# Design och migration av mjukvaruapplikationer till molnet

Jakob Nordman 40867
Kandidatavhandling i datateknik
Handledare: Ivan Porres
Fakulteten för naturvetenskap och teknik
Åbo Akademi
2020

# Innehåll

**1 Inledning** **1**

**2 Bakgrund** **2**

2.1 Tjänstemodeller för molnet . . . . . . . . . . . . . . . . . . . . . . 3

2.2 Monolitiska applikationer . . . . . . . . . . . . . . . . . . . . . . . 5

2.3 Mikrotjänster . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.4 DevOps . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

**3 Design av molncentrerade applikationer** **8**

3.1 Designmönster . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

3.1.1 API-Gateway pattern . . . . . . . . . . . . . . . . . . . . . 10

3.1.2 Service Registry pattern . . . . . . . . . . . . . . . . . . . 12

**4 Migration till molncentrerad arkitektur** **14**

4.1 Migrationsstrategier . . . . . . . . . . . . . . . . . . . . . . . . . . 14

4.2 Migrationsmönster . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

4.3 Fallstudie . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

**5 Diskussion** **20**

**Referenser** **21**

# 1 Inledning

Molntjänster har på den senaste tiden blivit allt mera kostnadseffektiva och lätta att använda. Detta för med sig att man inom mjukvaruutvecklingen har börjat utnyttja dem i allt högre grad, och nya designmönster har uppkommit för att effektivt kunna bygga molncentrerade (eng. cloud-native) system. Genom att utnyttja molnbaserade tjänster kan man bygga system som är lättare att underhålla och distribuera, samt i framtiden lätta att skala upp.

Applikationer byggda enligt mera traditionella metoder och designmönster saknar den flexibilitet som är kännetecknande för system specifikt byggda för att köras i molntjänster. Många företag strävar därför efter att överföra sina existerande system och applikationer till en molnbaserad kontext. Detta har inte visat sig vara någon lätt uppgift och som resultat har även flera nya metoder och mönster för överflyttning uppkommit.

Syftet med denna avhandling är att undersöka vilka designmönster som idag används för molncentrerade system och hur man kan flytta över en existerande applikation till en molnbaserad kontext. Inledningsvis kommer centrala begrepp att introduceras och definieras. Sedan behandlas existerande mönster för att designa molnbaserade system och slutligen diskuteras metoder för överföring av traditionella applikationer till molncentrerad arkitektur.

# 2 Bakgrund

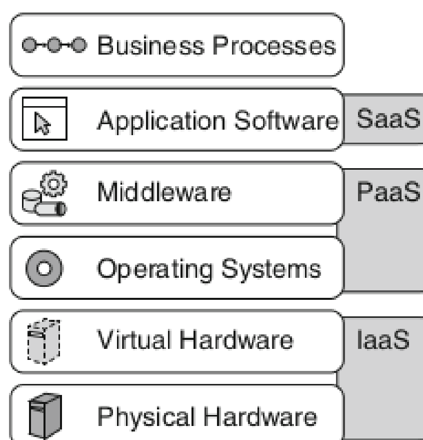
Den första allmänna molntjänsten, Simple Storage Service eller S3, publicerades år 2006 av AWS (Amazon Warehouse Services) [1]. Detta innebar uppkomsten av hela det fenomenet som numera är känt som molnet [2]. AWS räknas ännu idag till de världsledande leverantörerna av molntjänster, tillsammans med Microsoft Azure och Google Cloud.

För att förstå begreppet moln och molntjänster behövs en kännedom om typiska it-resurser som företag kan ha för att stöda sin dagliga verksamhet. Företaget kan till exempel behöva servrar för datalagring, system för databehandling eller kompletta mjukvaruapplikationer. En leverantör av molntjänster erbjuder typiskt dessa resurser som tjänster (eng. *as a Service*) [3]. Fördelen med detta är att företaget själv inte behöver anskaffa nödvändig infrastruktur för att stöda sina egna it-behov, utan kan köpa dem som en tjänst som leverantören tillhandahåller.

Termen ”cloud-native”, för vilken här används den svenska översättningen molncentrerad, är lite svårare att definiera. Kratzke och Quint [2] konstaterar i sin kartläggande studie, ”Understanding Cloud-native Applications after 10 Years of Cloud Computing - A Systematic Mapping Study”, att det inte existerar någon allmänt accepterad definition av begreppet och introducerar i sin artikel ett förslag till en definition. En molncentrerad applikation (eng. cloud-native application) är, enligt Kratzke och Quint, en applikation som består av flera olika moduler (se kap 2.3 Mikrotjänster) som är löst kopplade till varandra och kan skalas upp individuellt. Dessutom ska de vara designade enligt principer fokuserade kring molnkontext och vara elastiska, det vill säga kunna reagera automatiskt på ändringar i arbetsbörda [2].

## 2.1 Tjänstemodeller för molnet

Som tidigare nämnt är grundidén med molntjänster att kunna hyra it-resurser över nätet utan att själv behöva införskaffa nödvändig infrastruktur. De mest allmänna tjänstemodellerna som molnleverantörer erbjuder är Infrastructure as a Service (Iaas), Platform as a service (Paas) och Software as a Service (SaaS). I Figur 1 framkommer hur dessa tjänstemodeller motsvarar de olika lagren i en applikationsstack.




