



Faculteit Bedrijf en Organisatie

Hoe implementeert men een ecosysteem van microservices bij applicatieontwikkeling?

Nick Gysels

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Heidi Roobrouck
Co-promotor:
Thomas Blommaert

Instelling: AdvantITge

Academiejaar: 2019-2020

Tweede examenperiode

Faculteit Bedrijf en Organisatie

Hoe implementeert men een ecosysteem van microservices bij applicatieontwikkeling?

Nick Gysels

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Heidi Roobrouck
Co-promotor:
Thomas Blommaert

Instelling: AdvantITge

Academiejaar: 2019-2020

Tweede examenperiode

Woord vooraf

Het onderzoek naar microservices is een vrij technisch, maar toch interessant onderwerp. Ik had zelf nog geen ervaring met microservices en dankzij dit onderzoek heb ik een beter inzicht gekregen wat betreft de microservice-architectuur, alsook de voordelen en uitdagingen verbonden aan deze architectuur. Als toekomstig ontwikkelaar is het zeker een meerwaarde om bijgeleerd te hebben inzake deze materie. Graag wil ik daarom mijn co-promotor, **Thomas Blommaert (AdvantITge)**, bedanken dat ik in de eerste plaats de kans kreeg om dit onderzoek te doen en voor de verkregen steun, richting, tips,... tijdens het doorlopen van dit onderzoek. Graag wil ik ook mijn promotor, **Heidi Roobrouck (Ho-Gent)**, bedanken voor de correcte opvolging en controle van het onderzoek. Ik hoop alvast dat wie dit onderzoek nakijkt, ook iets bijgeleerd heeft wat betreft de microservice-architectuur.

Samenvatting

Microservices zijn een zeer populair thema binnen applicatieontwikkeling. Vele grote bedrijven passen deze architectuurvorm reeds enkele jaren toe om zo een hogere beschikbaarheid te kunnen garanderen voor de eindgebruikers. Echter houdt deze architectuurvorm in dat er veel bewegende onderdelen zijn, vooral in vergelijking met een monolithische architectuur. Hierdoor zijn er naast de voordelen heel wat uitdagingen verbonden aan de microservice-architectuur. Daarom is het belangrijk om een zo correct mogelijk beeld te krijgen van wat enerzijds de microservice-architectuur inhoudt om zo een overzicht te krijgen wat de bijhorende voordelen en uitdagingen zijn. Dit inzicht kan voor ontwikkelbedrijven interessant zijn om te evalueren of zij de keuze kunnen leggen bij de microservice architectuur in plaats van de monolithische architectuur. In dit onderzoek worden enerzijds beide architecturen beschreven. Anderzijds wordt ook een overzicht gegeven van de voordelen en uitdagingen verbonden aan de microservice-architectuur. Ook werden voor de verschillende uitdagingen van microservices eventuele oplossingen opgenomen in dit onderzoek.

Men kan concluderen dat de microservice-architectuur heel wat uitdagingen inhoudt, waarbij vele van deze uitdagingen op een geautomatiseerde manier kunnen opgevangen worden door gebruik te maken van *cloud computing*. Het gebruik van *cloud computing* in combinatie met *containerization* behoort ook tot dit onderzoek. Echter kan het onderzoeken van *cloud computing* en alle mogelijke oplossingen en integraties verder onderzocht worden. Daarnaast bleek ook uit dit onderzoek dat er nog enkele uitdagingen, zoals security en testing, zijn die nog verder onderzocht kunnen worden wat betreft de implementatie van microservices.

Inhoudsopgave

1	Inleiding	17
1.1	Probleemstelling	18
1.2	Onderzoeksvraag	18
1.3	Onderzoeksdoelstelling	19
1.4	Opzet van deze bachelorproef	19
2	Stand van zaken	21
2.1	Monolithische architectuur	21
2.2	Microservice-architectuur	23
2.2.1	Voordelen van de microservicearchitectuur	24
2.2.2	Uitdagingen van de microservicearchitectuur	26
3	Methodologie	47
3.1	Inleiding	47

3.2	Theoretisch onderzoek	47
3.2.1	Fase 1	48
3.2.2	Fase 2	48
3.2.3	Fase 3	48
3.2.4	Fase 4	48
3.3	Proof-of-concept	49
4	Proof-of-concept	51
4.1	Inleiding	51
4.2	Accounting applicatie	51
4.2.1	Beheren van klanten	51
4.2.2	Beheren van facturen	52
4.2.3	Beheren van banktransacties	53
4.3	Microservice-architectuur	54
4.3.1	Bepalen van de microservices	54
4.3.2	Ontwikkelen van de microservices	55
4.3.3	Deployment van de microservices	56
5	Conclusie	59
A	Onderzoeksvoorstel	61
A.1	Introductie	61
A.2	Literatuurstudie	61
A.3	Methodologie	62
A.4	Verwachte resultaten	62
A.5	Verwachte conclusies	63

B	Bijlagen	65
B.1	Docker Containerization	65
B.2	Google Kubernetes Engine	68
B.3	Istio	71
	Bibliografie	77

Lijst van figuren

2.1	Voorbeeld van een monolithische applicatie	22
2.2	Voorbeeld van een applicatie bestaande uit verschillende microservices	24
2.3	Voorbeeld van een microservicesysteem met API Gateway en Service Registry	29
2.4	Voorbeeld van het <i>Orchestration</i> communicatiepatroon	30
2.5	Voorbeeld van het <i>Choreography</i> communicatiepatroon	31
2.6	Voorbeeld van een service mesh	32
2.7	Verloop van statussen in een <i>circuit breaker</i>	41
2.8	Kubernetes architectuur	45
4.1	Visuele representatie <i>Client</i> -model	52
4.2	Visuele representatie <i>Invoice</i> -model	53
4.3	Visuele representatie <i>Transaction</i> -model	54
4.4	Visuele representatie van de microservices in de <i>Accounting</i> applicatie	55
4.5	Voorbeeld van verdeling van de ontwikkelteams en technologiekeuzes van microservices in de <i>Accounting</i> applicatie	55
B.1	Docker installatie - versie controle	65

B.2	Docker - Dockerfile en .dockerignore toevoegen	65
B.3	Docker - Dockerfile voorbeeld	66
B.4	Docker - .dockerignore voorbeeld	66
B.5	Docker - Docker-image aanmaken	67
B.6	Docker - Docker-images overzicht	67
B.7	Docker - Docker container starten	68
B.8	Docker - Docker actieve containers overzicht	68
B.9	Docker - Docker-image naar GC Container Registry sturen	68
B.10	Docker - Docker-image in GC Container Registry overzicht	69
B.11	Google Cloud - Project aanmaken	69
B.12	Google Cloud - Kubernetes API toevoegen aan project	70
B.13	Kubernetes - Cluster aanmaken	71
B.14	Kubernetes - Container toevoegen	72
B.15	Kubernetes - Container configureren	72
B.16	Kubernetes - Load balancer toevoegen	73
B.17	Google Cloud - Cluster monitoring	73
B.18	Google Cloud - Cluster logging	74
B.19	Google Cloud - Cluster tracing	74
B.20	Istio - Toevoegen aan cluster	74
B.21	Istio - Service mesh injectie activeren	74
B.22	Istio - Grafana monitoring UI	75

Woordenlijst

Agile software development Deze manier van applicatieontwikkeling staat voor een set van frameworks en toepassingen gebaseerd op de waarden en principes uit het *Manifesto for Agile Software Development*. Deze vorm onderscheidt zich van anderen doordat de focus ligt op de mensen die het werk verrichten en hoe deze samenwerken. Samenwerking en zelf-organiserende teams staan hierbij centraal (Alliance, 2020). 14, 15, 23

