Utveckling och design av mikrotjänstbaserade datorprogram

Maxemilian Grönblom 42047-27-2017
Kandidatavhandling i datavetenskap
Handledare: Ivan Porres
Fakulteten för naturvetenskaper och teknik
Åbo Akademi
2021

Innehållsförteckning

1 Introduktion

Nuförtiden kan datorprogram behöva tjäna tiotusentals användare på samma gång. Denna enorma mängd användare ställer höga krav på datorprogrammens tillgänglighet och skalbarhet. Både skalbarheten och tillgängligheten av moderna datorprogram växer dock i samband med att molnbaserade datortjänster blir allt vanligare [1]. Medan datorprogrammens användarantal växer så ökar även pressen på traditionella designarkitekturer att adaptera sig till molnbaserade datortjänster och deras designmönster.

Traditionellt är datorprogram utvecklade och designade som monolitiska datorprogram, vilket innebär att de är designade så att programmets alla funktioner och tjänster är samlade i samma process [1]. Datorprogram som följer monolitiska designprinciper kan ha svårigheter att skala när användarantalet växer. Denna svårighet beror på att hela datorprogrammet måste skalas, inte bara de delar av programmet som är under belastning [2].

En nyare arkitektur som har börjat adopterats allt mer i utvecklingen av moderna molncentrerade datorprogram är mikrotjänstarkitektur. Mikrotjänstarkitekturen är designad för att utnyttja fördelarna med molnbaserade datortjänster. Mikrotjänstsarkitekturen består av en samling designprinciper som är till för att underlätta utvecklingsflexibiliteten, produktiviteten, skalbarheten samt göra det lättare att distribuera datorprogram [3]. Mikrotjänstarkitekturen förespråkar att dela upp datorprogram i mindre bitar, så kallade mikrotjänster.

Syftet med denna avhandling är att undersöka och redogöra nuvarande designmönster för mikrotjänstbaserade datorprogram samt belysa diverse fallgropar som kan uppkomma under utvecklingen av mikrotjänster. Avhandlingen kommer även att behandla olika refaktoriseringar som kan göras för att undvika fallgroparna.

2 Bakgrund

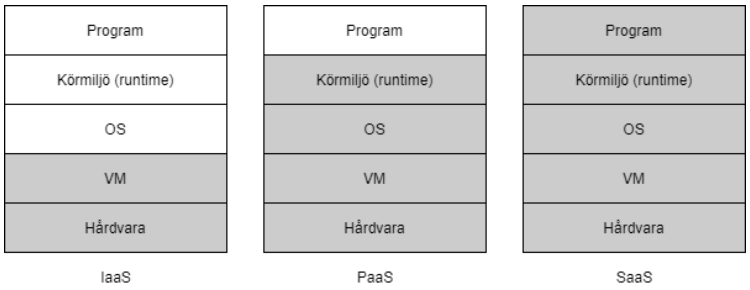
Molnbaserade datortjänster (*eng.* Cloud Computing) kan beskrivas som ett sätt för individer eller företag att få tillgång till nätverksbaserade it-resurser via en molnleverantör. Det finns några drivfaktorer för företag och individer att börja använda sig av molnbaserade datortjänster. En stor drivfaktor till att företag har börjat använda sig av molnbaserade datortjänster är att de inte själv behöver uppehålla en it-infrastruktur, utan det ansvaret faller på molnleverantörerna. En annan drivfaktor för molnbaserade datortjänsterna är att man endast betalar för de resurser som används, denna betalningsmodell kallas *pay-per-use*. [4], [5]

2.1 Modeller för molnbaserade datortjänster

Molnbaserade datortjänster delas traditionellt upp i tre olika modeller. Dessa modeller är infrastruktur som en nättjänst (IaaS, eng. *Infrastructure as a service*), plattform som en nättjänst (PaaS, eng. *Platform as a service*) och datorprogram som nättjänst (SaaS, eng. *Software as a service*). [4], [6]

IaaS-modellen innebära att man kan hyra fysisk eller virtuell infrastruktur på begäran. Till infrastrukturen som IaaS erbjuder hör bland annat serverdatorer, lagring samt nätverksinfrastruktur. Fördelarna med IaaS-modellen är att man snabbt och flexibelt kan skala upp resurser [4]. Nackdelarna illustreras i figur 2.1, som visar att IaaS-modellen inte omfattar konfiguration av infrastrukturen. Det innebär att man själv är tvungen att konfigurera och hantera operativsystemet och datorprogrammen.

