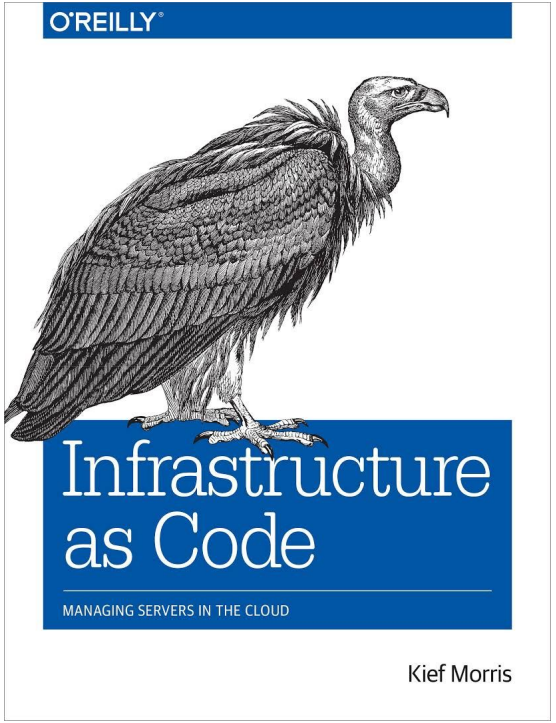
Deterministic Deployment



Immutable Infrastructure

"Immutable infrastructure make configuration changes by completely replacing the servers. Changes are made by building new server templates, and then rebuilding relevant servers using those templates. This increase predictability, as there little variance between servers as tested, and servers in production. It requires sophistication in server template management."

Kief Morris, "Infrastructure as Code: Managing Servers in the Cloud", p.70



What We Need?

- An automated provisioning/configuration tool
- An automated image generator tool
- An orchestrator
- A system to keep track of all the changes (we can use git for that)

The tools

Automated Provisioning

Shell scripts

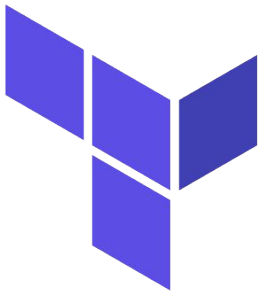
- Almost every developer can understand it
- Simple and at the same time very powerful

Image Builder

Packer

- JSON file configuration
- Multiple provisioners support:
  - Ansible
  - Puppet
  - Chef
  - Shell scripts
  - ...
- Multiple Builder support:
  - DigitalOcean
  - AWS
  - Google Cloud
  - Azure
  - ....

Orchestrator



HashiCorp
Terraform

Terraform

- DSL (HCL)
- Declarative language configuration
- Enable IaC
- Multi cloud support
  - AWS
  - Google Cloud
  - Azure
  - DigitalOcean
  - ...

Cloud platform

DigitalOcean

- Not expensive
- Simple
- Have every thing the you need:
  - APIs
  - Compute instances
  - Snapshots
  - Cloud-init
  - Floating IPs
  - Load Balancers

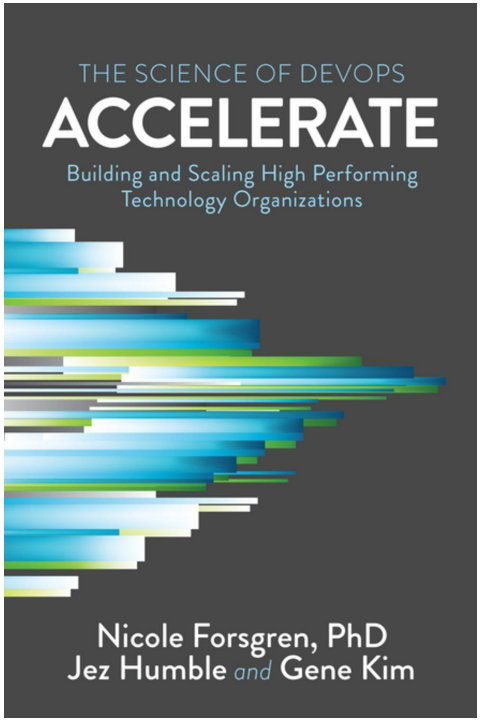
Why?

