Engineer

immutable infrastructure

❄ at udskifte servere i stedet for at ændre dem på stedet

Hvad det er

Immutable infrastructure er en måde at drive servere på, hvor en maskine aldrig ændres, når den først er sat i drift. På dansk møder man også betegnelsen uforanderlig infrastruktur. HashiCorps Well-Architected Framework beskriver det som infrastruktur, der aldrig ændres efter udrulningen, men kun udskiftes. En ny version af applikationen, et opgraderet bibliotek, en sikkerhedsrettelse og en ændret indstilling kommer alle ad samme vej: som en ny maskine eller container, bygget ud fra et nyt image, der overtager den gamles plads. Derefter slukkes den gamle og slettes.

Det modsatte er mutable infrastructure, hvor en server lever længe og opdateres, der hvor den står. I juni 2013 beskrev Chad Fowler, hvad sådan en server bliver til efter år med opgraderinger: patchet igen og igen, med konfigurationsændringer foretaget i hånden midt under et nedbrud og med cronjobs, som kun én person kender til, indtil holdet hverken tør ændre den eller skifte den ud. Hans løsning, som han kaldte immutable deployments, var at bygge hvert system automatisk og derefter lade det være urørt. En opgradering eller en ny version betyder så, at man bygger en ny server eller et nyt image og smider den gamle væk.

To slags images gør tilgangen praktisk mulig. Et maskinimage er ifølge Packers dokumentation en enkelt statisk enhed med et forudkonfigureret styresystem og den installerede software. Packer selv er et community-værktøj, der laver tilsvarende maskinimages til flere platforme ud fra én kildekonfiguration. Containerimages er den anden slags, og Dockers dokumentation siger det ligeud: når et image først er oprettet, kan det ikke ændres, og man kan kun lave et nyt image eller lægge ændringer ovenpå som nye lag.

Begrebet beskriver en måde at drive et system på, ikke et produkt, man installerer. Det har også en grænse. Applikationsserverne kan udskiftes ved hver release, men de data, de arbejder med, skal ligge et sted, der overlever hver udskiftning, og det er ved den grænse, det meste af det praktiske designarbejde ligger.

Brug begrebet i en beslutning

Brug »Hvad er immutable infrastructure?« i en konkret beslutning. Vi kan knytte begrebet til det system, I bygger eller køber. Hele opsætningen skrevet ned som filer, så den kan genskabes præcist og aldrig kun findes i én persons hoved.

Se vores ydelser Kontakt os →

Undersøg antagelserne bag ordet

Brug forklaringen og det tilknyttede arbejde til at formulere spørgsmål til jeres projekt. Vi kan undersøge antagelserne sammen, før et værktøj vælges. Har skrevet 1.000 RBAC-regler til Linux-servere — 1,000 adgangsregler skrevet for at lukke et sikkerhedshul

Har skrevet 1.000 RBAC-regler til Linux-servere Kontakt os →

Hvorfor det betyder noget

Den første grund er afdrift. HashiCorps framework siger det direkte: manuelle opdateringer og ændringer på stedet får servere til at glide væk fra deres oprindelige konfiguration med tiden. To servere, der blev sat op på samme måde med et års mellemrum, ender ikke nødvendigvis ens, fordi hver af dem har fået sin egen række af rettelser, genforsøg og håndredigeringer. Når hver server er en frisk kopi af ét image, er imaget den fulde beskrivelse af serveren, og der er intet tilbage, der kan glide.

Den anden grund er tilbagerulning. Hvis en release opfører sig forkert, er frameworkets svar at udrulle det forrige artefakt igen i stedet for at reparere den nuværende infrastruktur. Det forrige image er en kendt tilstand, der allerede har kørt i produktion, så holdet skal ikke under pres rekonstruere, hvordan serveren så ud i går, ud fra hukommelsen og shell-historikken.

Den tredje grund er færre hænder på produktion. Frameworket påpeger, at det giver menneskelige fejl som tastefejl og glemte trin, når man logger ind på produktionsservere og opdaterer dem i hånden, og at immutable infrastructure fjerner behovet for manuel adgang til serverne. Færre personer har overhovedet brug for et login på en kørende maskine, og de, der har et, får sjældent brug for det.

Den fjerde grund er sporbarheden. Revisorer vil ifølge samme side have bevis for, at produktion ikke er ændret uden for de godkendte processer, og manuelle ændringer efterlader ufuldstændige spor. Når hver ændring er et nyt artefakt, viser versionsstyringen og CI/CD-loggene, hvad der blev ændret, hvornår, og hvem der godkendte det. En sikkerhedsrettelse følger samme vej som en ny funktion: den bliver et nyt artefakt, der erstatter det kørende, så den bygges, testes og registreres på samme måde.