Figur 1: Applikationsstacken och motsvarande molntjänstmodeller - kopierad från [3]

IaaS innebär att tjänsteleverantören hyr ut it-infrastruktur, till exempel servrar eller lagringskapacitet, som en tjänst över nätet. Ur Figur 1 framkommer att kunden här själv ansvarig för att installera operativsystem och diverse mellanprogram för till exempel databashantering. Den stora fördelen med IaaS är snabb elasticitet, nya servrar och resurser kan allokeras och deallokeras på bara minuter [3]. Detta medför möjlighet till snabb respons vid en plötslig ändring i arbetsbörda. Nackdelen med att hyra infrastruktur som en

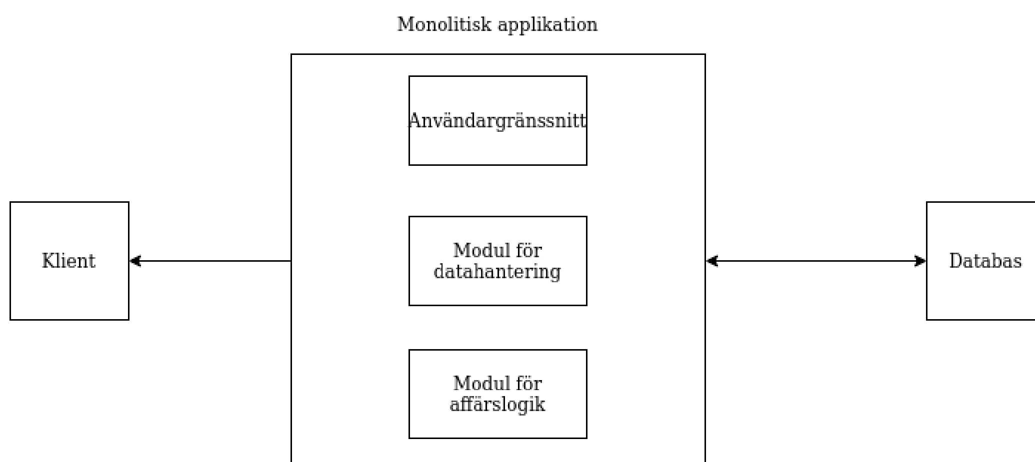
tjänst är att kunden själv har ansvar för att installera operativsystem och diverse mellanprogram, vilket kan leda till många överflödiga installationer om till exempel liknande miljöer krävs för körandet av flera applikationer [3].

Med PaaS modellen menas att tjänsteleverantören erbjuder en plattform över nätet, med färdigt installerade och konfigurerade operativsystem och mellanprogram (se Figur 1) enligt kundens specifikationer [3]. Även underhållet av operativsystemet och mellanprogrammen sköts av tjänsteleverantören. Kunden får alltså en färdigt konfigurerad miljö att utveckla eller köra sina applikationer i. PaaS löser dessutom delvis problemet med överflödiga installationer som existerar med IaaS modellen.

Att utveckla egna mjukvaruapplikationer är inte relevant eller ens möjligt för alla företag och att köpa en mjukvaruapplikation som företaget själv måste driftsätta antingen på en egen server eller i någon molntjänst medför ansvar för driftsättningen och underhållet av applikationen. Med SaaS modellen erbjuder leverantören en komplett applikation som kunden kan nå via nätet [3]. Allt ansvar för infrastruktur, lagring och underhåll faller på leverantören. Ett typiskt exempel på en mjukvaruapplikation som fungerar som en tjänst över nätet är Microsofts Office 365. Fördelen med SaaS är att applikationer snabbt kan tas i bruk utan installationer eller behov av egen it-infrastruktur. Ett problem med SaaS är att kunden inte själv har möjlighet att utvidga applikationens funktionalitet, men detta har tjänsteleverantörer delvis löst genom att erbjuda PaaS miljöer var applikationen lätt kan integreras och utvidgas [3].

## 2.2 Monolitiska applikationer

Det traditionella sättet att utveckla en applikation och mjukvara i allmänhet är att bygga dem som så kallade monoliter. Termen monolit betyder i mjukvarusammanhang att applikationen är byggd som en enda enhet som kompileras tillsammans [4]. Figur 2 illustrerar simplifierat hur en monolitisk applikation kan vara uppbyggd, med moduler för flera olika ansvarsområden samlade till en och samma enhet.




Figur 2: Exempelarkitektur hos en monolitisk applikation - baserad på https://microservices.io/patterns/monolithic.html

Det finns både fördelar och nackdelar med att bygga monolitiska applikationer. Speciellt när ett nytt system ska byggas kan det vara fördelaktigt att bygga en monolitisk applikation, eftersom de är lättare och snabbare att initialt utveckla och driftsätta. Nackdelen med monoliter uppstår när systemet har hunnit växa, både till storlek och komplexitet. Introduktionen av nya utvecklare till systemet blir allt svårare, vilket resulterar i att utvecklingsarbetet saktas ner. Dessutom blir ändringar till applikationen allt svårare att göra, eftersom hela applikationen måste driftsättas igen vid en ändring, och

inte bara den delen som har uppdaterats. Applikationen är också svårare att skala vid ökad arbetsbörda, jämfört med molncentrerade applikationer [5].