API Staat voor *Application Programming Interface* en laat twee applicaties toe om via network calls met elkaar te communiceren (MuleSoft.com, 2020). 24, 28, 34, 35

best practice Een Best Practice is een algemeen aanvaarde afspraak binnen een branche die de meest efficiënte manier standaardiseert om een gewenst resultaat te bereiken. Een Best Practice bestaat over het algemeen uit een techniek, methode of een proces. Het concept impliceert dat als een organisatie de Best Practice volgt, een resultaat met minimale problemen of complicaties wordt gewaarborgd. Een Best Practice is niet dwingend voorgeschreven maar is vrijwillig (Hoogenraad, 2017). 19

branche Een tak op een repository waarop geïsoleerd kan ontwikkeld worden zonder invloed te hebben op andere branches (GitHub.com, 2020a). 13, 33

build Het proces waarbij broncode wordt geconverteerd naar een vorm die ter beschikking kan gesteld worden voor eindgebruikers, testen, Hierbij is het compilatieproces een belangrijke stap, meestal uitgevoerd door een build-tool. (Techopedia, 2020). 33

codebase De totaliteit van alle code. 21, 23, 26, 33, 55

commit Het opslaan van één of meerdere wijzigingen aan één of meerdere bestanden in een branche op een repository. Elke wijziging wordt geregistreerd met een uniek ID waarmee kan worden nagegaan wanneer en door wie de wijzigingen zijn doorgevoerd (GitHub.com, 2020b). 36

- dependencies** Afhankelijkheden. In de context van applicatieontwikkeling kunnen extra libraries, packages,... die nodig zijn voor de applicatie, beschouwd worden als afhankelijkheden van de applicatie.. 27, 41, 42
- deploy** Inzetten, opstellen, in gebruik nemen. 8, 17, 21–25, 31–33, 39, 40, 43, 54, 56
- downtijd** De tijd wanneer een applicatie in productie niet beschikbaar is. 22, 32, 39, 45
- feature** Een eigenschap of voorziening van een applicatie. 14, 21, 23, 25, 31, 57
- firewall** Een beveiligingsapparaat dat inkomend en uitgaand verkeer behandelt. Deze laat verkeer toe of blokkeert verkeer gebaseerd op een set van beveiligingsregels. Het zorgt voor een scheiding dus het interne netwerk en het inkomende verkeer van externe bronnen (Forcepoint.com, 2020). 38
- logging** Het bijhouden van informatie bij bepaalde gebeurtenissen in een systeem in een digitaal logboek. 37, 57
- metric** Meeteenheid. 34, 37–39, 58
- module** Een eenheid waar code gegroepeerd wordt voor het verwezenlijken van een bepaald doel. 21
- monoliet** Bouwdeel uit één stuk steen (van Veen & van der Sijs, 1997). 21
- network call** Verzoek (En.: *request*) van een client aan een server over het netwerk. 13, 24, 27–29, 33, 36, 40
- package** Een eenheid waar code gegroepeerd wordt voor het verwezenlijken van een bepaald doel. 21
- proof-of-concept** Een proof of concept (PoC) is een methode om te demonstreren of bijvoorbeeld een idee, technologie of functionaliteit haalbaar is en aansluit bij de belevingswereld van de beoogde gebruikers. Oftewel of het enige potentie heeft om daadwerkelijk gebruikt te worden zodra het op de markt geïntroduceerd is. Met een proof of concept krijgen ideeën een stuk meer richting. Een Proof of Concept geeft veel sneller het inzicht én bewijs of je idee werkt of niet. Hiermee raken potentiële investeerders sneller overtuigd van jouw slagingskansen (Braun, 2020). 19, 49, 51, 54, 56
- release** Het beschikbaar stellen van aan eindgebruikers van een applicatie, nieuwe features, oplossingen,... (Silva, 2019). 23, 26
- repository** Opslagruimte waar de code zich bevindt. 13, 23, 33
- request** Verzoek (En.: *request*) van een client aan een server. 28–30, 33, 34, 38–40, 57
- resource** Een hulpbron. In de context van applicaties wordt hiermee gerefereerd naar het gebruik van hardware bronnen zoals CPU, RAM-geheugen, 17, 21, 22, 25, 42, 43
- response** Antwoord van een server aan een client. 28–30, 40
- Scrum** Scrum is een vorm van *Agile software development*. 23
- single point of failure** Een enkel punt waarop falen mogelijk is. 22, 28, 29, 34, 36, 37
- tool** Een instrument, een hulpmiddel, een middel. 13, 18, 19
- unit** Een eenheid, één geheel. 17, 22

use case Een use case omvat alle manieren waarop het systeem gebruikt kan worden om een bepaald doel voor een bepaalde gebruiker te behalen. Een complete set use cases geeft je alle zinvolle manieren om het systeem te gebruiken en illustreert de waarde die dit zal opleveren (Malfait, 2015). 17

XP Extreme Programming is een vorm van *Agile software development*. 23

1. Inleiding

Traditioneel worden verschillende soorten applicaties, zoals web- en softwareapplicaties, ontwikkeld volgens een monolithische architectuur. Deze architectuurvorm houdt in dat applicaties ontwikkeld en *gedeployed* worden als één *unit*. Initieel kan deze manier van applicatieontwikkeling voldoende zijn, zolang het project niet te groot wordt en dat er geen schaling van de resources vereist is.

Bij de implementatie van de microservice-architectuur daarentegen wordt de applicatie ingedeeld in verschillende kleine (micro)services, elk met hun eigen verantwoordelijkheid binnen het domein van de businessvereisten. Alle services communiceren met elkaar om een bepaalde functionaliteit te bieden aan de eindgebruiker. Deze architectuurvorm kent vooral verschillende voordelen ten opzichte van de monolithische architectuur zoals onder meer de *herbruikbaarheid*, *modulariteit*, *schaalbaarheid*, *beschikbaarheid* en *fouttolerantie* van de verschillende microservices. Echter zijn er ook verschillende uitdagingen verbonden aan microservices zoals *tijd en budget*, *performantie*, *security*, *error management*, *load balancing*, *service discovery*, *testing* en *monitoring*. *Cloud computing* wordt gezien als oplossing voor verschillende van deze uitdagingen.

Gezien de voordelen en uitdagingen verbonden aan de microservice-architectuur, is het duidelijk dat deze architectuurvorm niet voor alle *use cases* de meest ideale oplossing is. Daarom is het noodzakelijk om goed te weten wat de voordelen en uitdagingen exact inhouden om na te gaan of de keuze bij deze architectuurvorm kan gelegd worden bij toekomstige projecten of om bestaande monolithische applicaties te migreren naar applicaties bestaande uit microservices. In deze context werd dit onderzoek gedaan voor

AdvantITge, een ontwikkelbedrijf te Zwijnaarde. Uit dit onderzoek werd verwacht dat de microservice-architectuur zo volledig mogelijk in kaart gebracht werd, zowel de voordelen als uitdagingen uitvoerig onderzocht werden en de gepaste *tools* of oplossingen voor de gekende uitdagingen onderzocht werden. Het uiteindelijke doel van dit onderzoek bestaat eruit om een *AdvantITge* een inzicht te geven hoe een ecosysteem van microservices te implementeren.

1.1 Probleemstelling

Algemeen zijn ontwikkelaars meer vertrouwd met de monolithische architectuurvorm en houdt de microservice-architectuur echter veel technische en organisatorische uitdagingen in. Hoewel dit onderzoek in de eerste plaats voor *AdvantITge* werd gehouden, kan dit onderzoek ook nuttig zijn voor ontwikkelaars of ontwikkelbedrijven die willen weten wat de voordelen en uitdagingen zijn van de microservice-architectuur. Dit onderzoek zal daarnaast ook een beschikbaar overzicht van *tools* en oplossingen bieden voor de uitdagingen van deze architectuurvorm. Hiermee kan dit onderzoek ontwikkelaars of ontwikkelbedrijven helpen om de keuze te maken om een nieuw project te starten als microservices, een bestaand monolithische applicatie te migreren naar een microservices applicatie of duidelijk maken dat de technische en organisatorische uitdagingen van microservices niet opwegen ten opzichte van de voordelen van het gebruik van microservices, vooral bij projecten van kleinere schaal.