Intet af det er gratis. Frameworket bemærker, at images til virtuelle maskiner kræver flere ressourcer end containere og er længere tid om at starte, mens containere kræver en orkestrator for at kunne køre. Hver ændring, også en indstilling på én linje, betyder nu et fuldt build og en udskiftning, så buildet skal være automatisk og hurtigt nok til, at ingen fristes til at springe det over. Tilstanden skal flyttes væk fra de udskiftelige maskiner før den første udskiftning, ikke bagefter.

Sådan ser det ud i praksis

En Packer-template beskriver, hvordan et maskinimage bygges. Denne her bygger på eksemplet i dokumentationen til Packers Amazon EBS-builder, med en `build` -blok tilføjet i den form, som referencen for build-blokken viser:

```
source "amazon-ebs" "web" {
  access_key    = var.aws_access_key
  secret_key    = var.aws_secret_key
  region        = "us-east-1"
  source_ami    = "ami-fce3c696"
  instance_type = "t2.micro"
  ssh_username  = "ubuntu"
  ami_name      = "web {{timestamp}}"
}

build {
  sources = ["source.amazon-ebs.web"]

  provisioner "shell" {
    scripts = fileset(".", "scripts/{install,secure}.sh")
  }
}
```

`source` -blokken angiver builderen og basisimaget, `source_ami` , hvis rodvolumen kopieres og provisioneres på en kørende instans. `build` -blokken siger, hvilke kilder der skal bygges, og hvordan de skal provisioneres. Shell-provisioneren

kører de to scripts inde i den kørende maskine, før den bliver gjort til et statisk artefakt, og artefaktet er et nyt image. Erklæringerne af de to variabler er udeladt her. Én kommando kører alle builds i templaten og laver artefakterne:

```
packer build web.pkr.hcl
```

Udrulningen er den anden halvdel. HashiCorps framework stiller de to værktøjer side om side: images bygges med Packer, udrulles med Terraform og udskiftes helt, når de skal opdateres. Når en ændring ikke kan foretages på stedet, er Terraforms standard ifølge referencen for lifecycle at slette det eksisterende objekt og derefter oprette erstatningen. Opdateringer i dette forløb kræver ikke login på en kørende produktionsserver. En ændring i scriptene giver et nyt image, imaget giver nye maskiner, og de gamle fjernes.

Samme disciplin virker også på en enkelt vært. Porteføljeeksemplet om containere på Podman og Quadlet holder provisionering og udrulning adskilt. Root forbereder udrulningsstien én gang, og derefter klares udgivelserne af en almindelig bruger uden særlige rettigheder, som må køre præcis ét script og intet andet. En udgivelse er et nyt image efterfulgt af en genstart af tjenesten, uden root-shell og uden frit valgte kommandoer. Eksemplet om CI/CD med GitHub Actions går længere på image-siden: frontenden i produktion kører på et distroless-grundlag, der hverken har en shell eller en pakkehåndtering, og det er workflowene, ikke en person, der fører et build videre fra ét miljø til det næste. Et image af den slags har næsten intet at logge ind på og intet værktøj at installere noget med, så at bygge dets afløser er den eneste fornuftige måde at ændre det på.

Siden om servere, I kan bygge op fra bunden igen, viser den anden model udført med omhu. Ansible bringer en kørende vært i den tilstand, repositoryet beskriver, hvert play er idempotent, og repositoryet leverer konfigurationen til at genopbygge det erklærede miljø. Siden lægger ikke skjul på, hvad ændringer på stedet koster. Alt tager længere tid end en direkte rettelse på serveren, og en kørsel, der stopper halvvejs, efterlader værten i en dårligere stand end en kørsel, der afviser at begynde. Derfor kører et preflight-play nu forud for den. En vært, der kun er halvt ændret, er et skridt på vejen mod den skrøbelige server, Fowler beskrev, og at skifte maskinen ud i stedet for at ændre den er den anden vej uden om.

Hvor det går galt

Modellen er enkel at formulere og let at udhule. De fleste fejl skyldes én undtagelse, der virkede harmløs, da den blev gjort.

En hurtig rettelse over SSH bringer afdriften tilbage. Under pres logger nogen ind på en kørende server og retter en fil. Rettelsen virker, indtil næste udrulning udskifter maskinen, og så er den væk sammen med enhver forklaring på, hvorfor den var nødvendig. Indtil da passer den kørende server ikke længere til sit image, og det er netop den tilstand, tilgangen skal forhindre. Løsningen er at sende hver ændring, også den hastende, gennem image-buildet og holde buildet hurtigt nok til at være den hurtigste vej. HashiCorps vejledning er aldrig at logge ind på kørende servere for at lave manuelle ændringer eller køre opdateringskommandoer.