## 2.3 Mikrotjänster

Ett sätt att undkomma nackdelarna som en monolitisk arkitektur medför är att dela upp applikationens olika moduler i så kallade mikrotjänster. En mikrotjänst är alltså en modul av en applikation, som istället för att köras i samma process som resten av applikationen körs i en egen process. Mikroprocesser kommunicerar typiskt med varandra med hjälp av begäranden (eng. requests) över internet [6].

Den största fördelen med att använda mikrotjänster i en mjukvaruapplikation är att de kan köras och uppdateras oberoende av varandra[6]. Vid en uppdatering av en modul eller tjänst behöver alltså endast den tjänsten driftsättas igen, medan en ändring i en modul av en monolitisk applikation kräver att hela applikationen driftsätts på nytt.

Applikationer som utnyttjar mikrotjänster är också lättare att skala vid ökad arbetsbörda än monolitiska applikationer. För att skala en monolit körs oftast en kopia av applikationen på en annan server. Mikrotjänster kan i sin tur oberoende av varandra kopieras och distribueras över olika servrar[6].

Det existerar även nackdelar med en applikation som utnyttjar sig av mikrotjänster. Eftersom tjänsterna körs i olika processer så är kommunikation mellan dem mera kostsam än om de kördes i samma process. Detta medför även att gränssnitten mellan tjänsterna bör planeras mera omsorgsfullt för att undvika att flaskhalsar uppstår [6].

## 2.4 DevOps

DevOps, sammansatt av engelskans Developer och Operations, är en rörelse inom systemutvecklingen som eftersträvar att föra utvecklare och it-drift närmare varandra [7]. Grundidén är att minska tiden mellan att en ändring görs till systemet och att den ändringen blir levererad till produktionsmiljön, alltså den miljön var applikationen tjänar sitt syfte. Inom DevOps eftersträvas också att kvaliteten på koden och leveransmetoderna hålls höga [8].

Två av dom mest centrala begreppen inom DevOps är kontinuerlig leverans och kontinuerlig integrerig (eng. continuous delivery och continuous integration). Kontinuerlig integrering innebär att ändringar till ett systems kodbas snabbt skall kunna verifieras som fungerande och integreras. Detta implementeras ofta med en automatiserad pipa (eng. pipeline) som kör automatiserade tester och kompilerar kodbasen till ett nytt paket som kan driftsättas varje gång ny kod laddas upp [8].

Kontinuerlig leverans uppstår ofta som följd av kontinuerlig integrering [8]. Det syftar på en förenkling och automatisering av leveransen av nya kodpaket och kan implementeras antingen som en fortsättning på integreringspipan eller som en helt egen pipa. Kontinuerlig leverans är speciellt viktig för applikationer byggda för molnmiljöer består av små, oberoende tjänster som driftsätts i separata miljöer. Utan kontinuerlig leverans blir processen att leverera hela systemet tidskrävande och svår att övervaka [8].

# 3 Design av molncentrerade applikationer

Begrepper molncentrerat har, som tidigare konstaterat, ingen entydig definition. Flera studier har dock lyckats urskilja gemensamma principer och mönster gällande design och arkitektur hos molncentrerade applikationer [2][9]. Typiskt för de flesta molncentrerade applikationer är att de utnyttjar mikrotjänster (se kap. 2.3) och DevOps (se kap. 2.4).

Fehling et al. [3] föreslår i sin bok ”Cloud Computing Patterns - Fundamentals to Design, Build and Manage Cloud Applications”en princip kallad IDEAL för att designa molncentrerade applikationer. IDEAL står för: **I**solated state, **D**istribution, **E**lasticity, **A**utomated management och **L**oose coupling. I följande stycken kommer dessa begrepp att definieras och kort beskrivas.

Isolated state eller isolerat tillstånd syftar på en eftersträvan att så långt som möjligt göra applikationen tillståndslös(eng stateless).Begreppet tillstånd i kontexten av mjukvaruapplikationer syftar på om systemet påverkas av tidigare händelser eller inte. Att lagra tillstånd i de olika tjänsterna i ett molncentrerat mjukvarusystem påverkar negativt på möjligheten att skala upp systemet vid ökad arbetsbörda, och bör därför undvikas så långt som möjligt [3].

Distribution eller fördelning är en av de viktigaste principerna för molncentrerade system. En applikation designad för att köras i molnmiljö är till sin natur bestående av flera olika mikrotjänster som driftsätts och körs oberoende av varandra.

Elasticity eller elasticitet syftar på molncentrerade applikationers förmåga att reagera på ökad arbetsbörda. Fehler et al. påpekar att det finns två sätt

en applikation kan skalas på, antingen vertikalt eller horisontellt (eng. scale horizontally och scale vertically) [3]. Med vertikal skalning menas att applikationens prestanda höjs genom att allokera mera resurser till dess exekveringsmiljö i form av beräkningskraft eller lagring. Horisontell skalning, som är typisk för molncentrerade applikationer, innebär att antaler instanser av en applikation eller en tjänst ökas vid ökad arbetsbörda, utan att allokera extra resurser till de enskilda tjänsterna. Elasticitet syftar specifikt på systemets kapacitet att snabbt och dynamiskt kunna skalas horisontellt [3].

