Zarządzanie infrastrukturą jako kod

Infrastructure as Code (IaC)

mgr inż. Jakub Woźniak

Politechnika Poznańska
Wydział Informatyki i Telekomunikacji

Agenda wykładu

- Filozofia Infrastructure as Code
- Terraform – deklaratywne zarządzanie infrastrukturą
- Ansible – automatyzacja konfiguracji
- GitOps i Pulumi
- Zaawansowane wzorce
- Podsumowanie i Q&A

Problem: tradycyjne zarządzanie infrastrukturą

Typowe wyzwania:

- Manualna konfiguracja przez konsolę WWW
- *Snowflake servers* – każdy serwer unikalny, trudny do odtworzenia
- *Configuration drift* – środowiska rozjeżdżają się w czasie
- Brak wersjonowania i historii zmian
- Długi czas wdrożenia nowego środowiska
- Błędy ludzkie wynikające z operacji manualnych

Rozwiązanie: Infrastructure as Code

Definicja: *Infrastructure as Code* to praktyka definiowania i zarządzania infrastrukturą IT za pomocą kodu maszynowo-czytelnego, zamiast procesów manualnych i narzędzi interaktywnych. **Kluczowe zasady:**

- Deklaratywność – opisujemy co, nie jak
- Wersjonowanie – infrastruktura w Git
- Powtarzalność – łatwe odtwarzanie środowisk
- Automatyzacja – provisioning i konfiguracja przez narzędzia

Korzyści z IaC

Operacyjne:

- Szybki deployment (minuty vs dni)
- Spójność między środowiskami
- Łatwe skalowanie infrastruktury
- Self-service dla zespołów

Bezpieczeństwo i compliance:

- Code review dla zmian infrastruktury
- Audit trail w historii Git
- Policy as Code
- Disaster recovery w minuty

Kategorie narzędzi IaC

- **Infrastructure provisioning** – tworzenie zasobów chmurowych (VM, sieci, storage): Terraform, Pulumi, CloudFormation
- **Configuration management** – konfiguracja systemów i aplikacji: Ansible, Chef, Puppet, SaltStack
- **Container orchestration** – zarządzanie kontenerami: Kubernetes, Docker Swarm, ECS
- **Image building** – tworzenie immutable images: Packer, Docker, Buildpacks

Deklaratywne vs imperatywne podejście

Imperatywne (jak?)

```
# Bash script
aws ec2 run-instances \
  --image-id ami-123 \
  --instance-type t3.medium

aws ec2 create-tags \
  --resources i-456 \
  --tags Key=Name,Value=web
```

- Opisujemy kroki
- Musimy obsłużyć stany
- Trudniej o idempotentność

Deklaratywne (co?)

```
# Terraform
resource "aws_instance" "web" {
  ami           = "ami-123"
  instance_type = "t3.medium"
  tags = {
    Name = "web"
  }
}
```

- Opisujemy stan docelowy
- Narzędzie zarządza stanem
- Wbudowana idempotentność

IaC: dobre praktyki

- Wersjonuj wszystko (infrastruktura, konfiguracje, polityki)
- Małe, inkrementalne zmiany zamiast dużych wdrożeń
- Code review (PR) dla każdej zmiany
- Automatyczne testy (unit, integration, policy)
- Immutability – preferuj wymianę nad zmianą in-place
- Bezpieczne zarządzanie sekretami (Vault, SSM, KMS)
- Modularność – wielokrotnego użytku moduły
- Dokumentacja jako kod (README, komentarze, generowane docs)

Terraform – wprowadzenie

Terraform:

- Open-source narzędzie IaC od HashiCorp (od 2014)
- Multi-cloud: AWS, Azure, GCP, on-prem, SaaS
- Deklaratywny język HCL (HashiCorp Configuration Language)
- Architektura oparta o providery (tysiące providerów)

Kiedy używać?

- Provisioning infrastruktury chmurowej (VM, sieci, bazy)
- Złożone zależności i multi-cloud

Architektura Terraform

```
.tf files  ->  Terraform Core  ->  Providers  ->  Cloud APIs

State file (terraform.tfstate) śledzi
mapowanie zasobów i ich aktualny stan.
```

- Konfiguracje w plikach `.tf`
- Silnik Terraform (plan, apply)
- Providery dla chmur i usług
- Plik stanu (local, S3, Terraform Cloud)