In het kader van de onderzoeksnoed van *AdvantITge*, zal het onderzoek uitmaken of het voor het bedrijf de juiste keuze is om volledig over te schakelen naar een microservice-architectuur en vormt dit onderzoek een goede basis om rekening te houden met de uitdagingen van microservices en hoe hiermee om te gaan. De interesse van *AdvantITge* om te kiezen voor de microservice-architectuur is vooral te wijten aan de herbruikbaarheid van de microservices, waardoor dit voor het ontwikkelbedrijf een optimale keuze kan zijn op lange termijn. Deze keuze zal echter afhangen van het gewicht van de voordelen en uitdagingen van deze architectuur waarbij de oplossingen voor de uitdagingen mee in rekening zullen gebracht worden.

1.2 Onderzoeksvraag

Het uiteindelijke doel van dit onderzoek bestaat eruit om een inzicht te geven hoe een ecosysteem van microservices geïmplementeerd kan worden. Om een zo uitgebreid mogelijk antwoord te bieden op de onderzoeksdoelstelling van dit onderzoek, werden de volgende onderzoeksvragen opgesteld:

- Wat is de microservice-architectuur (in vergelijking met de monolithische architectuur)?
- Wat zijn de voordelen van de microservice-architectuur?
- Wat zijn de uitdagingen verbonden aan de microservice-architectuur?
- Wat zijn de beschikbare *tools* en oplossingen voor de uitdagingen verbonden aan de

microservice-architectuur?

1.3 Onderzoeksdoelstelling

Afgaande op de onderzoeksvragen die werden verwoord in de voorgaande sectie, werden volgende elementen aanwezig geacht in dit onderzoek:

- Een vergelijkende studie tussen de monolithische en microservice-architectuur bij applicatieontwikkeling.
- Een beschrijvende studie van de verschillende voordelen en uitdagingen verbonden aan de microservice-architectuur.
- Een beschrijvende studie van de beschikbare *tools* en oplossingen voor de implementatie van de microservice-architectuur.
- Een *proof-of-concept* waarbij een applicatie ontwikkeld wordt onder de microservice-architectuur. Hierbij moeten de *best practices* toegepast zijn om microservices te implementeren en verschillende oplossingen en *tools* gebruikt worden om een oplossing te bieden voor de uitdagingen verbonden aan microservices.

1.4 Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:

In Hoofdstuk 3 wordt de methodologie toegelicht en worden de gebruikte onderzoekstechnieken besproken om een antwoord te kunnen formuleren op de onderzoeksvragen.

In Hoofdstuk 2 wordt een overzicht gegeven van de stand van zaken binnen het onderzoeksdomein, op basis van een literatuurstudie. Hierbij worden beide architectuurvormen besproken en zowel de voordelen als uitdagingen verbonden aan de microservice-architectuur.

In Hoofdstuk 4 wordt een *proof-of-concept* toegelicht waarbij verschillende elementen uit de literatuurstudie gebruikt worden om de microservice-architectuur toe te passen.

In Hoofdstuk 5, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek binnen dit domein.

2. Stand van zaken

Inleiding

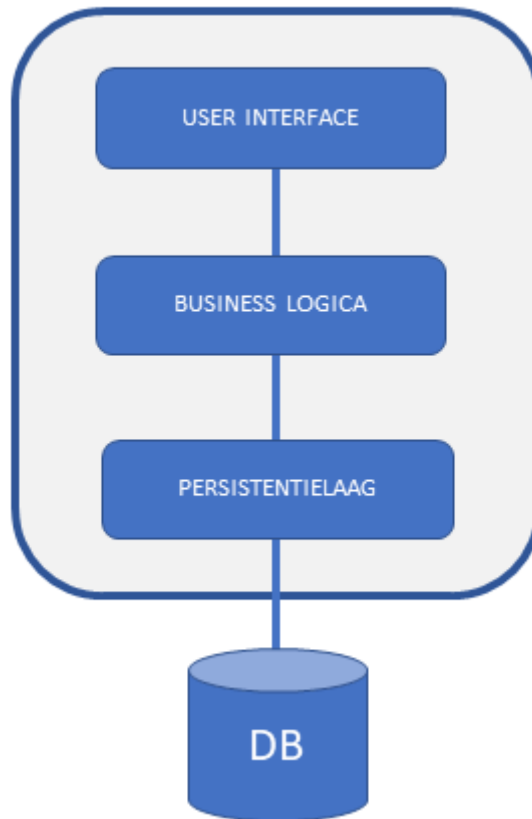
In dit hoofdstuk worden zowel de monolithische als de microservice-architectuur uitvoerig besproken. Daarnaast worden zowel de voordelen als de uitdagingen verbonden aan de microservice-architectuur ook gedetailleerd besproken.

2.1 Monolithische architectuur

De betekenis van het woord *monoliet* vindt zijn oorsprong uit het Frans en betekent *bouwdeel uit één stuk steen* (van Veen & van der Sijs, 1997). Een monolithische applicatie heeft een architectuurvorm waarbij de applicatie dus als één stuk kan beschouwd worden. Hoewel de applicatie, afhankelijk van de gebruikte technologie, kan bestaan uit verschillende *modules, packages,...* wordt de applicatie als één geheel stuk gezien. Hierbij wordt de applicatie als één geheel ontwikkeld, getest en *gedeployed* (Kalske e.a., 2018). Zie figuur 2.1 voor een grafisch voorbeeld van een monolithische applicatie. Deze architectuurvorm houdt dus in dat:

- de applicatie makkelijk te ontwikkelen is indien de applicatie niet te groot is. Een grote hoeveelheid code kan de productiviteit of codekwaliteit verlagen. Dit kan inhouden dat er een grotere kans is op dubbele code, dat het moeilijker is voor ontwikkelaars om de plaats te vinden in de *codebase* waar wijzigingen of nieuwe *features* moeten ontwikkeld worden,... . Bij integratieproblemen is het ook moeilijker om de oorsprong van het probleem terug te vinden in de totale code. Mocht er een probleem zijn in productie (bijvoorbeeld het uitvallen van *resources* waardoor de applicatie tijdelijk niet beschikbaar is), dan is dit sneller duidelijker voor

Monolithische applicatie



Figuur 2.1: Voorbeeld van een monolithische applicatie

de eindgebruiker, doordat de applicatie als geheel niet beschikbaar is. Dit is op zich niet gebruiksvriendelijk, maar dergelijke problemen worden daarentegen snel gedetecteerd.

- de applicatie makkelijk te *deployen* is omdat slechts één *unit* in productievorm wordt *gedeployed*. Bij een update dient de volledige applicatie *geredeployed* te worden, ongeacht de grootte van de wijziging. Daarnaast zal dit ook resulteren in een grotere *downtijd* wanneer het *redeployment* van de applicatie plaatsvindt. Een bijkomend probleem bij het *deployment* is dat als één service van de applicatie stopt met werken, de volledige applicatie niet beschikbaar is. Men kan de applicatie dus beschouwen als een *single point of failure*.
- de applicatie in productie enkel horizontaal geschaald kan worden. Dit betekent dat *hardware-resources*, zoals geheugengebruik onder meer, toegewezen worden aan de volledige applicatie. Het kan zijn dat bepaalde delen (*modules*, *packages*) van de applicatie minder *resources* nodig hebben om hun taken te voltooien en is dit, door de uniforme toewijzing van *resources*, niet optimaal bij een monolithische applicatie.
- de applicatie moeilijker te begrijpen is voor nieuwe ontwikkelaars die zich aansluiten bij het project. De inwerkingsperiode in de volledige applicatie zal langer duren door het totale pakket aan functionaliteiten en verantwoordelijkheden binnen de applicatie.

Er is doorgaans één gedeelde *codebase* op één *repository*. Daarnaast is er ook een grotere coördinatie vereist van de ontwikkelaars om nieuwe *features* of wijzigingen te integreren in de applicatie. Door deze integratie kan het zijn dat *reddeployments* minder frequent gebeuren waardoor dat de veranderingen zich opbouwen tijdens de *releases* van de applicatie.