Automated management eller automatiserad hantering hänger starkt ihop med molncentrerade applikationers elasticitet. Eftersom applikationen ofta behöver skapa nya instanser på nya servrar till följd av en ändring i arbetsbördan krävs ett automatiserat system för att övervaka systemet och tilldela det de nya resurserna som det behöver för att svara på ändringen [3]. Automatiserad hantering är också viktigt för att se till att systemet är resistant mot driftavbrott.

Loose coupling eller lös sammankoppling återkopplar till molncentrerade systems distribuerade natur. Eftersom ett system består av flera olika mikrotjänster som alla kan driftsättas och ändras oberoende av varandra är det viktigt att tjänsterna beror så lite som möjligt på varandra.Detta underlättar även skalningen av systemet och minskar effekten av ett eventuellt driftavbrott [3].

I den tidigare nämnda kartläggande studien ”Understanding Cloud-native Applications after 10 Years of Cloud Computing - A Systematic Mapping Study” [2], har Kratzke och Quint utöver dessa ovannämnda principer identifierat två trender inom molncentrerad mjukvaruarkitektur. Den första är en standardisering av driftsättningsenheter för systemets tjänster. Driftsättningsenheter förpackar en tjänst tillsammans med ett komplett filsystem och alla biblio-

tek och systemverktyg som tjänsten behöver för att köra. Detta försäkrar att tjänstens beteende hålls konsekvent, oavsett vilken miljö den driftsätts i. I praktiken används ofta containrar för att skapa standardiserade driftsättningsenheter, med hjälp av verktyg så som Docker. Den andra trenden är att använda sig av ett enkelt och skalbart sätt för de olika driftsättningsenheterna att kommunicera. Typiskt använder sig enheterna av programmeringsgränssnitt, även kallade API (från engelskans application programming interface), baserade på till exempel en REST standard för att kommunicera med varandra via internet.

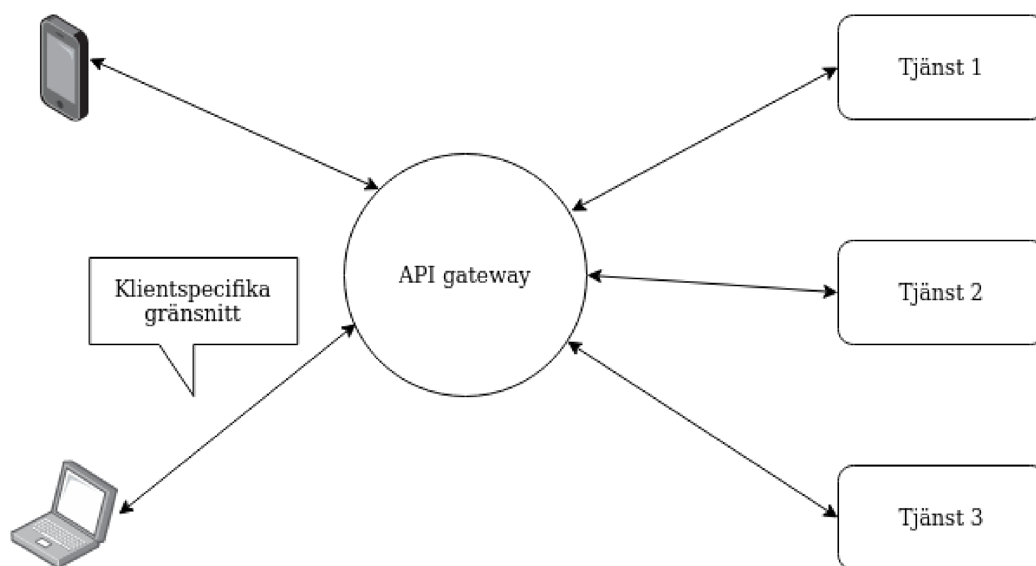
## 3.1 Designmönster

Taibi et al. har i sina två studier "Architectural Patterns for Microservices: a Systematic Mapping Study"[9] och "Continuous Architecting with Microservices and DevOps: A Systematic Mapping Study"[10] identifierat två huvudsakliga designmönster som är allmänt förekommande inom applikationer med en mikrotjänstbaserad arkitektur. Till näst kommer dessa designmönster att redogöras för och dess fördelar respektive nackdelar presenteras.

### 3.1.1 API-Gateway pattern

Som tidigare diskuterats kommunicerar de olika tjänsterna inom en molncentrerad applikation typiskt genom sina programmeringsgränssnitt över ett nätverk. För att kunna skapa en applikation krävs en mekanism för att binda samman dessa tjänster, för vilket en så kallad API gateway kan användas. En API Gateway, eller API-portmekanism, fungerar som en inkörsport till applikationen och ruttar en användares förfrågningar till de relevanta tjänsterna, sammanställer resultaten och presenterar dem för användaren [10]. Ingen kommunikation sker således rakt mellan användare och de diverse tjänsterna, utan allt går via API porten. I Figur 3 illustreras principen för en arkitektur

som utnyttjar sig av API Gateway mönstret.