State w Terraform

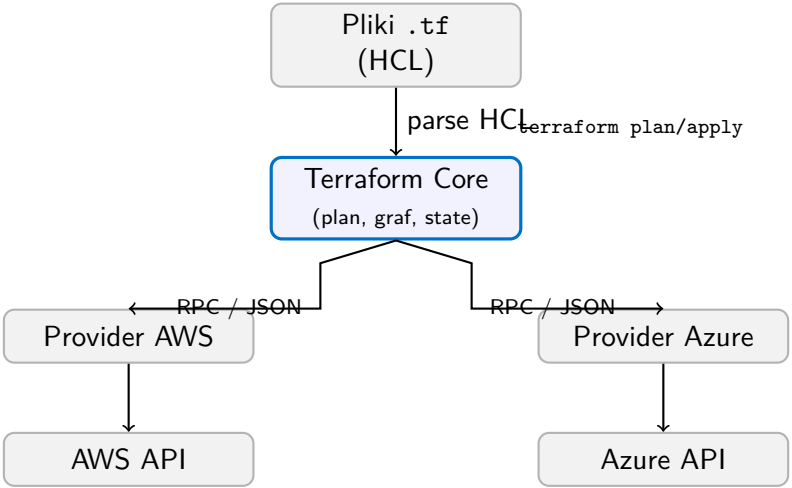
Dlaczego potrzebny jest state?

- Mapowanie zasobów w kodzie na realne zasoby w chmurze
- Lepsza wydajność (cache zamiast ciągłych wywołań API)
- Zarządzanie grafem zależności

Przykładowe backendy:

- `local` – domyślnie, do dev/test
- `s3` – produkcja i współpraca w zespole (z blokadą DynamoDB)
- Terraform Cloud – zarządzany state, UI, remote execution

Terraform Core vs Provider – diagram



Terraform Core *tylko* liczy plan i woła providery,

Typowy workflow Terraform

```
# 1. Inicjalizacja (pobranie providerów)
    $ terraform init
# 2. Formatowanie
    $ terraform fmt
# 3. Walidacja
    $ terraform validate
# 4. Plan (dry-run)
    $ terraform plan
# 5. Wdrożenie zmian
    $ terraform apply
# 6. Podgląd stanu
    $ terraform show
# 7. Usunięcie infrastruktury
    $ terraform destroy
```

HCL – podstawy składni

Provider i resource

```
# Provider
provider "aws" {
  region = "eu-west-1"
}

# Resource
resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"

  tags = {
    Name        = "main-vpc"
    Environment = "production"
  }
}
```

Data source

```
# Data source
data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"]

  filter {
    name   = "name"
    values = [
      "ubuntu/images/hvm-ssd/
       ubuntu-jammy-22.04-*"
    ]
  }
}
```

Przykład: klaster EKS na AWS

```
module "eks" {
  source  = "terraform-aws-modules/eks/aws"
  version = "~> 19.0"
  cluster_name    = "my-k8s-cluster"
  cluster_version = "1.28"
  vpc_id     = module.vpc.vpc_id
  subnet_ids = module.vpc.private_subnets
  eks_managed_node_groups = {
    general = {
      desired_size = 3
      min_size     = 2
      max_size     = 5
      instance_types = ["t3.medium"]
      capacity_type  = "SPOT"
    }
```

Zmienne i wyjścia w Terraform

Variables

```
# variables.tf
variable "environment" {
  type        = string
  default     = "dev"
}
variable "instance_count" {
  type    = number
  default = 3
}
resource "aws_instance" "app" {
  count = var.instance_count
}
```

Outputs

```
# outputs.tf
output "cluster_endpoint" {
  value = module.eks.cluster_endpoint
}

# CLI
$ terraform output cluster_endpoint
```

Drift detection w Terraform

- Drift = różnica między *desired state* (kod) a *actual state* (chmura)
- Źródła: zmiany manualne w konsoli, inne procesy, race conditions

```
# Wykrycie drifta
$ terraform plan -refresh-only

# Naprawa - doprowadzenie realnego stanu do docelowego
$ terraform apply
```

Ansible – przypomnienie

- Agentless – SSH/WinRM, brak agentów na maszynach docelowych
- Imperatywny, ale idempotentny
- Playbooki w YAML
- Bogaty ekosystem modułów

Terraform vs Ansible

- Terraform – provisioning infrastruktury
- Ansible – konfiguracja systemów i aplikacji

Struktura playbooka Ansible

```
tasks:
  - name: Install nginx
    apt:
      name: nginx
      state: present
      update_cache: yes

  - name: Copy config file
    template:
      src: nginx.conf.j2
      dest: /etc/nginx/nginx.conf
    notify: restart nginx
```

Idempotentność w Ansible

Cel: wielokrotne uruchomienie playbooka daje ten sam rezultat.