PaaS-modellen erbjuder utöver IaaS-modellen färdigt konfigurerade resurser, på vilka man kan distribuera datorprogram. Fördelarna med PaaS-modellen är alltså att kunden inte måste hantera operativsystemet eller programmets körmiljö. PaaS-modellen kan också erbjuda automatisk skalning vid ökad arbetsbörda. Nackdelarna med PaaS-modellen är att datorprogram ofta måste



Figur 2.1: Molnbaserade datortjänstmodeller.
De gråmarkerade delarna hanteras av molnleverantören. Anpassad från [7]

utvecklas på ett visst sätt för att kunna utnyttja PaaS-modellens alla fördelar. [4]

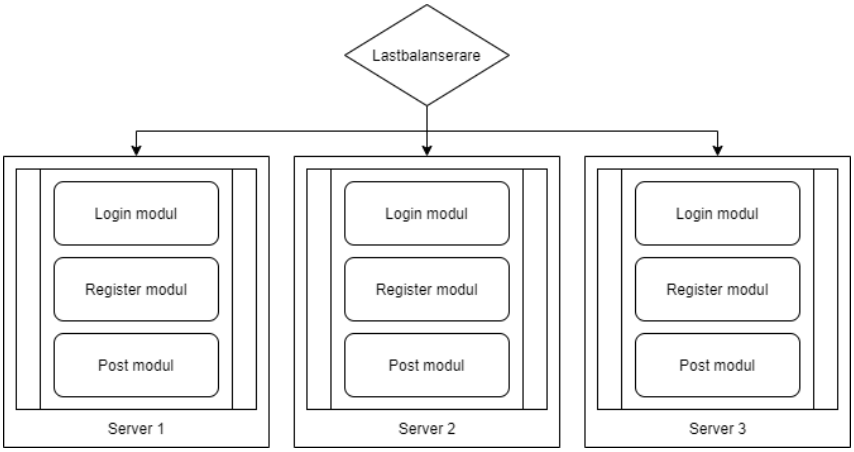
Figur 2.1 illustrerar även SaaS-modellen, vilken är en modell där alla delar hanteras av en leverantör. SaaS-modellen innebär alltså att leverantören har ett färdigutvecklat datorprogram som de hyr ut. Datorprogram som ofta erbjuds som SaaS är kommunikationsprogram eller kontorsviter, exempelvis Slack respektive Google Workspace. Den största fördelen med SaaS-modellen är att företag inte måste utveckla och underhålla anpassade datorprogram.

2.2 Mjukvaruarkitektur

Vid utveckling och design av datorprogram så följer man oftast någon sorts arkitektur. Kruchten et al. [8] definierar mjukvaruarkitektur som en struktur för systemkomponenter och delsystem samverkar för att skapa system. Men även de egenskaperna av ett system som man bäst kan designa och analysera på en systemnivå, hör till mjukvaruarkitekturen. Det kan alltså ses som en plan (eng. *blueprint*) för hur man skall utveckla och designa ett system [9]. Det finns ett flertal olika mjukvaruarkitekturer, de arkitekturer som är relevanta för denna avhandling är monolitisk arkitektur och mikrotjänstarkitektur. I de följande stycken kommer dessa definieras och deras fördelar och nackdelar kommer redovisas.