Figur 3: Arkitekturmönster för API Gateway [10]

Fördelarna med att använda sig av API Gateway mönstret är systemet blir lätt att utvidga. En ändring till en tjänsts API kräver endast en ändring till portmekanismen. Eftersom klienterna har egna gränssnitt till portmekanismen påverkar en ändring i gränssnittet mellan portmekanismen och de bakomliggande tjänsterna inte direkt användaren. Detta medför även god bakåtkompabilitet[9].

Flera nackdelar har även identifierats i användningen av detta mönster. Eftersom portmekanismen fungerar som ända inkörsport till systemet kan den ge upphov till en flaskhals, ifall dess design är bristfällig. Försiktighet bör även vidtas med de olika gränssnitten mellan portmekanismen och de olika tjänsterna, speciellt i fall var flera tjänster använder sig av samma gränssnitt hos portmekanismen. När antalet tjänster ökar inom systemet kan trafiken

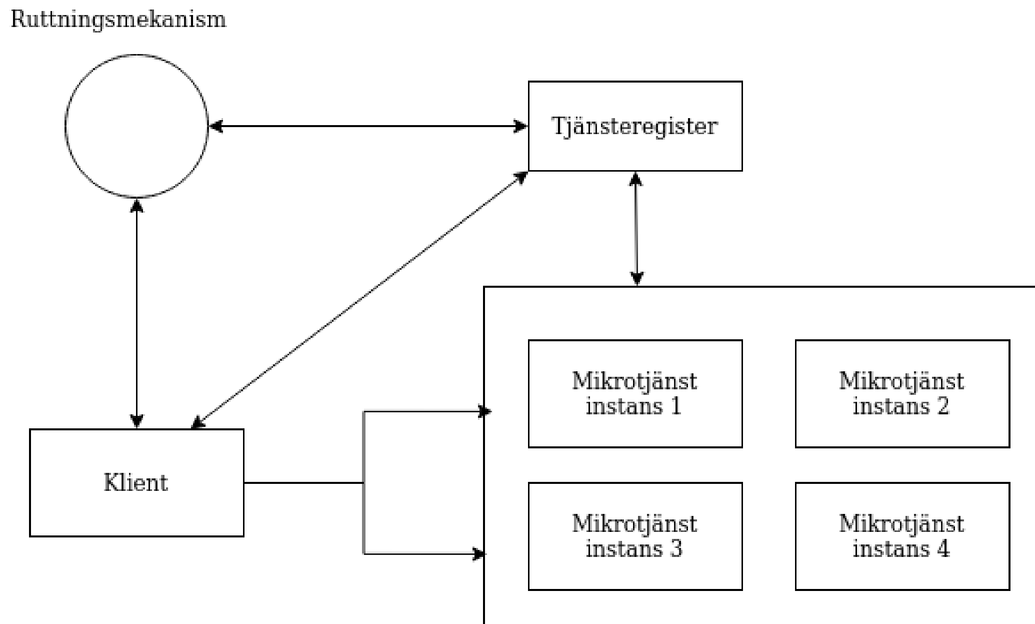
över portmekanismen bli för stor vilket kommer att kräva en effektivare och mera skalbar ruttningsmetod för kommunikationen inom systemet [9].

### 3.1.2 Service Registry pattern

I ett molncentrerat mjukvarusystem existerar typiskt flera olika instanser av samma mikrotjänst, driftsatta i olika driftsättningsenheter. Kommunikationen mellan dessa bör, enligt Taibi et al. [10], kunna definieras dynamiskt och en klient bör effektivt kunna få kontakt med en relevant instans av tjänsten. Därav behovet av ett så kallat tjänsteregister (eng. service registry), som känner till den dynamiska adressen för varje instans av en mikrotjänst. Service Registry mönstret kan delas upp i två delmönster för att upptäcka den relevanta instansen av en tjänst, upptäckning på klientsidan och upptäckning på serversidan. Figur 4 representerar båda dessa delmönster. I upptäckning på klientsidan skickar klienten en förfrågan direkt till tjänsteregistret, som väljer en relevant instans och vidarebefordrar förfrågningen. I delmönstret upptäckning på serversidan går klientens förfrågan först genom en ruttningsmekanism (se Figur 4), som därefter efterfrågar en relevant instans från tjänsteregistret och vidarebefodrar klientens förfrågan till den tjänsten [10].

Till skillnad från API Gateway mönstret kommunicerar klienten här direkt med systemets olika mikrotjänster via deras programmeringsgränssnitt [10]. Systemet förlitar sig på tjänsteregistret, som ruttar klientens förfrågningar vidare till den för klienten relevanta instansen av en mikrotjänst.

En fördel med att använda sig av tjänsteregistermönstret är ökad underhållbarhet och enklare kommunikation, speciellt i förhållande till API Gateway mönstret [9], i och med att klienten direkt kan kommunicera med de olika mikrotjänsterna utan något mellansteg (portmekanismen i API Gateway mönstret). Tjänsteregistermönstret medför också att mjukvaran blir lättare att förstå och därmed också lättare




Figur 4: Arkitekturmönster för Service Registry [10]

att utveckla [9].

