Engineer

immutable infrastructure

❄ att byta ut servrar i stället för att ändra dem på plats

Vad det är

Immutable infrastructure är ett sätt att driva servrar där en maskin aldrig ändras när den väl har tagits i drift. På svenska används också benämningen oföränderlig infrastruktur. HashiCorps Well-Architected Framework beskriver det som infrastruktur som aldrig ändras efter driftsättningen, utan bara byts ut. En ny version av applikationen, ett uppgraderat bibliotek, en säkerhetsuppdatering och en ändrad inställning kommer alla samma väg: som en ny maskin eller container, byggd från en ny avbildning (image), som tar den gamlas plats. Sedan stängs den gamla av och raderas.

Motsatsen är mutable infrastructure, där en server lever länge och uppdateras där den står. I juni 2013 beskrev Chad Fowler vad en sådan server blir efter år av uppgraderingar: patchad gång på gång, med konfigurationsändringar gjorda för hand mitt under ett avbrott och med cronjobb som bara en person känner till, tills teamet varken vågar ändra den eller byta ut den. Hans lösning, som han kallade immutable deployments, var att bygga varje system automatiskt och sedan låta det vara orört. En uppgradering eller en ny version innebär då att man bygger en ny server eller en ny avbildning och kastar den gamla.

Två slags avbildningar gör arbetssättet praktiskt möjligt. En maskinavbildning är enligt Packers dokumentation en enda statisk enhet med ett förkonfigurerat operativsystem och den installerade programvaran. Själva Packer är ett communityverktyg som skapar motsvarande maskinavbildningar för flera plattformar från en enda källkonfiguration. Containeravbildningar är det andra slaget, och Dockers dokumentation säger det rakt ut: när en avbildning väl har skapats kan den inte ändras, och man kan bara skapa en ny avbildning eller lägga ändringar ovanpå den som nya lager.

Begreppet beskriver hur ett system drivs, inte en produkt man installerar. Det har också en gräns. Applikationsservrarna kan bytas ut vid varje release, men de data de arbetar med måste ligga någonstans som överlever varje utbyte, och det är vid den gränsen det mesta av det praktiska designarbetet ligger.

Använd begreppet i ett beslut

Använd ”Vad är immutable infrastructure?” i ett konkret beslut. Vi kan knyta begreppet till systemet ni bygger eller köper. Hela uppsättningen nedskriven som filer, så att den kan återskapas exakt och aldrig bara finns i en enda persons huvud.

Se våra tjänster Hör av er →

Granska antagandena bakom ordet

Använd förklaringen och det länkade arbetet för att formulera frågor till projektet. Vi kan undersöka antagandena tillsammans innan ett verktyg väljs. Skrev 1 000 RBAC-regler som säkrar Linux-servrar — 1,000 åtkomstregler, skrivna för att täppa till en säkerhetslucka

Skrev 1 000 RBAC-regler som säkrar Linux-servrar Hör av er →

Varför det spelar roll

Det första skälet är att servrar glider isär. HashiCorps ramverk säger det rakt ut: manuella uppdateringar och ändringar på plats får servrar att med tiden avvika från sin ursprungliga konfiguration. Två servrar som sattes upp likadant med ett års mellanrum blir inte nödvändigtvis lika, eftersom var och en har fått sin egen följd av rättelser, nya försök och handredigeringar. När varje server är en färsk kopia av en enda avbildning är avbildningen den fullständiga beskrivningen av servern, och det finns ingenting kvar som kan glida isär.

Det andra skälet är att det går att rulla tillbaka. Om en release beter sig fel är ramverkets svar att driftsätta den föregående artefakten på nytt i stället för att reparera den nuvarande infrastrukturen. Den föregående avbildningen är ett känt tillstånd som redan har körts i produktion, så teamet behöver inte under press återskapa hur servern såg ut i går, ur minnet och skalhistoriken.

Det tredje skälet är färre händer i produktion. Ramverket påpekar att den som loggar in på produktionsservrar och uppdaterar dem för hand bjuder in mänskliga fel som stavfel och glömda steg, och att immutable infrastructure tar bort behovet av manuell åtkomst till servrarna. Färre personer behöver över huvud taget en inloggning på en maskin i drift, och de som har en behöver sällan använda den.