- de applicatie ontwikkeld wordt binnen éénzelfde technologie. Hierdoor is een volledige wijziging van de technologie moeilijker omdat de volledige applicatie veranderd moet worden.

In bepaalde gevallen kan de monolithische architectuur een goede keuze zijn voor applicatieontwikkeling. De bovengenoemde elementen zijn vooral afhankelijk van de grootte van de applicatie volgens Kalske e.a. (2018). De nadelen of uitdagingen die kunnen optreden bij een monolithische applicatie kunnen opgelost worden door de keuze te leggen bij de microservice architectuur. Echter heeft deze architectuurvorm ook zijn eigen nadelen of uitdagingen. Het is dus belangrijk om te weten of de uitdagingen verbonden aan de microservices voldoende opgevangen kunnen worden om tegemoet te komen aan de uitdagingen verbonden aan de monolithische architectuur. Hiervoor is het vooral belangrijk om enerzijds te weten wat de microservice-architectuur inhoudt en anderzijds wat de voordelen en uitdagingen zijn van microservices. Dit alles wordt besproken in het volgende onderdeel.

2.2 Microservice-architectuur

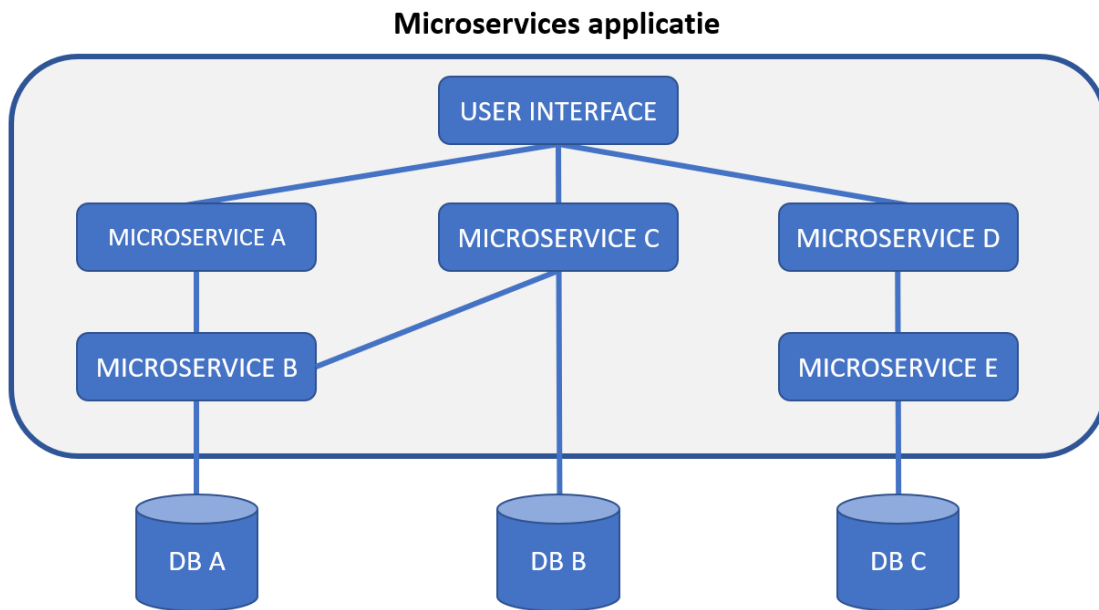
Volgens Newman (2015) is de microservice-architectuur afgeleid uit de *Service-Oriented Architectuur (SOA)* waarbij meerdere services samenwerken om een volledige set functionaliteit te voorzien. Het doel van *SOA* is om het hergebruik van software aan te moedigen en om software makkelijker te onderhouden. Microservice-architectuur is een specifieke aanpak voor *SOA* zoals *XP* of *Scrum* een specifieke aanpak voor *Agile software development* zijn (Newman, 2015).

De microservice-architectuur houdt in dat een volledige applicatie ingedeeld wordt in verschillende kleine microservices, waarbij *loose coupling* en *high cohesion* centraal staan (Dragoni e.a., 2017).

Loose coupling van de microservices houdt in dat een verandering in een bepaalde microservice geen invloed mag hebben op andere microservices. *High cohesion* houdt in dat er getracht moet worden gerelateerd gedrag te centraliseren. Elke microservice heeft zijn eigen afgebakende verantwoordelijkheid waarbij gerelateerde code zo goed als mogelijk gegroepeerd wordt. Volgens Newman (2015) wordt dit beschreven als het *Single Responsibility Principle*.

Zie figuur 2.2 voor een grafisch voorbeeld van een applicatie bestaande uit verschillende microservices. Daarnaast worden microservices onafhankelijk ontwikkeld, getest, *gedeployed* en hebben doorgaans hun eigen database, wat hun geïsoleerd karakter verder benadrukt. Een bijkomend voordeel aan deze architectuur is de vrije technologiekeuze per microservice.

Binnen de totale applicatie communiceren microservices met elkaar aan de hand van



Figuur 2.2: Voorbeeld van een applicatie bestaande uit verschillende microservices

network calls (API's). Microservices zijn relatief klein waarbij men kan stellen dat hoe kleiner de service is, hoe meer de voordelen en nadelen van de microservice-architectuur gemaximaliseerd worden. Als de services kleiner worden, nemen de voordelen zoals onderlinge onafhankelijkheid toe, maar neemt de complexiteit van de verschillende bewegende delen toe (Newman, 2015).

2.2.1 Voordelen van de microservicearchitectuur

De microservice architectuur omvat heel wat voordelen die in dit onderdeel besproken worden.

Modulariteit

Elke microservice wordt individueel en onafhankelijk ontwikkeld, getest en *gedeployed*. Elke service heeft ook zijn eigen afgebakende verantwoordelijkheden. Echter kan het definiëren van deze grenzen een uitdaging zijn en wordt dit verder besproken in sectie 2.2.2 (Uitdaging - Bepalen van de services). Door de modulariteit van de microservices kunnen bepaalde functionaliteiten herbruikt worden binnen de applicatie of zelfs door verschillende applicaties. Door het verbergen van implementatiedetails wordt het modulaire karakter van de services versterkt (*loose coupling*) (Newman, 2015). Wanneer *Microservice A* uit figuur 2.2 gegevens opvraagt aan *Microservice B*, dan maakt het voor *Microservice A* niet uit hoe *Microservice B* deze gegevens verzamelt. Hierdoor wordt het *Single Responsibility Principle* (zie 2.2) gehandhaafd. Enkel een uniforme manier van communiceren is hierbij van belang (zie sectie 2.2.2 - Uitdaging communicatie). Daarnaast kan, door uniforme communicatiepatronen, *Microservice B* zijn diensten (services) ter beschikking stellen van andere services, waarmee het hergebruik van microservices mogelijk wordt. Verder zijn er

andere voordelen verbonden aan microservices die nauw samenhangen met het modulair karakter van microservices. Deze worden besproken in onderstaande secties.

Vrije technologiekeuze

Door hun geïsoleerd karakter, is de technologiekeuze bij microservices vrij te bepalen per microservice. Hierdoor kan er een technologie gekozen worden die beter aansluit bij de vereisten van de microservice. Het type databank, de gebruikte programmeertaal,... zijn enkele voorbeelden hiervan (Newman, 2015). Er dient wel rekening gehouden te worden dat microservices op een uniforme manier moeten kunnen communiceren (zie 2.2.2 - Uitdaging communicatie).

Doordat elke microservice zijn eigen technologieën heeft en doordat elke microservice beperkt is in grootte, is het makkelijker om over te schakelen naar een andere technologie indien dit nodig zou zijn. Het is dan ook duidelijk dat de impact bij technologiewijzigingen binnen een microservice minder impact zal hebben ten opzichte van technologiewijzigingen bij een monolithische applicatie, waar de volledige applicatie veranderd moet worden.

