Assignment lab 03

YHL-18 CMT
By: Henrik Ählström

Task 1:

Observability:

- **Three Pillars of observability –** Loggar, metrics och tracing.
- **Blackbox vs Whitebox monitoring –** Skillnaden är hur mycket insyn man har I systemet som monitoreras. Whitebox syftar till att monitoreringen har I princip full insyn och kan moniterar ner på detaljnivå, vilket kan ge detaljerad information vid debugging, felsökning, benchmarks etc.
  Vid Blackbox monitorering å andra sidan har man ingen vetskap om hur ett system/applikation fungerar, man kan tänka sig att det är som att monitorera utifrån en end users perspektiv, tex man kan kolla att en server är uppe, att ett DNS uppslag som borde returnerar som det ska eller att en webbsida går att nå baserat på http svarskod med hjälp av tex ett probe script eller curl, men du kan tex inte fastställa att servern är nere pga att Nginx tjänsten har dött. Blackbox ger således endast vetskap om symptom och att ett fel uppstått, men inte vad det beror på.
- **Alerting –** En automatiskt genererad notifikation att något som avviker från det normala har inträffat. Ett alert kan nå ansvarig person via tex ett mail, ett meddelande eller läggas till som en ticket I någon form av kö för problemhantering.
- **Monitoring Signals –** Syftar till att avgränsa vad som ska monitoreras I ett system/en miljö. Finns olika “standarder/riktlinjer” som Golden och RED vilka ger tips om vad som är effektivt att prioritera när det kommer till monitorering I stora miljöer(särskilt vid distrubierade system). Självklart kan egna signalstrategier användas, dessa strategier bör förutom rent tekniska detaljer även ta hänsyn till företagsnytta, vad vill företaget att vi pritoriterar? Är uptime eller hastighet att prioritera?
- **Why do we need logs?** För att spara historik från system, I vissa fall finns det krav på att man ska spara användares historik(Tex hos en ISP) och dylikt. Loggar används även för att lokalisera felkällan vid felsökning/debugging samt säkerhetsrelaterade incidenter, tex audit loggar. Utan loggar hade det varit kaos, bokstavligen kaos, felsökning/debugging hade tagit betydligt längre tid. Någon obehörig tar sig in, ingen vet om det.
- **Logs and metrics –** I loggar finner vi human readable entries som blivit outputat av en applikation eller ett system.
  Metrics handlar mer om att samla in mätbar data exempelvis resursanvändning, allt från hårdvaruresurser till nätverksprestanda tex bandbredd, typiskt samlas datat in av någon typ av mjukvara och är väl egentligen inte tänkt att läsas direkt av människor, vanligt är att insamlad data sammanställs och visualiseras I form av grafer/diagram.
- **What’s tracing? -** Spårbarhet, används för att kartlägga den väg exempelvis en request går igenom ett system.

Service Mesh:

- **What’s service mesh? -** Ett lager som läggs på en existerande infrastruktur som har hand om kommunikation mellan services I en microservice arkitektur. Ett service mesh ämnar göra kommunikationen pålitlig genom att använda sig av saker som service discovery, circuit-breaking,lastbalansering med mera. En stor fördel med Service mesh är att det injicera en dos dynamik till ett system som annars (potentiellt) hade varit väldigt statiskt , exempelvis tar service mesh bort behovet av dedikerade loadbalancers, loadbalancers som måste anpassas för att överensstämma med en ever-changing topologi.