2.2.1 Monolitisk arkitektur

Datorprogram som följer en monolitisk arkitektur brukar kallas monoliter och definieras av Villamizar et al. [1, s. 234] på detta sätt: "... an application with a single codebase/repository that expose tens or hundreds of different services to external systems or consumers using different interfaces ...". De flesta traditionella program är designade som monoliter och den monolitiska arkitekturen har både sina fördelar och nackdelar. Fördelarna med att använda en monolitisk arkitektur är att det är enkelt att börja utveckla, testa och distribuera datorprogram. Mängden nackdelar ökar i samband med att programmets funktionalitet och storlek växer. Till dessa nackdelar hör svårigheter att förstå hela programmet, problem att skala programmet samt att distributionen av programmet blir svårare. [10], [11]



Figur 2.2: Uppskalning av monolitiska datorprogram. Anpassad från [2]

Uppskalningen av en monolit illustreras i figur 2.2. Skalande genomförs i praktiken genom att starta en ny instans av monoliten per serverdator och sedan låta en lastbalanserare rutta användare till en instans. Några av de tidigare nämnda nackdelarna belyses även av Figur 2.2, eftersom en ändring av en del av programmet kräver att alla instanser startas om med den nya ändringen. Detta kan leda till långa distributionstider samt risk för misslyckande av distribution. [10], [11]

Problemen som uppkommer vid större skala för monoliter är inte heller en-

bart relaterade till utvecklingen och distributionen. På grund av den inflexibla skalbarheten kan det uppkomma större kostnader med att köra monoliter på molnbaserade datatjänster [1]. Eftersom det uppkommer en mängd svårigheter med att köra en monolit på större skala så har behovet av en skalbar och molncentrerad (eng. *"cloud native"*) arkitektur växt.

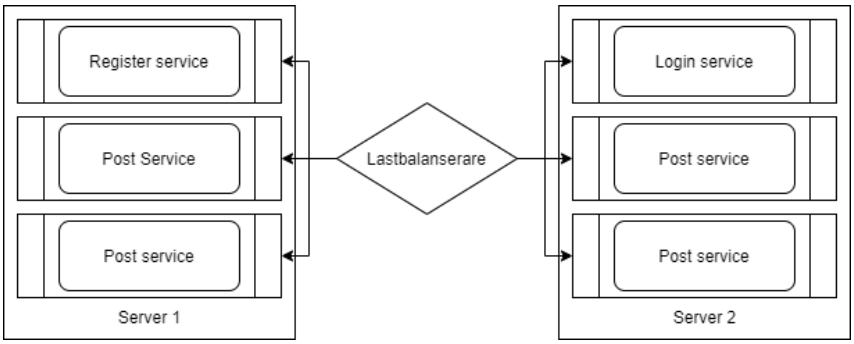
2.2.2 Mikrotjänstarkitektur

Mikrotjänstarkitekturen är en arkitektur som anses kunna lösa problemen som monoliter stöter på allt eftersom deras storlek växer. Fowler och Lewis [12] beskriver mikrotjänstarkitekturen som ett sätt att utveckla ett datorprogram som en samling små oberoende tjänster (mikrotjänster). Mikrotjänsterna är alltså individuella processer som kommunicerar via till exempel ett REST programmeringsgränsnitt (eng. *API - Application Programming Interface*). För att identifiera och dela upp programmets delar i mikrotjänster skall man identifiera företagsförmågor och använda förmågorna som grundstenar vid fördelningen. [12], [13]

Mikrotjänster som följer en bra uppdelning är alltså små i både storlek och funktionalitet, vilket Dragoni et al. [11] betonar att leder till mindre sannolikhet för programfel, samt lättare testning. De påpekar även att eftersom mikrotjänster är oberoende av varandra, så leder det till att distributionen är snabbare, eftersom endast den berörda tjänsten måste startas om och inte hela systemet.

Skalandet av mikrotjänster i jämförelse med skalande av monolitisk arkitektur (Figur 2.2) så skalas endast de delar av programmet som är under störst stress, vilket i Figur 2.3 fall är *Post service*. Att varje mikrotjänst kan skalas självständigt leder till en mer effektiv resursanvändning. Det här kan i sin tur kan leda till mycket mindre kostnader att köra programmet [1].