Det fjärde skälet är spårbarheten. Revisorer vill enligt samma sida ha belägg för att produktionen inte har ändrats utanför de godkända processerna, och manuella ändringar lämnar ofullständiga spår. När varje ändring är en ny artefakt

visar versionshanteringen och CI/CD-loggarna vad som ändrades, när och vem som godkände det. En säkerhetsuppdatering följer samma väg som en ny funktion: den blir en ny artefakt som ersätter den som körs, så den byggs, testas och registreras på samma sätt.

Inget av detta är gratis. Ramverket noterar att avbildningar för virtuella maskiner kräver mer resurser än containrar och tar längre tid att starta, medan containrar kräver en orkestrerare för att köras. Varje ändring, även en inställning på en rad, innebär nu ett fullständigt bygge och ett utbyte, så bygget måste vara automatiskt och snabbt nog för att ingen ska frestas att hoppa över det. Tillståndet måste flyttas bort från de maskiner som byts ut före det första utbytet, inte efteråt.

Så ser det ut i praktiken

En Packer-mall beskriver hur en maskinavbildning byggs. Den här bygger på exemplet i dokumentationen för Packers Amazon EBS-builder, med ett `build` -block tillagt i den form som referensen för build-blocket visar:

```
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` -blocket anger buildern och basavbildningen, `source_ami` , vars rotvolym kopieras och provisioneras på en körande instans. `build` -blocket anger vilka källor som ska byggas och hur de ska provisioneras. Shell-provisioneraren kör de två skripten inne i den körande maskinen innan den görs om till en statisk

artefakt, och artefakten är en ny avbildning. Deklarationerna av de två variablerna är utelämnade här. Ett enda kommando kör alla byggen i mallen och skapar artefakterna:

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

Driftsättningen är den andra halvan. HashiCorps ramverk ställer de två verktygen sida vid sida: avbildningar byggs med Packer, driftsätts med Terraform och byts ut helt när de behöver en uppdatering. När en ändring inte kan göras på plats är Terraforms standard enligt referensen för lifecycle att först ta bort det befintliga objektet och sedan skapa ersättningen. Uppdateringar i det här flödet kräver ingen inloggning på en produktionsserver som är i drift. En ändring i skripten ger en ny avbildning, avbildningen ger nya maskiner, och de gamla tas bort.

Samma disciplin fungerar också på en enda värd. Portföljexemplet om containrar på Podman och Quadlet håller provisionering och driftsättning isär. Root lägger upp driftsättningsvägen en enda gång, och sedan sköter en vanlig användare utan särskilda rättigheter utrullningarna; användaren får köra exakt ett skript och inget annat. En utrullning är en ny avbildning följd av en omstart av tjänsten, utan root-skal och utan fritt valda kommandon. Exemplet om CI/CD med GitHub Actions går längre på avbildningssidan: frontenden i produktion körs på en distroless-grund som varken har skal eller pakethanterare, och det är arbetsflödena, inte en person, som för ett bygge vidare från en miljö till nästa. En sådan avbildning har nästan inget att logga in på och inget verktyg att installera med, så att bygga en efterträdare är det enda rimliga sättet att ändra den.

Sidan om servrar som går att bygga upp från grunden igen visar den andra modellen utförd med omsorg. Ansible för en värd som är i drift till det tillstånd som repot beskriver, varje play är idempotent och repot innehåller konfigurationen för att bygga upp den deklarerade miljön igen. Sidan döljer inte vad ändringar på plats kostar. Allt tar längre tid än att bara ändra direkt på servern, och en körning som stannar halvvägs lämnar värden i sämre skick än en körning som vägrar starta. Därför körs nu en preflight-play före den. En värd som bara är halvt ändrad är ett steg mot den sköra server Fowler beskrev, och att byta ut maskinen i stället för att ändra den är den andra vägen runt.

Var det går fel

Modellen är enkel att formulera och lätt att urholka. De flesta fel beror på ett enda undantag som verkade ofarligt när det gjordes.

En snabb rättelse över SSH får servrarna att glida isär igen. Under press loggar någon in på en server i drift och redigerar en fil. Rättelsen fungerar tills nästa driftsättning byter ut maskinen, och då är den borta, tillsammans med varje förklaring till varför den behövdes. Fram till dess stämmer servern inte längre med sin avbildning, och det är precis det tillstånd som arbetssättet finns till för att förhindra. Lösningen är att skicka varje ändring, även den brådskande, genom avbildningsbygget och hålla bygget så snabbt att det är den snabbaste vägen. HashiCorps vägledning är att aldrig logga in på servrar i drift för att göra manuella ändringar eller köra uppdateringskommandon.