- Container vs VMs: The vm are a more familiar concepts, not all want or need to switch to container
- Configuration Management vs shell scripts: The learning steps can be too high, for some simple tasks a shell script is enough for the work
- Complex orchestrator vs IaC: For most of the company a complex orchestrator (like kubernetes) is too much
  - You end up with two problems:
    - Manage the service orchestration
    - Manage the orchestrator
- Full features cloud platform vs Simple cloud platform
  - Usually you use only a subset of functionality offered
  - The practitioners prefer simple and easy interface
  - The management are more inclined to approve the use of a cloud platform were costs are low and the pricing is clear

Why?

"What tools or technologies you use is irrelevant if the people who must use them hate use them, or if they don't archive the outcomes and enable the behaviors we care about."

Nicole Forsgren PhD, Jez Humble, Gene Kim, "Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations", p. 68



The Implementation

Application

The simple app example:

- Single Go binary
- Deployed on Github releases
- 1 attached database
- Follow the 12 Factor app principles:
  - **Codebase**: One codebase tracked in revision control, many deploys
  - **Config**: Store config in the environment
  - **Processes**: Execute the app as one or more stateless processes
  - **Disposability**: Maximize robustness with fast startup and graceful shutdown

Git Repository Layout

```
.
├── packer.json
├── provisioning
│   └── files
│       └──
app.service
└── terraform
    ├── database.tf
    ├── domains.tf
    ├── droplet.tf
    ├── image.tf
```

Systemd Unit File

```
[Unit]
Description=App server
After=network.target cloud-init.service

[Service]
Type=simple
User=root
EnvironmentFile=-/opt/app/conf.env
WorkingDirectory=/opt/app
Environment=GIN_MODE=release
ExecStart=/opt/app/app

[Install]
WantedBy=multi-user.target
```

Packer Configuration

```
{ "variables": {
        "url": "https://github.com/Amygos/immutable_deploys",
        "version": "v1"
},
  "builders": [{
        "type": "digitalocean",
        "image": "centos-7-x64",
        "region": "ams3",
        "size": "s-1vcpu-1gb",
        "ssh_username": "root",
        "snapshot_name": "app-{{user `version`}}-{{isotime \"2006/01/02-15:04:05\"}}"
  }],
  "provisioners": [{
        "type": "file",
        "source": "provisioning/files/app.service",
        "destination": "/usr/lib/systemd/system/app.service"},
     {"type": "shell",
        "inline": [
        "mkdir -p /opt/app",
        "curl -L {{ user `url` }}/releases/download/{{user `version`}}/app > /opt/app/app",
        "chmod 0755 /opt/app/app",
        "systemctl daemon-reload",
        "systemctl enable app"]}]}
```

Packer Output

```
==> digitalocean: Creating temporary ssh key for droplet...
==> digitalocean: Creating droplet...
==> digitalocean: Waiting for droplet to become active...
==> digitalocean: Using ssh communicator to connect: 178.62.207.7
==> digitalocean: Waiting for SSH to become available...
==> digitalocean: Connected to SSH!
==> digitalocean: Uploading provisioning/files/app.service => /usr/lib/systemd/system/app.service
==> digitalocean: Provisioning with shell script: /tmp/packer-shell648441204
==> digitalocean: Gracefully shutting down droplet...
==> digitalocean: Creating snapshot: app-v1-2020/01/25-22:07:03
==> digitalocean: Waiting for snapshot to complete...
==> digitalocean: Destroying droplet...
==> digitalocean: Deleting temporary ssh key...
Build 'digitalocean' finished.

==> Builds finished. The artifacts of successful builds are:
--> digitalocean: A snapshot was created: 'app-v1-2020/01/25-22:07:03' (ID: 58285042) in regions 'ams3'
```