Data, der skrives på maskinen, slettes sammen med den. Dockers dokumentation siger, at når en container slettes, forsvinder de filændringer, der er lavet inde i den, også. Uploads, genererede filer eller en lokal database på en udskiftelig maskine forsvinder ved næste release. Løsningen er at holde tilstanden uden for den del, der udskiftes. Docker-volumes gemmer data ud over den enkelte containers levetid, og en ny container, der kobles til samme volume, finder filerne igen; for maskiner udfylder en separat disk eller en databasetjeneste uden for imaget samme rolle.

Udskiftningen tager tjenesten ned. Terraforms standardrækkefølge er først at slette det eksisterende objekt og derefter oprette erstatningen, og det efterlader et vindue, hvor intet kører. Indstillingen `create_before_destroy` i en ressources `lifecycle` -blok vender rækkefølgen om:

```
resource "aws_instance" "web" {
  # ...
  lifecycle {
    create_before_destroy = true
  }
}
```

HashiCorp gør det til et tilvalg, fordi mange objekttyper har krav om unikke navne eller andre begrænsninger, som to eksemplarer på samme tid ville bryde, og referencen beder en om at kende begrænsningerne for hver ressourcetype, før man stoler på det.

Et nyt build starter i det stille fra en nyere base. I stedet for en fast `source_ami` kan Amazon EBS-builderen udfylde den fra et `source_ami_filter`, og med `most_recent` sat til true vælger filteret det nyeste image, der matcher, hvilket dokumentationen kalder mest nyttigt til et dagligt distro-build. Det er praktisk, hvis man vil holde sig opdateret, men så starter to builds af den samme template på forskellige dage fra forskellige baser, og en gammel release bliver ikke den samme, når den bygges igen. Løsningen for release-images er at fastlåse `source_ami` til et bestemt image og flytte det frem med vilje, i et commit, der er gennemgået.

Hemmeligheder bliver bagt ind i imaget. HashiCorps framework er klart: artefakter må aldrig indeholde hemmeligheder som databaseadgangskoder, API-nøgler eller krypteringsnøgler, fordi alle med adgang til artefaktet kan trække dem ud, og fordi rotation af en adgangsoplysning kræver, at hvert artefakt bygges og udrulles igen. Løsningen er at hente hemmelighederne, når maskinen eller containeren starter. HashiCorps eget mønster bruger Vault: en maskine, der starter, autentificerer sig og får midlertidige adgangsoplysninger, der udløber, og adgangsoplysningerne roteres i Vault, uden at noget skal bygges om. Templaten ovenfor følger samme regel for sine egne nøgler, som kommer ind som variabler og ikke står som tekst i filen.

Det forrige artefakt er væk, når der er brug for det. Tilbagerulning betyder at udrulle det forrige artefakt igen, og det forudsætter, at det stadig findes. Et oprydningsjob, der sletter gamle images, eller en vane med at genbruge ét image-navn til hvert build gør en tilbagerulning til en genopbygning fra gammel kildekode på en base, der kan have flyttet sig. Løsningen er at beholde imagene fra de seneste releases, der har kørt i produktion, give hvert af dem et navn, der aldrig genbruges, og øve en tilbagerulning til et af dem, før den dag det gælder.

Læs mere

- Create immutable infrastructure — Well-Architected Framework om afdrift, tilbagerulning og hemmeligheder.
- Trash Your Servers and Burn Your Code: Immutable Infrastructure and Disposable Components — Chad Fowlers indlæg fra juni 2013, læst i en kopi hos Internet Archive.
- Introduction to Packer — hvad Packer bygger, og hvad et maskinimage indeholder.
- Amazon EBS builder — builderen fra eksemplet, med `source_ami` og `source_ami_filter`.

- lifecycle meta-argument reference — Terraforms standard med slet og opret, og `create_before_destroy`.
- Persisting container data — hvorfor filer i en container forsvinder med den, og hvordan volumes hjælper.

Ordliste

Vælg ud fra jeres faktiske vilkår

Medbring udrulningsforløbet, driftsrisiciene og dem, der har vagten. Vi kan afgrænse en ændring med kontroller, en vej tilbage og tydeligt ansvar.

Se, hvad det koster Kontakt os →

Gør begreberne praktiske

Spørg, hvordan »Hvad er immutable infrastructure?« gælder i jeres organisation. Et konkret eksempel fra jeres nuværende system er nok til at begynde samtalen.

Se vores ydelser

welcome@engineer.company +45 50 10 61 84 København, Danmark

Tag denne side med dig

Visitkort

Lavet med kærlighed af D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – nu · Engineer ApS · Alle rettigheder forbeholdes.

Udskrevet fra https://engineer.company/da/glossary/immutable-infrastructure/