Dragoni et al. [11] poängterar dock att mikrotjänstarkitekturen har några utmaningar som inte uppstår inom den monolitiska arkitekturen. Till dessa ut-



Figur 2.3: Uppskalningen av mikrotjänstbaserade datorprogram. Anpassad från [2]

maningar hör bland annat säkerheten. Ett exempel på en säkerhetsutmaning de belyser är att attackytan är större för mikrotjänstbaserade datorprogram, i jämförelse med monoliter. Detta beror på att nästan all interaktion mellan mikrotjänster sker via externa kanaler, medan en monolit kan för det mesta använda interna kanaler. Mikrotjänstbaserade datorprogram exponerar därför fler delar av programmet externt, vilket kan leda att datorprogrammet är mer mottagligt för attack.

3 Design av mikrotjänstbaserade program

Mikrotjänstarkitekturen anses alltså vara en arkitektur för att bygga molncentrerade datorprogram (eng. *"cloud-native" applications*). För att kunna designa och utveckla mikrotjänstbaserade datorprogram är det därför viktigt att definiera och förstå vad molncentrerade datorprogram är. Kratzke och Quint [14] påpekar att de inte finns en de facto definition av molncentrerade datorprogram. Men de föreslår följande definition på vad ett molncentrerat datorprogram är:

> A cloud-native application (CNA) is a distributed, elastic and horizontal scalable system composed of (micro)services which isolates state in a minimum of stateful components. The application and each self-contained deployment unit of that application is designed according to cloud-focused design patterns and operated on a self-service elastic platform.

De påpekar också att definitionen som de har kommit fram till består av flera termer som behöver definieras och vidareutvecklas.

Termen mikrotjänster har redan tidigare definierats (se kap. 2.2.2). Det har även termen "self-service elastic platform" (se kap. 2 samt 2.1) dock i denna avhandling under namnet molnbaserade datortjänster. Det resterande termerna elasticitet (eng. *elastic*), skalbarhet (eng. *scalability*) samt fristående distributionsenhet (eng. *self-contained deployment unit*) behöver definieras.

Med termen elasticitet, menas ett system som kan hantera och adaptera sig till nuvarande efterfrågan. Detta uppnås genom att systemets mängd resurser skalas upp och ner, i samma takt som efterfrågan ökar och minskar. Ett system kan vara skalbart på två olika sätt betonar Kratzke och Quint [14]. Att ett system är strukturellt skalbart innebär att systemet kan skalas upp, antingen horisontellt eller vertikalt, utan att man måste göra stora arkitektoniska förändringar. Vertikal skalning innebär att man ger mer beräkningskraft åt existerande hårdvara, medan horisontell skalning innebär att man lägger till

beräkningskraft genom att starta ny hårdvara [15]. Ett system kan också vara lastskalbart, detta innebär att systemet klarar av olika mängder trafik. [14]

En fristående distributionsenhet definieras av Kratzke och Quint [14] som en komponent av datorprogrammet paketerad i en container. En container är ett portabelt sätt att paketera ett datorprogram, dess underkomponenter och bibliotek. Denna container kan sedan köras i olika kompatibla exekveringsmiljöer utan behovet att installera några extra komponenter [16]. På så sätt kan man distribuera olika programkomponenter skilt från varandra på olika hårdvara, utan att behöva installera bibliotek och underkomponenter skilt på varje maskin.

Definitionen av molncentrerade datorprogram som Kratzke och Quint [14] föreslår innehåller också termen molnfokuserade designmönster (eng. *cloud-focused design patterns*). Designmönster är till skillnad från mjukvaruarkitektur (se kap. 2.2) en testad och bra lösning på ett vanligt problem som uppkommer vid utveckling av datorprogram [17]. I de följande stycken kommer några molncentrerade designmönster för utveckling av mikrotjänstbaserade datorprogram beskrivas, samt så kommer fördelarna och nackdelarna med mönstren redovisas.

3.1 Mikrotjänstarkitekturens designmönster

Som tidigare diskuterats är mikrotjänstarkitekturen molncentrerad, vilket innebär att mikrotjänstbaserade datorprogram borde vara elastiska och skalbara. Detta innebär att en mikrotjänst kan ha ett dynamiskt antal instanser av sig själv, samt att det behövs ett sätt för klienter att effektivt anropa mikrotjänstinstanser. Klient hänvisar i detta och följande stycken till exempelvis en mobilapplikation, webbapplikation eller en annan mikrotjänst. När man designar datorprogram enligt mikrotjänstarkitekturen så är denna dynamiska natur något som man är tvungen att ta i beaktande.