Droplet Configuration

```
resource "digitalocean_droplet" "app" {
  image  = data.digitalocean_image.app.image
  name   = "app"
  region = "ams3"
  size   = "s-1vcpu-1gb"
  user_data = data.template_cloudinit_config.app.rendered

  lifecycle {
    create_before_destroy = true
  }
}

data "digitalocean_image" "app" {
  name = "app-v1-2020/01/25-22:07:03"
}
```

cloud-init User Data

```
data "template_cloudinit_config" "app" {
  gzip          = false
  base64_encode = false

  part {
    content_type = "text/cloud-config"
    content      = <<-EOT
    #cloud-config
    write_files:
      - path: /opt/app/conf.env
        content: |
          DB_HOST="${digitalocean_database_cluster.app.host}"
          DB_PORT="${digitalocean_database_cluster.app.port}"
          DB_USER="${digitalocean_database_cluster.app.user}"
          DB_PASSWORD="${digitalocean_database_cluster.app.password}"
          DB_NAME="${digitalocean_database_cluster.app.database}"
    EOT
  }
}
```

DNS records configuration

```
resource "digitalocean_domain" "app" {
  name = “example.com”
}

resource "digitalocean_record" "app" {
  domain = digitalocean_domain.app.name
  type   = "A"
  name   = "app"
  ttl    = "60"
  value = digitalocean_floating_ip.app.ip_address
}

resource "digitalocean_floating_ip" "app" {
  droplet_id = digitalocean_droplet.app.id
  region     = "ams3"
}
```

Database Configuration

```
resource "digitalocean_database_cluster" "app" {
  name       = "app"
  engine     = "pg"
  version    = "11"
  size       = "db-s-1vcpu-1gb"
  region     = "ams3"
  node_count = 1
}
```

Immutable Infrastructure Workflow

- Deploy new app version
  1. Build new image with packer
  2. Add it in the terraform configuration
  3. Apply the changes

- Modify the configuration
  1. Change or add new configuration to the cloud-init template
  2. apply the changes

Conclusions

Immutability trade-offs

Separate what is immutable from what is mutable, eg.;

Immutable resources

- Application code/binary
- graphical assets

Mutable resource

- Database
- HTTPS Certificates (yes, they are a mutable resource

The Benefits

Lowering the Deployment Pain

- **Simple provisioning**: you don't have to care about to previous state, every time is from scratch
- **Simple rollback**: most of the time is a simple git revert or git restore

Horizontal scalability

- The server are not unique anymore so you can easily scale
- Reproducibility
- All is automatized and tracked, you can easily reproduce a deployment and create a local environment

Further Steps

- Centralized Loging System
  - Graylog
  - ELK
  - Loki
- Centralized Monitoring System
  - Prometheus, Grafana
- Distribuite Tracing Tool
  - Jaeger
- Observability
  - Honeycomb
  - Tempo

But it is a recent idea?

Pp. 99-120 of the *Proceedings of LISA '02: Sixteenth Systems Administration Conference* (Berkeley, CA: USENIX Association, 2002).

Why Order Matters: Turing Equivalence in Automated Systems Administration

Steve Traugott – TerraLuna, LLC
Lance Brown – National Institute of Environmental Health Sciences

ABSTRACT

Hosts in a well-architected enterprise infrastructure are self-administered; they perform their own maintenance and upgrades. By definition, self-administered hosts execute self-modifying code. They do not behave according to simple state machine rules, but can incorporate complex feedback loops and evolutionary recursion.

The implications of this behavior are of immediate concern to the reliability, security, and ownership costs of enterprise and mission-critical computing. In retrospect, it appears that the same concerns also apply to manually-administered machines, in which administrators use tools that execute in the context of the target disk to change the contents of the same disk. The self-modifying behavior of both manual and automatic administration techniques helps explain the difficulty and expense of maintaining high availability and security in conventionally-administered infrastructures.

Thanks for listening!

Questions?

Matteo Valentini

Developer at Nethesis

 Amygos

 _Amygos

 Matteo Valentini

 matteo.valentini@nethesis.it