Zo kan het zijn dat *Microservice A* uit het voorbeeld in figuur 2.2 geïmplementeerd is in Java en dat *Microservice B* geïmplementeerd is in NodeJS.

Schaalbaarheid

Microservices kunnen individueel geschaald worden, waarbij getracht wordt een hoge beschikbaarheid en fouttolerantie te verzekeren van de verschillende microservices. Dit wil zeggen dat *resources* (hardware-capaciteiten van de host) op maat kunnen toegekend worden aan de microservices. Er kunnen microservices zijn die CPU-intensieve operaties uitvoeren en andere die minder CPU-intensieve operaties uitvoeren. Elk van deze services zal dus andere hardware-capaciteiten nodig hebben om zijn operaties te voltooien. Er is minder sprake van overcapaciteit aan *resources* zoals dat bij een monolithische applicatie meestal wel het geval is. Bij een monolithische applicatie worden de *resources* toegekend aan de volledige applicatie, ongeacht de *resources* die nodig zijn voor de operaties, waardoor men kan stellen dat voor kleine operaties dus een overcapaciteit is aan *resources* (Villamizar e.a., 2015).

Voor microservices is het ook aan te raden de services te verdelen over verschillende hosts, zodat wanneer er een panne is bij een host, deze zo weinig mogelijk invloed heeft op de andere services. Mochten alle services op één host draaien, zullen alle microservices onbeschikbaar zijn bij een panne bij de host. Door ze te verdelen over verschillende hosts, verspreidt men het risico.

Een *feature* verbonden aan de schaalbaarheid is *auto-scaling*. Hosts kunnen de *resources* aan de instanties van de applicatie op een dynamische manier toekennen op basis van het verkeer naar de instanties. Zo kunnen meer instanties van services door de hosts gedeployed worden op piekmomenten of net minder op dalmomenten volgens Balalaie e.a. (2018). Dit alles is kostenbesparend want men betaalt voor effectief gebruikte *resources*. Cloud platformen bieden hier doorgaans de oplossing door het schalen van microservices te automatiseren (zie sectie 2.2.2 - Uitdaging cloud computing).

Veerkracht

Newman (2015) stelt dat er geen sprake is van een watervaleffect wanneer één microservice stopt met functioneren. In tegenstelling tot een monolithische applicatie, waar de volledige applicatie stopt met functioneren bij een fout, zorgt een applicatie gebaseerd op microservices voor een hogere fouttolerantie.

Daarnaast zijn microservices flexibel omdat er makkelijk ingespeeld kan worden op veranderingen, zowel technologische veranderingen als veranderingen binnen de business-vereisten (Dragoni e.a., 2017).

Teamwork

Doordat de microservices onafhankelijk ontwikkeld worden, zijn de teams doorgaans kleiner en werken dus aan kleinere codeblokken. Volgens Newman (2015) moet er getracht worden om het aantal ontwikkelaars aan een bepaalde service te minimaliseren. Dit zal uiteindelijk leiden tot een verhoogde productiviteit van het team. Door de afgebakende verantwoordelijkheden van de services zijn de teams gespecialiseerd in het domein van hun microservice en is er ook minder coördinatie vereist tussen de verschillende teams om nieuwe *releases* te doen van de microservices.

Een bijkomend voordeel van onafhankelijke microservices is dat het makkelijker is voor nieuwe teamleden om zich in te werken in de bestaande *codebase* van een microservice. Het nieuwe teamlid zal zich volledig inwerken in het domein van een microservice.

Bij een monolithische applicatie is dit moeilijker omdat er veel meer *codebase* is en dat er veel meer samenhangende delen zijn (*high coupling*).

Zo zal een nieuw teamlid voor *Microservice A* uit het voorbeeld in figuur 2.2 na verloop van tijd gespecialiseerd zijn in het domein van *Microservice A*. Door het *Single Responsibility Principle* hoeft dit teamlid geen kennis te hebben van het domein van andere microservices. Dit zorgt ervoor dat de onafhankelijke teams problemen uiteindelijk sneller kunnen detecteren en oplossen.

2.2.2 Uitdagingen van de microservicearchitectuur

Hoewel de microservicearchitectuur enkele voordelen inhoudt, zijn er ook enkele uitdagingen waarmee men rekening moet houden wanneer men deze architectuur wenst toe te passen. Deze uitdagingen worden in dit onderdeel besproken.

Bepalen van de services

Het is belangrijk om het volledige domein van de applicatie te kennen voordat de verantwoordelijkheden van de verschillende services vastgelegd worden. Het is moeilijk om hiervoor een vaste maatstaf te bepalen omdat elke applicatie uniek opgebouwd is en ook anders gebruikt wordt. Bij de migratie van een monolithische applicatie naar een applicatie op basis van microservices, is het alvast aangeraden om de reeds bestaande architectuur zowel visueel als informeel te gaan vastleggen (Balalaie e.a., 2018). Hierbij kan men

onder meer communicatierichtingen van services, afhankelijkheden van componenten,... documenteren. Het uiteindelijke doel is een inzicht te krijgen van de dynamiek van de monolithische applicatie.

Voor het bepalen van de grenzen van de microservices zijn er volgende opties waarmee men rekening kan houden:

- **Organisatie:** Men kan zich onder meer richten op afgebakende contexten van de organisatie. Zo kunnen de verschillende afdelingen van de organisatie (bijvoorbeeld *magazijn, finance, ...*) een goed startpunt vormen om de verantwoordelijkheden (contexten) vast te leggen per microservice. Deze contexten, *bounded contexts* genoemd door Newman (2015), kunnen op zich ook verder onderverdeeld worden in andere contexten (*nested contexts*) als deze ook als onafhankelijke microservice kunnen beschouwd kan worden. (Newman, 2015)
- **Domein:** Door het evalueren van het domein (*Domain-driven design (DDD)*) van de applicatie, kan men subdomeinen identificeren die als microservice geïmplementeerd kunnen worden. (Balalaie e.a., 2018)
- **Migratie monolith naar microservices:** Wanneer men een monolithische applicatie, gebruikmakend van modules, wil omvormen tot een microservice applicatie, dan vormen de modules van de monolithische applicatie goede kandidaten om de verantwoordelijkheden vast te leggen van de verschillende services. Hierbij is het aangeraden om te letten op de interne afhankelijkheden (*dependencies*) van de verschillende modules. Een module met veel afhankelijkheden naar andere modules, zal echter moeilijk te isoleren zijn. Men kan ervoor kiezen om deze modules te groeperen tot één microservice. Een module met weinig of geen afhankelijkheden zal echter makkelijker te isoleren zijn tot een microservice (Newman, 2015). *Structure101* is een tool die kan helpen bij het visueel weergeven van verschillende *dependencies* binnen een project. Ook *SchemeSpy* kan een handige tool zijn door het documenteren van de database om zo koppelingen tussen verschillende tabellen terug te vinden.
- **Security:** Men ook kan kijken naar de beveiliging van het systeem. Het kan zijn dat een volledig deel van de applicatie (beter) beveiligd moet zijn, zoals een financiële module. Deze module kan men eventueel als aparte microservice beschouwen omdat op deze manier de individuele beveiligingsvoorzieningen voor deze microservice geïmplementeerd kunnen worden (zie sectie 2.2.2 - Uitdaging security). (Newman, 2015)
- **Geografische verspreiding van het team:** Het is zelfs mogelijk dat bepaalde delen van een applicatie geografisch verdeeld zijn. Zo kan een deel van een applicatie in België ontwikkeld worden en een ander deel in Nederland. Men kan hier eventueel opteren voor de grenzen van de microservices te baseren op de geografische verspreiding van de teams. (Newman, 2015)

Communicatie

Bij een monolithische applicatie worden functionaliteiten intern in de applicatie afgehandeld. Bij een microservice systeem kan dit echter een uitdaging zijn, gezien de communicatie tussen microservices gebeurt over het netwerk aan de hand van *network*

calls (Newman, 2015). Hierbij is het van belang is om standaarden vast te leggen voor de APIs van de verschillende microservices. Namiot en Sneps-Snepe (2014) vat volgende primitieven samen wat betreft communicatie in microservice systeem:

- Gebruik van network calls met gestructureerde data.
- Communicatie kan van en naar een microservice gaan (beide richtingen).
- Gebruik van een standaard formaat voor communicatie zoals *JSON*, *XML*, etc.