- **How does it enable it data plane?** - Med en proxy sidecar som har hand om service discovery, routing, observability med mera.
- **mTLS –** Multiplexed TLS – Bygger på att flera instanser(tex applikationer) kan dela en och samma TLS session, till exempel I en stor distrubierad miljö med många instanser av olika slag kan det vara suboptimalt att upprätta en enskild TLS session till varje berörd instans, istället möjliggör mTLS att flera services kan dela på en TLS session.
- **Circuit Breaking –** Ett designmönster för att bygga in feltolerans I distrubierade system. Circuit breaking är dock inte till för att lösa själva problemet, utan snarare se till att services I systemet snabbt blir notifierade om att den tjänsten de försöker kontakta ligger nere så att de eventuellt kan vidta plan B. Detta hindrar onödiga "loopar", dvs request skickas, får inget svar, ny request skickas, får inget svar osv osv.
  Detta betyder att istället för att services ska skicka request efter request till en tjänst som är död ska de snabbt notifieras att mottagaren är död och istället snabbt kunna gå vidare och på så sätt minska risken för att system får en "låsning" som följd av att det någonstans I kedjan står still, exempelvis en frontend applikations pool med trådar är tom för att backend servicen inte svarar(eller svarar väldigt långsamt),man har då "låst" systemet pga en felande backend.
  Man undviker även slöseri av resurser då man tex inte slösar bort nätverksresurser på att konstant göra remote calls som troligtvis ändå inte får svar, requesten hejdas och servicen som requesten kommer ifrån får ett response som antyder att något är fel.
  Circuit breaking ämnar göra mest nytta vid problem av allvarligare karaktär och inte tillfälliga hickups, dvs en fördröjning I nätverket lär inte hanteras av circuit breaking då någon av nästkommande requests troligtvis kommer komma fram utan problem och systemet fungerar normalt, men om tjänsten slutar svara helt och hållet lär circuit breaking vara till hjälp. Circuit breaking kan även notifiera services om att en service som tidigare var nere numera är uppe igen.
- **Traffic steering/splitting –**
- **Fault injection –** Ett sätt att utföra tester för att se hur ett system beter sig om ett fel inträffar. Detta görs genom att man medvetet ställer till trubbel och sedan utvärderar resultatet, av den anledningen kan man se det som en slags simulering.
- **Rolling update –** En metod för att uppdatera flertalet instanser/applikationer under en viss period, men inte alla samtidigt utan en(eller ett antal, beroende på systemets storlek) åt gången, Exempel: Vi har 3 instanser, vi börjar med att sluta ta emot requests dvs vi pekar trafiken **bort** från den som ska uppdateras, sedan tas den ur bruk, uppdateras, tas I bruk igen samt pekar återigen trafik till den. Sedan upprepas samma process på nästa instans osv tills alla är uppdaterade till önskad version. Man kan tänka lite som att man vill ta bort stolen man sitter på, fast här så vill man uppdatera utan att behöva stänga ner helt.

Serverless:
Pros:

- Om man använder en provider: Behöver inte betala när functionerna inte körs.
- Behöver inte ta hand om underliggande infrastruktur. Här är det viktigt att inte tänka “det behöver man ju inte med exempelvis en EC2 instans från AWS heller”. För jo, även om man inte behöver ansvara över hårdvara exempelvis med en EC2 instans har man endå fullt ansvar över den virtuella instansen, exempelvis: se till att den är uppdaterad, se till att den är härdad osv. Med serverless slipper man den biten också.
  Sen är ju frågan, kan det räknas som en fördel alltid? Vart drar man gränsen om man kör exempelvis OpenFaaS på egen hårdvara?
- Enkelt att skala.
- Sparar kostnader relaterat till personal?, Man överlåter ju nästan allt ops relaterat arbete till en provider.

Cons:

- Låst till provider, providerns regler gäller, bara rätta sig. Första tanken är ju då “då byter vi provider”, en applikation som är optimerad för en viss providers FaaS behöver inte nödvändigtvis vara enkel att porta till en annan, det går ju helt klart, men det krävs nog lite jobb.
- Svårare att debugga och felsöka, man har inte tillgång till den bakomliggande infrastrukturen.
- Cold starts kan vara ett problem, om applikationen tar för lång tid att starta upp om den gått I “viloläge” kan prestandan/responsen försämras.

Task 2:

Monitoring:

- **Grafana –** Grafana används för att presentera insamlad data visuellt på ett åskådligt sätt. Grafana kan få data från många olika källor tex Elasticsearch och Prometheus med många flera. Grafana är anpassningsbart, man kan skapa egna dashboards(eller använda färdiga) som presenterar relevant data. Grafan är bra därför att det kan framställa saker som metrics I en format som är enkelt att förstå för den som monitorerar.
- **Prometheus -** Samlar in metrics I tidsordning till en databas som andra program/applikationer sedan kan få av Prometheus via exportering(Tex Grafana). Ett bra sätt att samla metrics.
- **Sentry –** För monitorering(samt tracing) av applikationer post-deployment samt notifiera(alert) vid problem för att snabbt kunna identifiera felets context och frekvens.
- **Kibana –** Visualiseringsverktyg för Elasticsearch för att visualisera insamlad data tex loggar I grafer mm för att enkelt kunna få en överblick och för att se negativa trender.