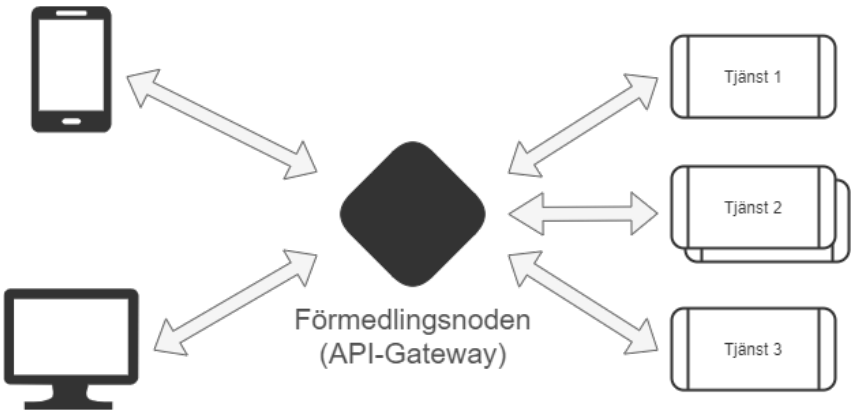
Det enklaste designmönstret som kan användas för att utveckla ett datorprogram enligt mikrotjänsarkitekturen är att klienterna anropar mikrotjänstinstanserna direkt. Nackdelen med denna design är att man oftast förlorar möjligheten att utnyttja mikrotjänstarkitekturens skalbarhet [18]. Detta designmönster kan därför anses vara ett så kallat antimönster (eng. *anti-pattern*). Det krävs alltså ett mera robust designmönster som kan hantera den dynamiska naturen av mikrotjänster.

I Taibi et al. [19] studie: ”Architectural Patterns for Microservices: A Systematic Mapping Study” samt i Richardsons [20] bok ”Microservices Patterns: with examples in Java” identifieras och diskuteras istället två designmönster som är bättre anpassade för molncentrerade och mikrotjänsbaserade datorprogram.

3.1.1 API-gateway mönstret

API-gateway mönstret är ett av designmönstren som kan användas vid utveckling av mikrotjänstbaserad datorprogram. Taibi et al. [19] beskriver API-gateway mönstret som ett sätt att aggregera olika mikrotjänsters gränssnitt till ett sammanslaget och skräddarsytt programmeringsgränssnitt (eng. *tailored API*) per klient. Detta uppnås via en så kallad förmedlingsnod (eng. *gateway*). Figur 3.1 illustrerar förmedlingsnodens centrala uppgift att dirigera trafiken mellan klienterna och mikrotjänsterna. Förmedlingsnoden kan också hantera lastbalansering till nya mikrotjänstinstanser allt eftersom mikrotjänstinstanserna skalas upp och ner.

API-gateway mönstret har flera fördelar över att anropa mikrotjänster direkt från klienten. En fördel är att förmedlingsnoden kan sköta autentisering och övervakning centralt, istället för att varje mikrotjänst måste implementera det skilt. En annan fördel är att förmedlingsnoden kan anropa flera mikrotjänster och slå ihop resultaten. Genom att slå ihop flera resultat minskar det på antalet anrop en klient är tvungen att göra, vilket kan spara på klientens nätverksförbrukning. [19], [20, s. 263–267]



Figur 3.1: API-gateway mönstret. Anpassad från [19]

Taibi et al. [19] betonar också att API-gateway mönstrets gör det möjligt att uppnå bakåtkompatibilitet, eftersom förändring i tjänsterna endast innebär en uppdatering i förmedlingsnoden. Taibi et al. belyser API-gateway mönstrets bakåtkompatibilitet med ett exempel: ”[...] merging or partitioning two or more microservices only requires updating the API-Gateway to reflect the changes to any connected client.”. Som detta exempel visar så behöver alltså inte klienten göra några ändringar utan allt kan ske direkt i förmedlingsnoden.