Data som skrivs på maskinen raderas tillsammans med den. Dockers dokumentation säger att när en container tas bort försvinner också de filändringar som gjorts inne i den. Uppladdningar, genererade filer eller en lokal databas på en maskin som byts ut försvinner vid nästa release. Lösningen är att hålla tillståndet utanför den del som byts ut. Docker-volymer bevarar data bortom livslängden för en enskild container, och en ny container som kopplas till samma volym hittar filerna kvar; för maskiner fylls samma roll av en separat disk eller en databastjänst som inte ingår i avbildningen.

Utbytet tar ner tjänsten. Terraforms standardordning är att först ta bort det befintliga objektet och sedan skapa ersättningen, vilket lämnar ett fönster då tjänsten står still. Inställningen `create_before_destroy` i `lifecycle` -blocket för en resurs vänder på ordningen:

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

HashiCorp gör detta till ett aktivt val, eftersom många objekttyper har krav på unika namn eller andra begränsningar som två exemplar samtidigt skulle bryta mot, och referensen ber dig förstå begränsningarna för varje resurstyp innan du förlitar dig på det.

Ett nytt bygge utgår i tysthet från en nyare bas. I stället för en fast `source_ami` kan Amazon EBS-buildern fylla i den från ett `source_ami_filter`, och med `most_recent` satt till true väljer filtret den nyaste avbildningen som uppfyller villkoren, vilket dokumentationen kallar mest användbart för ett dagligt distro-bygge. Det är praktiskt för att hålla sig uppdaterad, men då utgår två byggen av samma mall på olika dagar från olika baser, och en gammal release blir inte densamma när den byggs om. Lösningen för release-avbildningar är att låsa `source_ami` till en bestämd avbildning och flytta fram den medvetet, i en granskad commit.

Hemligheter bakas in i avbildningen. HashiCorps ramverk är tydligt: artefakter får aldrig innehålla hemligheter som databaslösenord, API-nycklar eller krypteringsnycklar, eftersom alla med åtkomst till artefakten kan plocka ut dem, och eftersom rotation av en inloggningsuppgift kräver att varje artefakt byggs och driftsätts på nytt. Lösningen är att hämta hemligheterna när maskinen eller containern startar. HashiCorps eget mönster använder Vault: en maskin som startar autentiserar sig och får tillfälliga inloggningsuppgifter som löper ut, och uppgifterna roteras i Vault utan att något behöver byggas om. Mallen ovan följer samma regel för sina egna nycklar, som kommer in som variabler och inte står som text i filen.

Den föregående artefakten är borta när den behövs. Att rulla tillbaka innebär att driftsätta den föregående artefakten på nytt, och det förutsätter att den fortfarande finns. Ett städjobb som raderar gamla avbildningar, eller en vana att återanvända ett och samma avbildningsnamn för varje bygge, gör det till ett nybygge från gammal källkod på en bas som kan ha flyttat sig. Lösningen är att behålla avbildningarna från de senaste releaserna som har körts i produktion, ge var och en ett namn som aldrig återanvänds och öva på att rulla tillbaka till en av dem innan dagen då det gäller.

Läs mer

- Create immutable infrastructure — ramverkets sida om avvikelser, återgång, avbildningstyper och hemligheter.
- Trash Your Servers and Burn Your Code: Immutable Infrastructure and Disposable Components — Chad Fowlers inlägg från juni 2013, läst i en kopia hos Internet Archive.
- Introduction to Packer — vad Packer bygger och vad en maskinavbildning innehåller.
- Amazon EBS builder — buildern från exemplet, med `source_ami` och `source_ami_filter`.

- lifecycle meta-argument reference — Terraforms standard att ta bort före skapande, och `create_before_destroy`.
- Persisting container data — varför filer i en container försvinner med den, och hur volymer bevarar dem.

Ordlista

Välj utifrån era faktiska förutsättningar

Ta med driftsättningsvägen, driftsriskerna och dem som har jour. Vi kan avgränsa en förändring med kontroller, en väg tillbaka och tydligt ansvar.

Se vad det kostar Hör av er →

Gör begreppen praktiska

Fråga hur ”Vad är immutable infrastructure?” gäller i er organisation. Ett konkret exempel från ert nuvarande system räcker för att börja samtalet.

Se våra tjänster

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

Ta med dig den här sidan

Visitkort

Omsorgsfullt skapad med kärlek av D.

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

Upphovsrätt © 2025 – nu · Engineer ApS · Alla rättigheter förbehållna.

Utskrivet från https://engineer.company/sv/glossary/immutable-infrastructure/