Logging:

- **Fluentd –** En big data mjukvara för insamling och formatering av bland annat event(tex access/system)loggar och applikations loggar. Kan ta input från flera olika källor och outputa dessa I JSON format till flera destinations, output kan sedan användas av exempelvis någon analyseringsmjukvara eller för arkivering. Gör loggning mera uniform då den tar flera olika format In och outputar I ett gemensamt format.

- **Elasticsearch –** En sökmotor för att snabbt hitta data baserat på sökmönster, dvs applikationer kan query Elasticsearch efter data som Elasticsearch även har möjlighet att lagra.
  Ett usecase: om vi tänker ett scenario där vi lagrar loggar I Elasticsearch, och så har vi någon typ av analyserings/visualiseringsmjukvara som vi vill ska presentera allt som har med SSH att göra, med Elasticsearchs full-text-search kapabilitet får resultatet på vårat sökmönster väldigt snabbt.
  En key selling point är just möjligheten till full-text-search till skillnad från tex vissa SQL DMBS som inte har stöd för det(lika bra iaf).
- **Logstash –** Likt fluentd för hantering/insamling/processering av loggar och metrics. Kan likt fluentd få data från flera källor och skicka vidare till flera källor(i exempelvis JSON format). En intresant sak Logstash kan göra är *data enrichment* dvs Logstash kan lägga till relevant information baserat på vad som exempelvis står I logentries, tex: source IP står I log entries, Logstash kan ta reda på geografisk location kopplad till den IPn.

Tracing:

- **Jaeger –** Distrubierat tracing system som hjälper till att dels spåra trafik I distrubierade system samt har metoder för att hitta felkällan.
- **Zipkin –** Även detta ett distrubierat tracing system främst fokuserad på laterncy relaterade problem. Kan med hjälp av trace ID alternativt en query baserat på tex service söka I loggar och visa hur lång tid viss service tar att göra det den ska och om operationen den gjorde gick bra eller ej. Har också möjlighet att traca beroenden genom att visa hur många tracade requests rör sig genom varje service.
- **OpenTracing –** Till skillnad från ovanstående varianter är inte OpenTracing en specifik mjukvara, OpenTracing är istället ett vendor neutralt API som finns till flera programmeringsspråk.

Observability:

- **Istio –** Istio är ett service mesh och en platform som I contexten observability möjliggör ett uniformt sätt att monitorera distrubierade system med minimal anpassning I själva applikationens kod. Istio kan sköta insamling av metrics, loggar samt ha hand om tracing. Det är som en komplett lösning för att ge gåvan av observabiity till komplexa distrubierade system som annars är typiskt svåra att monitorera/traca. Istio kan implementeras som en sidvagn(i analogi med sidvagn på en cykel) till varje tjänst där “sidvagnen” har hand om bland annat det som omfattas av observability.
- **Elastic Stack –** Består av ett antal mjukvaror: Elastic Search, Logstash, Kibana och optionally även någon typ av beat(file eller metric exempelvis) samt X-trace.
  Används för att implementera observability I ett system.
  - Elastic Search – Är själva backenden I stacken, sparar data bland annat.
  - Logstash - hämtar(så vida en beat inte används, då sköter den hämtningen), modifierar eventuellt datat samt routar och filtrerar datat.
  - Kibana - visualiserar data I I grafer/diagram.
  - Beat – Olika beats har olika funktioner, två vanliga beats är:
    - Filebeat – Används för att hämta logentries
    - Metricbeat – Används för att hämta metrics.

Task 3 – Essay:

By: Henrik Ählström

Automation & Configuration

I denna essä valde jag att vrida lite på ämnet till att diskutera mycket kring Ansible vs Terraform eller varför Ansible+Terraform=Epic. Motiveringen till det är att jag syftar till att reda ut de frågetecken jag själv har och har haft I ämnet samt att om man söker hittar typ ämnen med rubriken “Terraform vs Ansible, which to use?”.

Om vi börjar med att fastställa skillnaden. Verktyg som Ansible, Saltstack, Chef och Puppet är ändå rätt lika varandra och nämns ofta I samma context,
men sen hör man talas om vertyg som Terraform, som inte nämns som det femte alternativet utan ibland som same same but different, varför är det så?

Ansible, Salt, Chef och Puppet är så kallade *config management tools* medans Terraform är ett *provisioning tool* eller *Orcestration tool* med förklaringen *infrastructure-as-code.*
För att inte dra ut för mycket kommer Ansible stå som representant för config management I essän från och med nu då det är det verktyg vi pratat om I kursen.