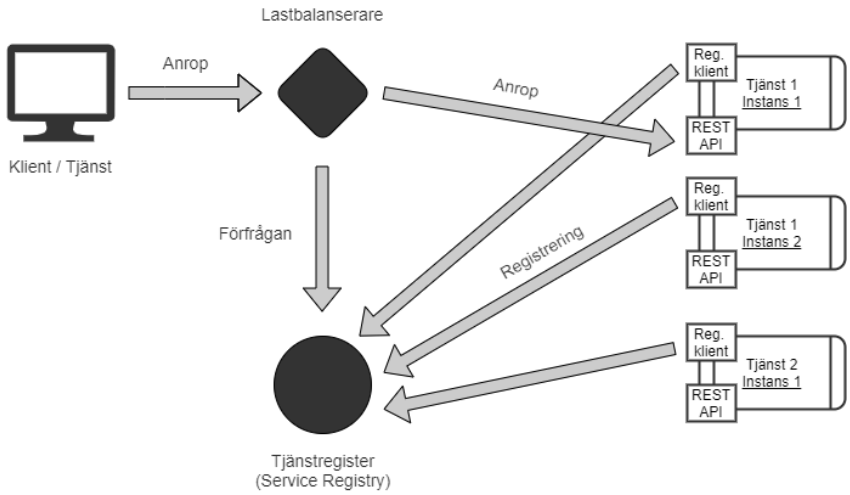
Det existerar också några nackdelar med API-gateway mönstret. Både Richardson [20] och Taibi et al. [19] påpekar att att förmedlingsnodens centrala roll i mönstret (se figur 3.1), betyder att den kan bli en flaskhals ifall den är dåligt implementerad. Taibi et al. [19] identifierar också några nackdelar som kan leda till att utvecklings- och underhållsskalbarheten kan bli invecklad i samband med att mängden tjänster och skräddarsydda programmeringsgränssnitt växer. Problemet uppkommer eftersom varje programmeringsgränsnitt på förmedlingsnoden kan behöva uppdateras ifall en tjänst läggs till eller modifieras.

3.1.2 Service discovery mönstret

Det andra designmönstret som Taibi et al. [19] och Richardson [20] identifierat är tjänsteupptäckts mönstret (eng. *Service discovery pattern)*. Som tidigare diskuterats så kan mikrotjänster ha ett dynamiskt antal instanser, alla med en egen dynamisk adress. Detta innebär att klienter eller andra tjänster som

försöker nå en mikrotjänst behöver få tag på de dynamiska adresserna. Tjänsteupptäckts mönstret löser detta problem genom att utnyttja ett så kallat tjänsteregister (eng. *Service registry*). Tjänsteregistrets uppgift är att upprätthålla en databas på alla mikrotjänstinstansernas dynamiska adresser. Registreringen av nya instanser sker samtidigt de startas, vid start registrerar de sig i tjänsteregistret, via till exempel en inbyggd registrerings klient (se figur 3.2). Tjänsteregistret kan också övervaka de olika instanserna och se till att de är aktiva och kan ta emot anrop. [19], [20]

Taibi et al [19] och Richardson [20] urskiljer två olika typer av tjänsteupptäckts mönstret, dessa är tjänsteupptäckt på klientsidan samt tjänsteupptäckt på serversidan. Tjänsteupptäckt på klientsidan innebär att klienten direkt begär tjänsteregistret efter adresserna till en mikrotjänstsinstans. Klienten kan sedan direkt anropa en instans av mikrotjänsten. Fördelen med denna typ av tjänsteupptäckt är att den är enkel att utveckla. Nackdelarna är att klienten måste vara direkt kopplad till tjänsteregistret, samt att det är klienten som är tvungen att sköta lastbalanseringen.