Hoewel er rechtstreeks met microservices kan gecommuniceerd worden, is het volgens Li e.a. (2020) aangeraden om de externe communicatie met het systeem van microservices te laten verlopen via een *API gateway*. Het gebruik van een *API gateway* heeft volgende voordelen:

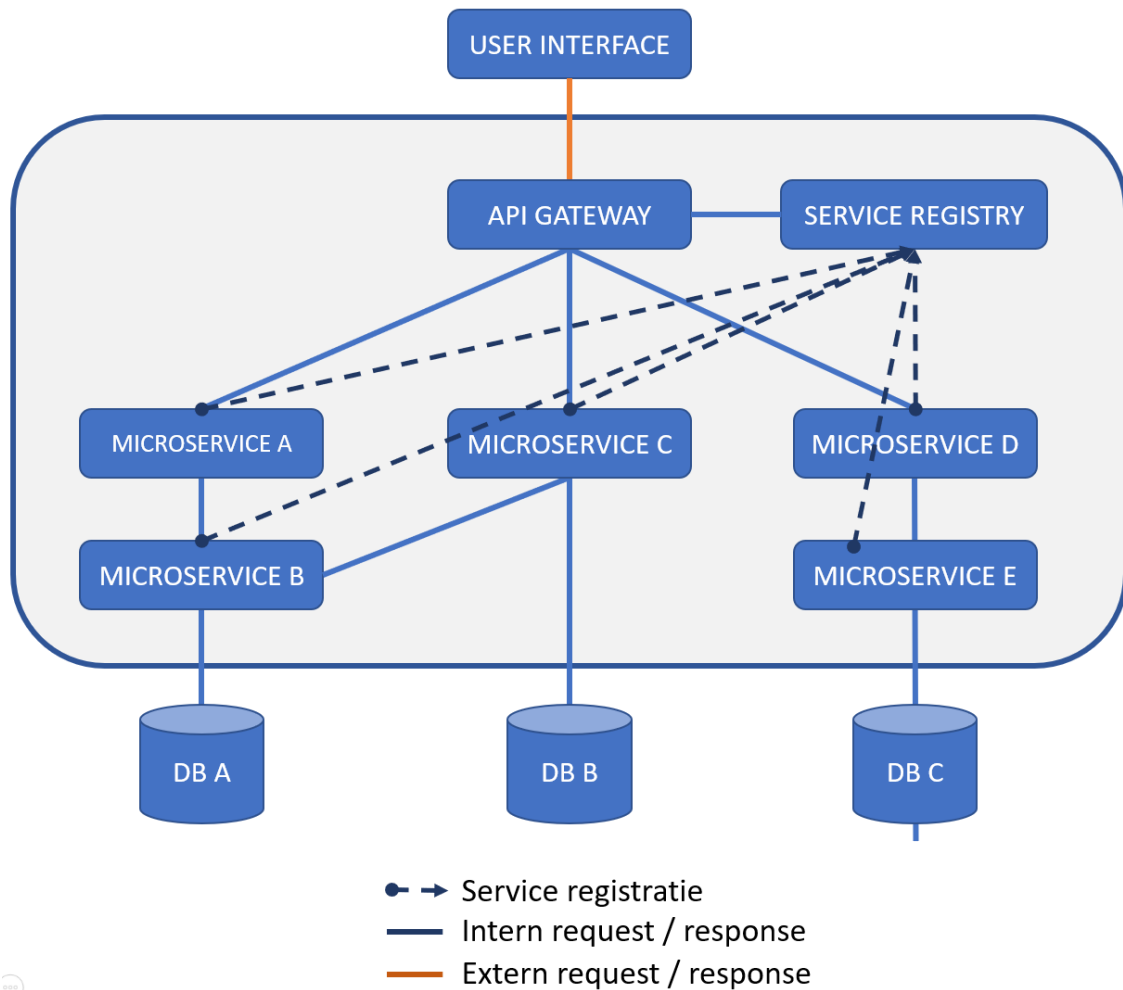
- Men kan het aantal *network calls* verminderen door voor microservices caching te voorzien waarbij vaak opgevraagde data snel ter beschikking kan gesteld worden.
- Er is één centraal punt voor externe *requests* waardoor de interne configuratie van de microservices afgeschermd blijft.
- Er is één centraal punt voor monitoring (zie sectie 2.2.2 - Uitdaging monitoring).
- Er is één centraal punt waar specifieke beveiligingsmaatregelen kunnen geïmplementeerd worden (zie sectie 2.2.2 - Uitdaging security)

Zie figuur 2.3 voor een visuele representatie van een systeem gebaseerd op de microservice-architectuur in combinatie met een *API gateway*. Een *API gateway* omvat alle interfaces van alle microservices en is verantwoordelijk voor het doorsturen van de requests van de gebruikers naar de desbetreffende microservices. Echter is het grote nadeel van een *API gateway* dat deze een *single point of failure* vormt.

Newman (2015) categoriseert twee specifieke technologieën omtrent communicatie bij microservices:

- ***Request/response technologie***: synchrone communicatie waarbij *Microservice A* een request stuurt naar *Microservice B* en wacht op een response van *Microservice B*. Hierbij kan er vertraging optreden doordat er gewacht wordt op een antwoord. Implementaties zoals *Remote Procedure Calls*, *REST*, etc... zijn hier mogelijk.
- ***Async Event-Based technologie***: asynchrone communicatie waarbij *Microservice A* een *event* verstuurd wordt naar *Microservice B*. *Microservice A* hoeft niet te wachten op het resultaat, maar wordt verwittigd wanneer het resultaat verzonden is.

Het spreekt voor zich dat de asynchrone *event-based* communicatietechnologie het beste aansluit bij het onafhankelijke karakter van microservices, waarbij *loose coupling* tussen de microservices gerespecteerd dient te worden. Een microservice heeft bij dergelijke technologie enkel de verantwoordelijkheid om een *event* te versturen wanneer nodig of om reageren op inkomende *events*. Het nadeel aan deze communicatievorm is dat er geen controle is over de status van het proces. Wanneer een *event* verstuurd wordt, kan men moeilijk nagaan of de corresponderende microservices ook degelijk het *event* ontvangen hebben en hun taak al dan niet uitgevoerd hebben. Men kan hierbij eventueel kiezen om gebruik te maken van een *message broker*. De *events* verstuurd door de microservices worden naar de *message broker* gestuurd. Ook hier kan dit centraal onderdeel voor meer



Figuur 2.3: Voorbeeld van een microservicesysteem met API Gateway en Service Registry

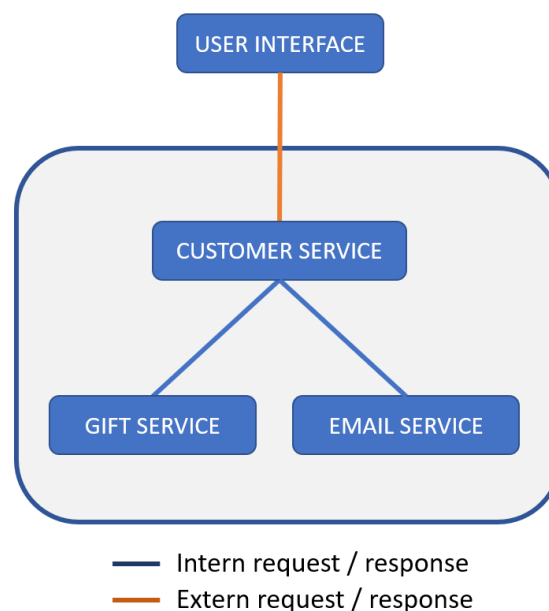
zekerheid zorgen. Doordat microservices zich abonneren via dit centraal orgaan kan er ook onder meer geïnformeerd worden wanneer een *event* al dan niet is aangekomen bij de gewenste microservice.