Flera nackdelar har även identifierats i användningen av tjänsteregistermönstret. Eftersom tjänsterna kommunicerar direkt med klienten och även sinsemellan resulterar en ändring till en tjänsts programmeringsgränssnitt i att alla klienter och tjänster som kommunicerar med den också kräver ändringar. Tjänsteregistret riskerar dessutom, precis som portmekanismen i API Gateway mönstret, att bli en flaskhals [9]. Systemet blir som en följd av tjänsteregistermönstret även mer distribuerad, vilket i sin tur medför komplexitet till följd av ökad kommunikation mellan tjänsterna [9].

# 4 Migration till molncentrerad arkitektur

Som tidigare diskuterats har intresset för att migrera existerande applikationer till molnet ökat under de senaste åren. Företag eftersträvar med migrationen att kunna utnyttja de fördelar som molntjänster för med sig, så som ökad flexibilitet, bättre skalbarhet och en mer kostnadseffektiv prissättning [11]. Enligt Jamshidi et al:s litteraturstudie [12] finns de tre primära orsaker till varför organisationer väljer att överväga en migration av sitt mjukvarusystem. Den första är en reducering av operativa kostnader, den andra är applikationens skalbarhet och den tredje att effektivare kunna utnyttja it-resurser [12].

Att migrera en applikation till molnet har inte visat sig vara någon lätt uppgift och för en del mjukvaruapplikationer är det inte ens ett alternativ. Mjukvaruapplikationer så som säkerhetskritiska system och inbyggda system kan inte enkelt anpassas för att köras i molnmiljö [12]. Detta kapitel kommer främst att fokusera på migration av äldre system (eng. legacy systems) som med tiden blivit för stora eller komplexa för att effektivt kunna vidareutvecklas eller underhållas. Först kommer ett antal olika migrationsstrategier att presenteras och definieras, sedan behandlas olika mönster för att migrera en applikation till en molncentrerad och mikrotjänstbaserad arkitektur och slutligen kommer en fallstudie av en lyckad migration att presenteras.

## 4.1 Migrationsstrategier

Som tidigare diskuterats är en molnmigration ingen enkel uppgift och kan till och med orsaka avbrott i organisationens affärsverksamhet. Därför är det viktigt med en väl utarbetad strategi som följs vid både beslutsfattningen och utförandet av migrationen [13]. Åtskilliga strategiska ramverk existerar för molnmigrationer, av vilka den mest kända är "de fem R:en"(eng. the

5 R's) som ursprungligen sammanställdes av konsultföretaget Gartner [13]. Dessa 5 R har sedan dess omarbetats till 6 stycken [14] och står för re-host, re-platform, refactor, repurchase, retain och retire.

Re-host innebär att applikationen flyttas över till en ny värd (eng. host) så som den är utan att göra några ändringar [13], i allmänhet genom att utnyttja IaaS tjänstemodellen (se kap. 2.1). Strategin brukar ibland benämnas "lyfta och flytta"(eng. lift and shift) och har trots sin enkelhet visat sig kunna reducera en organisations operativa kostnader och underlätta framtida optimering av applikationen för molnet [14]. Re-platform innebär att små optimeringar görs till applikationen i samband med att den flyttas över till en ny värd, utan att ändra applikationens struktur eller arkitektur [14]. Oftast utnyttjas PaaS tjänstemodellen vid denna strategi [13]. Refactor innebär att hela applikationen refaktoriseras med hjälp av molncentrerade metoder för att optimalt kunna utnyttja de fördelar som molnmiljön erbjuder [13]. Refaktorisering är den strategi som främst behandlas senare i detta kapitel. Repurchase betyder att applikationene återköps, ofta i form av en SaaS applikation. Retain innebär att applikationen bevaras som den är och migrationen omprövas, medan retire innebär att applikationen tas ur drift [13]. Trots att de tre sista strategierna inte fulländar någon migration av existerande applikationer till molnet är de ändå rimliga strategier för en organisation som efterstävar att flytta sina it-resurser till molnet och bör övervägas i samband med beslut om migration.

Zhao och Zhou [15] har i sin studie identifierat tre strategier som baserar sig på vilka tjänstemodeller som utnyttjas: migration till IaaS, migration till PaaS och migration till SaaS. Migration till IaaS är ekvivalent med föregående styckes re-host och innebär att systemet flyttas över så som det är. Migration till PaaS motsvarar re-platform och innebär att små ändringar görs till

systemet i enlighet med plattformen som det flyttas till. Zhao och Zhou delar ytterligare in migration till SaaS i tre delmönster: utbyte av SaaS, omformning med hjälp av SaaS och omformning till SaaS. Omformning med hjälp av SaaS innebär att några av systemets komponenter ersätts med mjukvarutjänster som molntjänsteleverantören erbjuder, medan omformning till SaaS innebär att hela systemet byggs om enligt SaaS modellen. Utbyte av SaaS motsvarar föregående styckes repurchase.