Figur 3.2: Tjänsteupptäckt på serversidan. Anpassad från [19]

Den andra typen av tjänsteupptäckts mönstret är tjänsteupptäckt på serversidan. Taibi et al. [19] identifierade en större rapporterad användning av detta mönster än med tjänsteupptäckt på klientsidan. Richardson [20] rekommenderar också användning av detta mönster över tjänsteupptäckt på klientsi-

dan. Som figur 3.2 visar så anropar inte klienten tjänsteregistret direkt i detta mönster utan klienten anropar en lastbalanserare. Lastbalanseraren frågar sedan efter den anropade mikrotjänstens instansadresser av tjänsteregistret. Slutligen anropas en tillgänglig instans via lastbalanseraren. Till skillnad från API-gateway mönstret så görs ingen filtrering eller aggregering av resultatet hos lastbalanseraren, klienten anropar egentligen direkt en mikrotjänstinstans programmeringsgränsnitt [19].

Taibi et al. [19] identifierade flera fördelar med tjänsteupptäckt på serversidan. Till fördelarna de identifierade såg de bland annat att kommunikation mellan tjänsterna underlättades, eftersom de kan kommunicera direkt med varandra. De identifierade också fördelar inom underhåll av systemet. Till exempel hanteringen och återhämtningen av kritiska fel i mikrotjänstinstanser underlättas av tjänsteregistrets förmåga att övervaka hälsan av mikrotjänstinstanser. Slutligen betonar de också att användning tjänsteupptäckts mönstret ökade förståelsen av datorprogrammets olika mikrotjänster, samt att utvecklingen av programmet blev lättare.

Användningen av tjänsteupptäckt på serversidan har också ett antal nackdelar som Taibi et al. [19] identifierat. Eftersom tjänsteregistret spelar en lika central roll som förmedlingsnoden gör i API-gateway mönstret, så riskerar tjänsteregistret på samma sätt att bli en flaskhals i systemet. Ifall en mikrotjänsts programmeringsgränsnitt ändras så innebär det att varje klient som använder sig av gränssnittet i fråga måste uppdateras. Taibi et al. [19] påpekar att ett i förtid välplanerat programmeringsgränsnitt underlättar och minskar på mängden ändringar som måste göras vid en uppdatering. På grund av att klienten kommunicerar direkt med mikrotjänsten, istället för via en förmedlingsnod som kan hantera skräddarsydda programmeringsgränsnitt per klient. Detta leder till att komplexiteten att implementera dessa gränssnitt flyttas från en central punkt, som i API-gateway mönstret, ut till varje mikrotjänst. [19].

4 Fallgropar vid design av mikrotjänstbaserade datorprogram

4.1 Skakiga mikrotjänstinteraktioner

4.2 Delad persistens

5 Diskussion

5. Referenser

[1] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano och M. Lang, "Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS lambda architectures," *Service Oriented Computing and Applications*, årg. 11, nr 2, s. 233–247, 1 juni 2017, ISSN: 1863-2394. DOI: 10.1007/s11761-017-0208-y. URL: https://doi.org/10.1007/s11761-017-0208-y (hämtad 2021-02-16).

[2] N. Dragoni, I. Lanese, S. T. Larsen, M. Mazzara, R. Mustafin och L. Safina, "Microservices: How to make your application scale," i *Perspectives of System Informatics*, A. K. Petrenko och A. Voronkov, utg., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2018, s. 95–104, ISBN: 978-3-319-74313-4. DOI: 10.1007/978-3-319-74313-4_8.

[3] N. Alshuqayran, N. Ali och R. Evans, "A Systematic Mapping Study in Microservice Architecture," i *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, nov. 2016, s. 44–51. DOI: 10.1109/SOCA.2016.15.

[4] C. Fehling, F. Leymann, R. Retter, W. Schupeck och P. Arbitter, "Cloud computing fundamentals," i *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*, C. Fehling, F. Leymann, R. Retter, W. Schupeck och P. Arbitter, utg., Vienna: Springer, 2014, s. 21–78, ISBN: 978-3-7091-1568-8. DOI: 10.1007/978-3-7091-1568-8_2. URL: https://doi.org/10.1007/978-3-7091-1568-8_2 (hämtad 2021-02-05).

