Immutable deployments: the new classic way for service deployment

Adopt the new immutable infrastructure paradigm using your old toolbox.

Matteo Valentini

@_Amygos
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 almost two years of using the methodology on a production service.
The Problems
The Problems: SnowflakeServer

THIS IS MY SERVER.

THERE ARE MANY LIKE IT, BUT THIS ONE IS MINE.
The Problems: SnowflakeServer

A server that is unique[1]:

- Manual installation
- Manual configuration
- Manual maintenance

*It is your server and you take care of it,*

*as you do with your pet.*

The Problems: Configuration Drift

The drift from a well-known start state, even if automated configuration tools are used[2]:

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

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.*

"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
The Problems: Not Deterministic Deployment

A → Deployment of B → A + B
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
Immutable Infrastructure: 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
Immutable infrastructure: 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
The Tools: Automated Provisioning

Shell scripts

- Almost every developer can understand it
- Simple and at the same time very powerful
The Tools: Image Builder

Packer

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

Terraform

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

DigitalOcean

- Not expensive
- Simple
- Have everything you need:
  - APIs
  - Compute instances
  - Snapshots
  - Cloud-init
  - Floating IPs
  - Load Balancers
The Tools: Why?

- **Container vs VM**
  - The VM are a more familiar concept.
  - Not all companies want or need to switch to containers.

- **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 companies, 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.
  - Practitioners prefer simple, easy interfaces and ways.
  - Management is more inclined to approve the use of a cloud platform where costs are low and the pricing is clear.
The Tools: 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.”

The Implementation
The application

The simple app example:

- Single Go binary
- Deployed on Github releases
- 1 attached database
- Follow the 12 Factor app principles[3]:
  - **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

[3]https://12factor.net/
Git Repository Layout

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

```hcl
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

```hcl
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"
}
```

---

@Amygos
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**
  a. Build new image with packer
  b. Add it in the terraform configuration
  c. Apply the changes

- **Modify the configuration**
  a. Change or add new configuration to the cloud-init template
  b. apply the changes
Conclusions
Conclusions: The Benefits

Lowering the Deployment Pain

- **Simple provisioning**: you don't have to care about 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 aren't unique anymore so you can easily scale

Reproducibility

All is automatized and tracked, you can easily reproduce a deployment and create a local environment
The benefits: Immutable Infrastructure Impacts

- Automated Provisioning
- Source Version Control
- Infrastructure as Code
- Immutable Infrastructure
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)
- ...

@Amygos
Conclusions: Further Steps

Centralized Logging System
- Graylog
- ELK
- ...

Centralized Monitoring System
- Prometheus, Grafana

Distributed Tracing Tool
- Jaeger
Thanks for listening!
Questions?

Matteo Valentini
Developer @ Nethesis (mostly Infrastrutture Developer)

Amygos
@_Amygos
amygos@paranoici.org, matteo.valentini@nethesis.it