## 4.2 Migrationsmönster

För att kunna dra maximal nytta av en molnmiljös fördelar krävs det att den applikation som ska migreras refaktoriseras och att dess arkitektur ändras enligt molncentrerade principer. Detta betyder i praktiken att applikationens moduler delas upp i mikrotjänster och att metoder för kontinuerlig integration och leverans integreras (se kap. 2.4). Balalaie et al. har i sin studie "Microservices migration patterns"[11] genom empiriska studier av industriella migrationsprojekt identifierat 15 olika mönster för att migrera en applikation eller ett system till molnet. Mönstren är atomära och deras ordningsföljd har ingen betydelse. Till näst kommer dessa mönster att beskrivas och definieras för att sedan återkopplas till i följande kapitels fallstudie.

De första två mönstren är att möjliggöra kontinuerlig integration och att klargöra applikationens nuvarande arkitektur. Kontinuerlig integration är det första steget mot kontinuerlig leverans [11], något som är av vital betydelse i ett molncentrerat system som till sin natur är distribuerat. Att klargöra den nuvarande arkitekturen av applikationen ligger till grund för den fortsatta migrationen. Gruppen som skall planera och utföra migrationen bör känna till applikationens nuvarande struktur till exempel för att korrekt kunna separera den i mikrotjänster.

Med mönster 3, 4 och 5 beskriver Balalaie et al. uppdelningen av applikationen i mikrotjänster. Mönster 3 och 4 beskriver två olika metoder för uppdelning av ett system i mikrotjänster, antingen baserat på deras kontext eller på ägande av data. Uppdelning baserat på kontext innebär att man delar upp systemets moduler baserat på dess kontext i applikationen, till exempel en varukorgsfunktionalitet i en nätbutik. Uppdelning baserat på ägande av data innebär att man delar upp applikationens data till sammanhängande enheter som kan ha en unik ägare. Sedan separeras denna dataenhet tillsammans med sin ägare till en mikrotjänst. Det femte mönstret förespråkar en ändring av beroende mellan modulerna från kodnivå till tjänstenivå. Målet med mönster 5 är tjänster som så långt som möjligt delar så lite kod som möjligt [11].

Mönster 6-12 berör införandet av stödkomponenter som en molncentrerad applikation behöver. Mönster 6 förespråkar införandet av ett tjänsteregister (se kap. 3.1.2), för att hantera systemets olika tjänsters dynamiska adresser. Mönster 7 inför en tjänsteregisterklient till varje tjänst. Med hjälp av den klienten kan en tjänst meddela sin adress till tjänsteregistret vid driftsättning [11]. Mönster 8 introducerar en intern belastningsreglerare (eng. load balancer) till varje tjänst, med hjälp av vilken en lista på tillgängliga instanser av en begärd mikrotjänst tillhandafås, vilken belastningsregleraren sedan kan använda för att välja en lämplig instans till exempel baserat på instansens svarstid [11]. Mönster 9 introducerar en extern belastningsreglerare till systemet, som centrerat ser till att trafiken i systemet fördelas jämnt. Mönster 10 introducerar en kretsbrytare till systemet (eng. circuit breaker), med vilken en utklockning kan undvikas vid försök att nå en otillgänglig instans av en tjänst. Mönster 11 introducerar en konfigurationstjänst, med hjälp ev vilken en tjänsts konfiguration kan ändras under exekvering utan behov av

att omdriftsätta tjänsten. Mönster 12 introducerar en gränsserver till systemet, som fungerar som ett mellanskikt mellan klienten resten av systemet [11]. Introduktionen av gränsservern resulterar i en arkitektur som kan liknas vid en kombination av de två designmönstren från kapitel 3.1, med både ett tjänsteregister och en portmekanism.

Mönster 13, 14 och 15 berör driftsättningen av systemet. Mönster 13 förespråkar att systemets tjänster omsluts i driftsättnigsenheter i from av containrar och mönster 14 förespråkar att dessa enheter driftsätts i from av ett kluster som kan orkestreras till exempel med hjälp av verktyg så som Kubernetes. Mönster 15 introducerar slutligen en övervakningsmekanism till varje tjänst, för att kunna följa med tjänstens förbrukning av resurser. Detta medför att utvecklarna av systemet lättare kan upptäcka flaskhalsar eller andra avvikelser i systemet [11].

## 4.3 Fallstudie

Backtory är ett kommersiellt grundsystem (eng. backend) för mobilutvecklare, som erbjuds som en tjänst [11]. Applikationen började som ett hanteringssystem för relationsdatabaser som till följd av ökade krav på funktionalitet migrerades till en mikrotjänstbaserad arkitektur [8], för att lättare kunna hantera framtida krav på funktionalitet. Migrationen utfördes av Balalaie et al. och har beskrivits utförligt i studierna [11][8].