[5] ——, "Introduction," i *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*, C. Fehling, F. Leymann, R. Retter, W. Schupeck och P. Arbitter, utg., Vienna: Springer, 2014, s. 1–20, ISBN: 978-3-7091-1568-8. DOI: 10.1007/978-3-7091-1568-8_1.

URL: https://doi.org/10.1007/978-3-7091-1568-8_1 (hämtad 2021-03-29).

[6] L. Wang, G. von Laszewski, A. Younge, X. He, M. Kunze, J. Tao och C. Fu, "Cloud computing: A perspective study," *New Generation Computing*, årg. 28, nr 2, s. 137–146, 1 april 2010, ISSN: 1882-7055. DOI: 10.1007/s00354-008-0081-5. URL: https://doi.org/10.1007/s00354-008-0081-5 (hämtad 2021-02-23).

[7] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau och R. H. Arpaci-Dusseau, "Serverless computation with OpenLambda," s. 7,

[8] P. Kruchten, H. Obbink och J. Stafford, "The Past, Present, and Future for Software Architecture," *IEEE Software*, årg. 23, nr 2, s. 22–30, mars 2006, Conference Name: IEEE Software, ISSN: 1937-4194. DOI: 10.1109/MS.2006.59.

[9] Software Engineering Institute. (dec. 2017). "Software architecture," URL: https://www.sei.cmu.edu/our-work/projects/display.cfm?customel_datapageid_4050=21328&customel_datapageid_4050=21328 (hämtad 2021-03-27).

[10] Chris Richardson. (2014). "Microservices Pattern: Monolithic Architecture pattern," microservices.io, URL: http://microservices.io/patterns/monolithic.html (hämtad 2021-02-18).

[11] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin och L. Safina, "Microservices: yesterday, today, and tomorrow," *arXiv:1606.04036 [cs]*, 20 april 2017. arXiv: 1606.04036. URL: http://arxiv.org/abs/1606.04036 (hämtad 2021-02-18).

[12] Martin Fowler och James Lewis. (25 mars 2014). "Microservices," martinfowler.com, URL: https://martinfowler.com/articles/microservices.html (hämtad 2021-02-17).

[13] S. Newman, *Building microservices: designing fine-grained systems*, First Edition. Beijing Sebastopol, CA: O'Reilly Media, 2015, 259 s., OCLC: ocn881657228, ISBN: 978-1-4919-5035-7.

[14] 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, 1 april 2017, ISSN: 0164-1212. DOI: 10.1016/j.jss.2017.01.001. URL: https://www.sciencedirect.com/science/article/pii/S0164121217300018 (hämtad 2021-03-27).

[15] C. Liu, M. Shie, Y. Lee, Y. Lin och K. Lai, "Vertical/Horizontal Resource Scaling Mechanism for Federated Clouds," i *2014 International Conference on Information Science Applications (ICISA)*, ISSN: 2162-9048, maj 2014, s. 1–4. DOI: 10.1109/ICISA.2014.6847479.

[16] OCI. (1 dec. 2016). "Opencontainers/runtime-spec," GitHub, URL: https://github.com/opencontainers/runtime-spec (hämtad 2021-03-28).

[17] R. C. Martin, "Design principles and design patterns," s. 34, 2000.

[18] D. Neri, J. Soldani, O. Zimmermann och A. Brogi, "Design principles, architectural smells and refactorings for microservices: A multivocal review," *SICS Software-Intensive Cyber-Physical Systems*, årg. 35, nr 1, s. 3–15, 1 aug. 2020, ISSN: 2524-8529. DOI: 10.1007/s00450-019-00407-8. URL: https://doi.org/10.1007/s00450-019-00407-8 (hämtad 2021-02-05).

[19] D. Taibi, V. Lenarduzzi och C. Pahl, "Architectural Patterns for Microservices: A Systematic Mapping Study," 23 mars 2018. DOI: 10.5220/0006798302210232.

[20] C. Richardson, *Microservices patterns: with examples in Java*. Shelter Island, New York: Manning Publications, 2019, 490 s., OCLC: on1002834182, ISBN: 978-1-61729-454-9.