Nie-idempotentny przykład:

```
- name: Add line
  shell: |
    echo "export PATH=/app/bin:$PATH" \
      >> ~/.bashrc
```

Idempotentny przykład:

```
- name: Add line
  lineinfile:
    path: ~/.bashrc
    line: 'export PATH=/app/bin:$PATH'
    state: present
```

Best practice: używaj modułów zamiast `shell`/`command`.

Zmienne i szablony (Jinja2)

Variables

```
# group_vars/webservers.yml
app_name: myapp
app_port: 8080
environment: production

# Użycie
- name: Start app
  shell: |
    ./{{ app_name }} \
      --port {{ app_port }}
```

Templates

```
server {
  listen {{ app_port }};
  server_name {{ inventory_hostname }};
}

- name: Deploy config
  template:
    src: config.j2
    dest: /etc/app/config.conf
```

GitOps – Git jako źródło prawdy

Definicja: GitOps używa repozytorium Git jako *single source of truth* dla deklaratywnej infrastruktury i aplikacji. **Zasady GitOps:**

- Deklaratywność – cały desired state w Git
- Wersjonowanie i niezmienność – historia Git = audit trail
- Operatorzy (np. ArgoCD, Flux) pobierają zmiany z Git
- Ciągła obserwacja stanu i korekta driftów.

Workflow GitOps

```
Git repo (manifesty)
    |
    |  git push
    v
Git hook / CI
    |
    v
Operator GitOps (ArgoCD, Flux)
    |
    |  pull & apply
    v
Kubernetes cluster (deployment)
```

Korzyści: zero-touch deployment, rollback przez git revert, pełna audytowalność.

Pulumi – nowoczesne IaC

Cechy Pulumi:

- Języki ogólnego przeznaczenia: Python, TypeScript, Go, C# itd.
- Brak własnej DSL – pełna moc języka
- Multi-cloud i wsparcie dla Kubernetes, Dockera, Helm
- Łatwe testowanie (unit/integration) i refaktoryzacja

Pulumi vs Terraform:

- Terraform – HCL, moduły
- Pulumi – klasy, funkcje, pakiety, IDE/IntelliSense

Immutable infrastructure

Idea: serwery nie są modyfikowane po wdrożeniu – każda zmiana = nowy serwer/obraz.

Podejście mutable

- Aktualizacje in-place
- Configuration drift
- Snowflake servers

Podejście immutable

- Wymiana zasobów przy zmianie
- Brak drifta
- Łatwe odtwarzanie

Implementacja: kontenery Docker, obrazy VM (Packer), blue-green deployments.

Drift detection – strategia

- Terraform: `terraform plan -refresh-only` cyklicznie w CI
- Ansible: `ansible-playbook --check --diff`
- GitOps (ArgoCD): ciągła rekonsyliacja i alerty
- Narzędzia chmurowe (np. AWS Config) do compliance

Praktyka: alertuj o drifcie, automatycznie naprawiaj zmiany niekrytyczne.

Policy as Code – OPA

- Polityki bezpieczeństwa i compliance jako kod
- Walidacja konfiguracji przed wdrożeniem

```
# policy.rego
package terraform.analysis

# Zakaz publicznych bucketów S3
deny[msg] {
  resource := input.resource_changes[_]
  resource.type == "aws_s3_bucket"
  resource.change.after.acl == "public-read"
  msg := sprintf("S3 bucket %v must not be public",
                 [resource.name])
}
```

Zarządzanie sekretami

Czego nie robić:

- Nie trzymamy kluczy API i haseł w repo (ani w `providerach`)

Dobre praktyki:

- Menedżery sekretów chmurowych (SSM Parameter Store, Secrets Manager, Secret Manager)
- HashiCorp Vault jako centralny magazyn sekretów
- SOPS/Sealed Secrets dla K8s
- Sekrety przekazywane z CI/CD jako zmienne środowiskowe

Najczęstsze pułapki

- Brak backupu state i wersjonowania
- Hardkodowanie wartości zamiast użycia zmiennych
- Apply bez wcześniejszego planu
- Zbyt duży *blast radius* jednego state
- Ignorowanie drifta i braku regularnych checków
- Słabe zarządzanie sekretami
- Brak testów i środowisk pośrednich (dev/stage)

Podsumowanie i dalsza praktyka

- IaC jako fundament nowoczesnego DevOps i chmury
- Terraform do provisioningu, Ansible do konfiguracji
- GitOps i Pulumi jako kolejny krok w dojrzałości
- Zaawansowane wzorce: immutable infra, Policy as Code, drift detection

Pytania?

Dziękuję za uwagę!

Pytania, komentarze, dyskusja.