Balalaie et al. poängterar att vid en migration bör sådana mönster som inte har några grundförutsättningar väljas först. I fallet Backtory valdes att först introducera kontinuerlig integration och att dokumentera applikationens dåvarande arkitektur (mönster 1 och 2). I migrationens följande steg bör användarna av applikationen beaktas och ändringar som görs till applikatio-

nen bör ej påverka dem. Balalaie et al valde således att bryta ut en av applikationens moduler som inte var synlig för användarna [11] med hjälp av mönster 5. Ytterligare introducerades en konfigurationstjänst (mönster 11) samt en introduktion av driftsättningsenheter för att underlätta leveransen (mönster 13). Efter denna ändring kunde inte längre ändringar som inte påverkar användarna göras, varpå man valde att introducera en gränsserver (mönster 12) som ett mellanskikt mellan användarna och resten av applikationen. Utan någon mekanism för att dynamiskt kunna upptäcka tjänsternas adresser krävdes i början en statisk konfiguration av gränsservern [11]. Följande steg var att introducera ett tjänsteregister samt tjänsteregisterklienter till systemets tjänster (mönster 6 och 7) för att möjliggöra kommunikation mellan tjänsterna och undvika statiska konfigurationer. Därefter introducerades belastningsreglering till alla tjänster för att jämnt kunna fördela systemets trafik. Som nästa steg introducerades kretsbrytare till applikationen för att uppnå bättre resistens mot driftavbrott. I detta skede var redan en stor del av allt grundarbete gjort och Balalaie et al. fortsatte att dela upp resten av applikationen i mikrotjänster, baserat på ägande av data (mönster 4). Slutligen utnyttjades mönster 14 och 15 för att förbättra systemets skalbarhet och göra det mer observerbart [11].

Efter migrationen har Backtory framgångsrikt utvidgats med ny funktionalitet, mycket tack vare ett noggrant planeringsarbete och välgjora designval [11]. Arbetsgruppen som vidareutvecklar tjänsten har även genomgått strukturella ändringar och introduktionen av nya utvecklare till projektet har underlättats [8]. Balalaie et al. noterar att en lyckad migration kräver skickliga utvecklare och väldesignade gränssnitt mellan tjänsterna. Slutligen poängteras, trots en i detta fall lyckad migration, att en mikrotjänstbaserad arkitektur medför en ökad komplexitet till systemet och kräver ökad ansträngning av systemets utvecklare [8]. ’

# 5 Diskussion

TODO

# Referenser

[1] (2006). Release: Amazon S3 on 2006-03-13, URL: https://aws.amazon.com/releasenotes/release-amazon-s3-on-2006-03-13/ (hämtad 2020-02-11).

[2] N. Kratzke och P.-C. Quint, ”Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study”, *Journal of Systems and Software*, årg. 126, s. 1–16, 2017. DOI: 10.1016/j.jss.2017.01.001.

[3] C. Fehling, F. Leymann, R. Retter, W. Schupeck och P. Arbitter, *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014. DOI: 10.1007/978-3-7091-1568-8.

[4] (2014). What is a Monolith?, URL: http://www.codingthearchitecture.com/2014/11/19/what_is_a_monolith.html (hämtad 2020-02-20).

[5] C. Richardson. (2019). Pattern: Monolithic Architecture, URL: https://microservices.io/patterns/monolithic.html (hämtad 2020-02-20).

[6] J. Lewis och M. Fowler. (2014). Microservices, URL: https://www.martinfowler.com/articles/microservices.html (hämtad 2020-02-20).

[7] C. Sweden och IDG. (2018). DevOps, URL: https://it-ord.idg.se/ord/devops/ (hämtad 2020-02-20).

[8] P. J. Armin Balalaie Abbas Heydarnoori, ”Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture”, *IEEE Software*, årg. 33, nr 3, s. 42–52, 2020. DOI: 10.1109/MS.2016.64.

[9] D. Taibi, V. Lenarduzzi och C. Pahl, ”Architectural Patterns for Microservices: A Systematic Mapping Study”, mars 2018. DOI: 10.5220/0006798302210232.

[10] ——, ”Continuous Architecting With Microservices and DevOps: a Systematic Mapping Study”, i. aug. 2019, ISBN: 978-3-030-29192-1. DOI: 10.1007/978-3-030-29193-8_7.

[11] A. Balalaie, A. Heydarnoori, P. Jamshidi, D. A. Tamburri och T. Lynn, ”Microservices migration patterns”, årg. 48, s. 2019–2042, 2018. DOI: https://doi.org/10.1002/spe.2608.

[12] P. Jamshidi, A. Ahmad och C. Pahl, ”Cloud Migration Research: A Systematic Review”, *IEEE Transactions on Cloud Computing*, årg. 1, nr 2, s. 142–157, 2013.

[13] N. Ahmad, Q. N. Naveed och N. Hoda, ”Strategy and procedures for Migration to the Cloud Computing”, i *2018 IEEE 5th International Conference on Engineering Technologies and Applied Sciences (ICETAS)*, 2018, s. 1–5. DOI: 10.1109/ICETAS.2018.8629101.

[14] S. Orban. (2016). 6 Strategies for Migrating Applications to the Cloud, URL: https://aws.amazon.com/blogs/enterprise-strategy/6-strategies-for-migrating-applications-to-the-cloud/ (hämtad 2020-03-28).

[15] J.-F. Zhao och J.-T. Zhou, ”Strategies and Methods for Cloud Migration”, *International Journal of Automation and Computing*, årg. 11, s. 143–152, 2014. DOI: https://doi.org/10.1007/s11633-014-0776-7.