Newman (2015) specificeert ook twee communicatiepatronen binnen een microservice-systeem, hoe microservices met elkaar communiceren om een bepaalde functionaliteit te voorzien over de verschillende grenzen (zie sectie 2.2.2 - Uitdaging Bepalen van de microservices) van de individuele microservices:

- **Orchestration:** Bij dit patroon is er een centrale microservice die het volledige proces initieert en opvolgt, meestal aan de hand van een reeks *request/response network calls*. Men kan deze centrale microservice vergelijken met een dirigent van een koor, waarbij de leden van het koor de andere individuele microservices voorstellen. Het nadeel van dit patroon is dat de centrale microservice enerzijds teveel autoriteit heeft (*high coupling*) en anderzijds een *single point of failure* vormt. Echter kan men door het gebruik *Request/Response* technologieën wel een betere monitoring voorzien doordat de status (gelukt, niet-gelukt) van het proces beter opgevolgd kan worden.
- **Choreography:** Bij dit patroon wordt elke microservice geïnformeerd over zijn taak,

meestal op basis van een event van een andere microservice.

Een voorbeeld voor een implementatie van beide communicatiepatronen zou volgende kunnen zijn: Wanneer een gebruiker zich registreert (*CustomerService*) op een webshop moet het systeem ook andere functionaliteiten voorzien zoals het versturen van een e-mail (*EmailService*) en het versturen van een welkomstgeschenk (*GiftService*). Als men kiest voor *orchestration* communicatie voor het uitvoeren van dergelijke functionaliteit, dan kan de *CustomerService* als het centrale brein gezien worden die het proces initialiseert en verder opvolgt. Het zal dus de *EmailService* en *GiftService* verwittigen dat er een gebruiker aangemaakt is en dat ze beide hun taak moeten uitvoeren. Zie figuur 2.4 voor een visuele voorstelling van dit voorbeeld met gebruik van het *Orchestration* communicatiepatroon. Wanneer men echter kiest voor een choreografische vorm van communicatie zou men



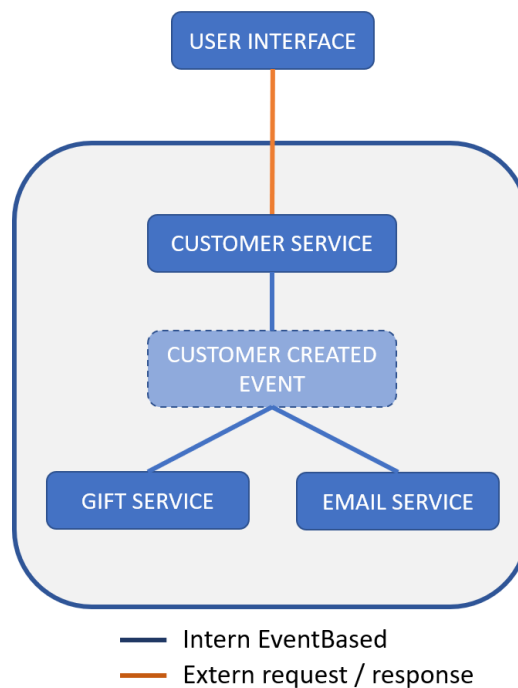
Figuur 2.4: Voorbeeld van het *Orchestration* communicatiepatroon

kunnen stellen dat de *CustomerService* een *event* verstuurt waarop de andere microservices kunnen reageren wanneer het *event* zich voordoet. De *CustomerService* hoeft niet te weten welke microservices een taak moeten volbrengen wanneer een gebruiker is aangemaakt. Zie figuur 2.5 voor een visuele voorstelling van dit voorbeeld met gebruik van het *Choreography* communicatiepatroon.

Samengevat kan men volgende communicatiemogelijkheden stellen bij een microservice-systeem:

- *Orchestration* communicatiepatroon (synchroon *Request/Response*)
- *Choreography* communicatiepatroon (asynchroon *Event-Based*)
- Combinatie van beide

Hierbij heeft het *Choreography* communicatiepatroon de voorkeur omdat deze beter aansluit bij de principes van microservices: *high cohesion* en *loose coupling*.



Figuur 2.5: Voorbeeld van het *Choreography* communicatiepatroon

Echter wordt er bij *cloud computing* (zie sectie 2.2.2 - Uitdaging *cloud computing*) vaak gebruik gemaakt van een georchestreerd patroon. Netwerkcommunicatie bij microservices bij *cloud computing* kan doorgaans beheerd worden door een *service mesh*.

Service mesh

(Zhang, 2020) Microservices communiceren met elkaar via het netwerk, waarbij een *service mesh* kan gebruikt worden om deze onderlinge communicatie af te handelen en een makkelijke / flexibele manier te voorzien netwerkfuncties tussen microservices te automatiseren. De *service mesh* controleert en observeert het netwerkverkeer tussen de verschillende microservices. Hierdoor kan een *service mesh* een positieve bijdrage leveren tot enkele uitdagingen verbonden aan de microservice-architectuur:

- **Monitoring:** inzicht in netwerkverkeer tussen de microservices en logs.
- **Fouttolerantie:** *features* zoals gezondheidscontrole (*health management*), *circuit breaker* (zie sectie 2.2.2 - Uitdaging fouttolerantie), *retries* (pogingen)/*timeouts*, ... zijn mogelijk.
- **Testing:** *canary releasing* (zie sectie 2.2.2 - Uitdaging testen)

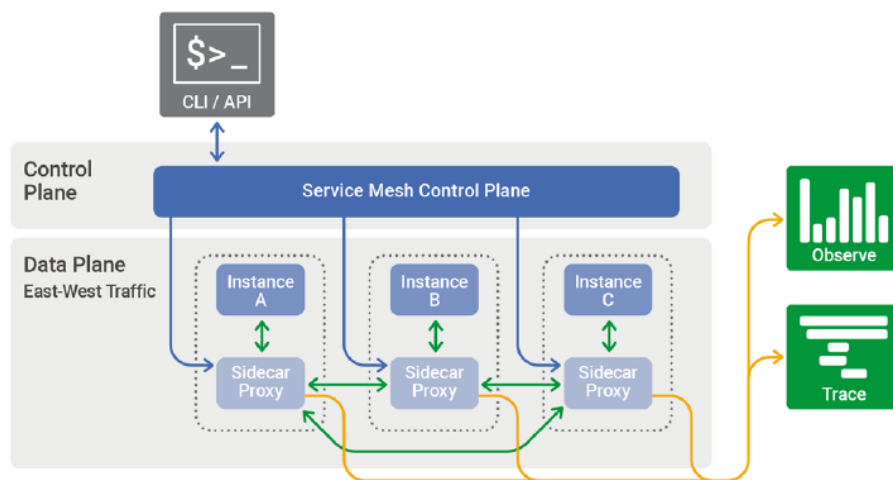
Een *service mesh* bestaat doorgaans uit twee componenten:

- **Data plane:** Deze handelt netwerkverkeer af tussen de microservices, door het deployen van een *sidecar proxy* samen met elke microservice instantie. Deze *sidecar proxy* onderschept het netwerkverkeer van en naar een microservice. Door het analyseren van de pakketten op de netwerklaag kunnen de microservices beter geob-

serveerd worden en kan sneller voorkomen worden dat een microservice overladen wordt wat betreft netwerkverkeer.

- **Control plane:** Deze reguleert het gedrag van de *data plane* aan de hand van configuraties gedefinieerd door de gebruiker.