Terraforms styrka ligger I att enkelt kunna skapa virtuella instanser hos ett flertal providers exempelvis Google, Azure, AWS men även lokalt genom att lägga till en libvirt provider eller till VMware.

Ansibles styrka ligger att enkelt konfiguera instanser, genom att använda Playbooks där konfigurationen skrivs på ett sätt som är lätt att förstå och skapa sig en uppfattning om vad som gäller. Kan naturligtvis även användas mot infrastruktur I cloudet.

Om vi snabbt betar av en uppenbar skillnad vi tidigare diskuterat I kursen men som ändå har stor inverkan på varför slutsatsen av essän blir som den blir: Terraform är till skillnad från Ansible immutable, dvs ny konfiguration appliceras inte på en redan existerande instans, utan en ny instans skapas där ny konfiugrationen appliceras.
Terraform är också stateful, dvs Terraform håller koll på de instanser den har hand om I en *state file,* denna statefile är inte obetydlig, när en förändring sker exempelvis vi vill skala från 5 till 10 EC2 instanser jämför Terraform desired state med det state som återfinns I statefilen och startar I det här fallet ytterligare 5 instanser.

Eftersom Ansible är mutable appliceras förändringar på en redan existerande instans.
Ansible har dock inget sätt att hålla koll på states, detta medför att om vi tar samma exempel från ovan, vi vill skala från 5 till 10 EC2 instanser, och vi anger antalet 10, kommer Ansible starta 10 ytterligare instanser vilket ger totalen 15. Ok, men vi skriver bara 5 då så får vi 10? Jo visst men hur blir det när man vill skala ner dvs ta bort X antal instanser? It goes without saying att Terraform är enklare I denna typ av situtation.

Ytterligare en faktor att väga in är procedural och declarative syntax. Bägge dessa mjukvaror har fördelar förknippat med vilken typ av syntax de använder: Terraform som använder declarative, du behöver inte skriva allt för mycket info, utan “jag vill ha X antal EC2 instanser” osv och Ansible som använder Procedural, vilket kan vara en fördel om man tex har saker som beror på att något annat redan skett, exempelvis att Python bör installeras innan man försöker köra sitt Python script.

Exempel på varför vi inte bör försöka använda den ena till det andra, fast den andra faktiskt är bättre på det:

Om vi tänker oss att vi vill använda Terraform för config management då, Terraform är ju som tidigare nämnt immutable vilket kan leda till att små förändringar ibland kan få stora konsekvenser, Ett bisarrt exempel: vi har företagets namn på meddelandet som visas när man loggar in på en server, företaget har bytt namn, vi vill ändra bannern, känns det som en vettig approach att sänka alla servrar och återskapa dom med en ny image med en ny banner? Svar nej.

Ett annat exempel där immutable kan bita en, men som går att undvika med lite stil och finess är om du hanterar instanser där stateful applikationer körs, tänk att du exempelvis har Docker på en maskin, och så vid förändringar skrotas hela maskinen inklusive stateful datat och startas om på nytt. Detta går dock som sagt att lösa genom att externalisera lagringen av data(vilket I många fall kan anses best practice).

Om vi använder Ansible för provisioning, likt nämnt tidigare om vi kör en Playbook som ska säg skapa 10 instanser, för eller senare blir det svårt att hålla reda på vad som är vad. Man kan då använda count_taggar som håller koll på antal, exempel sätter vi en count_tagg till något namn och count till max 10 skapas 10 instanser med det namnet.
Men här vill jag knyta in till ett blogginlägg jag läste på: https://www.thoughtworks.com/insights/blog/why-configuration-management-and-provisioning-are-different där det här fenomenet förklaras, just att du kan provisionera med Ansible men det blir I slutändan mycket kod för att göra saker som anses enkla att göra I Terraform. Playbooks som likt tidigare nämnt ska ju vara relativt enkla att förstå blir plötsligt onödigt röriga.

Så slutsatsen av detta är att: Om behovet finns av att göra bägge dessa typer av arbete: provisionering och configuration management…..Bör dessa två verktyg absolut användas tillsammans, det ena bör inte utesluta det andra. De är bra på olika saker, de fungerar annorlunda och de bör komplettera varandra, inte slå ut varandra.
Men jag tycker endå vi bör anstänga oss för att inte kalla Terraform för ett config management verktyg och inte använda Ansible för provisioning, de är 2 olika verktyg, klockrena på sina egna typer av uppgifter.