Figuur 2.6 geeft een visuele representatie van een *service mesh*. Volgende platformen bie-



Figuur 2.6: Voorbeeld van een service mesh

Bron: Zhang (2020)

den *service mesh* technologieën aan, vaak integreerbaar in verschillende *cloud* platformen (zie sectie 2.2.2 - Uitdaging *cloud computing*):

- **Istio:** Wang en Ma (2019) onderzochten verschillende oplossingen en besloten dat *Istio* het beste platform was gezien zijn hoge graad van aanpasbaarheid.
- **Linkerd**
- **Consul**
- **Kuma**
- **Maesh**
- **Tetrate**

Deployment

Het individueel *deployen* van microservices is een positieve karaktereigenschap van de microservice-architectuur. Tijdens het *redployment* van een service kan dit leiden tot een tijdelijke onbeschikbaarheid van de service voor de eindgebruiker.

Er kan voor gekozen worden om gelijktijdig meerdere versies van de desbetreffende service te *deployen*. Hierbij blijft de huidige versie van de service online, gelijktijdig met de nieuwe versie van de service. Hierdoor kan de rest van de applicatie migreren naar gebruik van de nieuwe versie van de service. Dit resulteert in een hogere beschikbaarheid doordat de *downtijd* van de service geminimaliseerd wordt.

Echter kunnen *cloud* platformen (zie sectie 2.2.2 - Uitdaging *cloud computing*) oplossingen

bieden voor meer automatisatie, continue beschikbaarheid en een hogere fouttolerantie.

Daarnaast zijn er volgens Newman (2015) twee belangrijke concepten wat betreft deployment van microservices: *Continuous Integration* en *Continuous Delivery*. Hierbij worden *build* en *test* processen geautomatiseerd waarbij beschikbaarheid en productieklare artefacten (*builds*) gegarandeerd worden.

Continuous Integration (CI) Het doel van *Continuous Integration* is om alle teamleden gesynchroniseerd te houden. Wanneer code gecommit wordt, gaat een *CI*-server controleren of deze code geïntegreerd kan worden met de rest van de *codebase*. De code wordt gecompileerd door de server en er worden automatische tests uitgevoerd. Wanneer alle testen geslaagd zijn, wordt er een productieklare artefact gebuild. Enkele voordelen van *CI*:

- Snelle feedback over de kwaliteit van de code.
- Creatie van binaire artefacten (*builds*) kunnen geautomatiseerd worden.
- De code is onder versiebeheer.

Hierbij moet men rekening houden met hoe de *codebase* wordt bijgehouden. Alles in één *repository* met één *CI-build* loopt niet samen met deze architectuur omdat microservices onafhankelijk gedeployed moeten kunnen worden. Een variant hierbij is dat er één *repository* is waarbij de microservices op verschillende *branches* staan van de *repository*. Hierbij kan men dan *CI-builds* specificeren per *branche* wat resulteert in individuele *builds* van de microservices. Een extra variant hierop is dat voor elke microservice een aparte *repository* voorzien wordt met elk zijn eigen *CI-build*.

Continuous Delivery (CD) Het doel van *Continuous Delivery* is het definiëren van het pad naar productie. In een *build* zijn verschillende stages, met elk hun eigen tests. Het is aangewezen om één pipeline te voorzien voor elke microservice zodat ook hier de microservices individueel getest kunnen worden. Meer over testen in sectie 2.2.2 (Uitdaging testen).

Load balancing

Volgens Newman (2015) is het aan te raden, indien gebruik gemaakt wordt van meerdere hosts voor een microservice, om de instanties van de microservice achter een *load balancer* te plaatsen. De *load balancer* verdeelt de *network calls* die hij ontvangt over de verschillende instanties van een microservice, gebaseerd op een algoritme. De *load balancer* heeft ook de verantwoordelijkheid om instanties te verwijderen indien deze niet goed meer functioneren en voegt ze terug toe wanneer ze terug operationeel kunnen zijn. Dit draagt op zich bij tot een hogere fouttolerantie.

Een *load balancer* kan ook gebruikt worden bij monolithische applicaties, waarbij inkomende *requests* verdeelt worden over de verschillende instanties van de volledige applicatie. Het verschil tussen het gebruik van een *load balancer* bij zowel een monolithische applicatie als bij microservices, is volgens Dragoni e.a. (2018) dat een *load balancer* bij

microservices zowel externe als interne *requests* afhandelt. Een *load balancer* bij een monolithische applicatie verdeelt enkel externe *requests*.

Er kan volgens Balalaie e.a. (2018) een onderscheid gemaakt worden tussen een externe en een interne *load balancer*. Wanneer een microservice wil communiceren met een andere microservice, dan haalt de interne *load balancer* een lijst op van beschikbare instanties van de gewenste microservice op bij een *service registry* (zie sectie 2.2.2 - Uitdaging *Service Discovery*). De interne *load balancer* zal dan de interne *requests* verdelen over de beschikbare instanties uit de lijst gebaseerd op *metrics*. Een externe *load balancer* zal een lijst ophalen van de verschillende instanties van de *service registry* en de externe *requests* verdelen over de verschillende instanties van de *service registry*.

Ribbon is een tool die toelaat om *load balancing* regels op te stellen, die ook kan gebruikt worden met *Eureka*. *Eureka* is een *service registry* tool.

Bij het gebruik van een *load balancer* is het aangewezen, wanneer men de gezondheid van het systeem wil opvolgen (zie sectie 2.2.2 - Uitdaging monitoring), men niet enkel de gezondheid van de microservices individueel moet opvolgen maar ook de *load balancer*, gezien dit een *single point of failure* vormt.

Echter kunnen ook hier *cloud* platformen (zie 2.2.2 - Uitdaging *cloud computing*) geautomatiseerde oplossingen bieden.

Service discovery

Wanneer men wil weten welke microservices er beschikbaar zijn of wanneer men wil weten welke beschikbare *API's* er zijn, dan is het volgens Newman (2015) aangeraden om *service discovery* te voorzien in de vorm van een *service registry*. *Service discovery* laat instanties van een microservice toe zich te registreren bij de *service registry* waardoor een instantie makkelijk terug te vinden is in het systeem en dus dynamisch gelokaliseerd kunnen worden (Balalaie e.a., 2018). Een instantie van een microservice wordt verwijderd wanneer er geen periodieke hartslag meer gestuurd wordt door de desbetreffende microservice of wanneer de microservice afgesloten wordt en dit signaleert aan de *service registry*. Zie figuur 2.3 voor een voorbeeld van microservices met gebruik van een *service registry*. Deze manier van dynamische registratie van microservices is een goede optie wanneer men een zeer dynamische omgeving heeft, waarbij de totaliteit van microservices vaak onderhevig is aan wijzigingen. Wanneer men eerder een statische omgeving heeft, waarbij er weinig of geen microservices toegevoegd of verwijderd worden, kan men ook gebruik maken van *Domain Name Servers (DNS)*. *DNS* laat ons toe om een bepaalde domeinnaam te associëren met een bepaald IP-adres. Hierbij kunnen we een bepaalde microservice onder een bepaalde domeinnaam registreren zodat deze op deze manier makkelijk teruggevonden kan worden (bijvoorbeeld: *accounts.app.com*, *users.app.com*). Wanneer men *DNS*-namen vaak moet veranderen is het toch eerder aan te raden om gebruik te maken van een centrale *service registry*.

Alshuqayran e.a. (2016) voegt er aan toe dat voor *service discovery/registry* een bepaalde strategie moet voorzien worden zodat dit op een uniforme manier gebeurt voor alle microservices.

Er zijn verschillende tools beschikbaar omtrent *service discovery*: *Zookeeper*, *Consul*, *Eureka*,... . Echter zijn oplossingen omtrent *Service Discovery* vaak inbegrepen in *cloud* platformen (zie sectie 2.2.2 - Uitdaging *cloud computing*).

Datamanagement

Men kan in eerste instantie denken aan een gedeelde database, waarbij de verschillende microservices gebruik maken van één database. Echter sluit het idee van een *shared database* niet aan bij de onafhankelijkheid van de verschillende microservices. Het zorgt er onder meer voor dat er geen *loose coupling* mogelijk is doordat de microservices afhankelijk zijn van de gebruikte technologie van de *shared database*. Als men later de technologie van de database wil veranderen (bijvoorbeeld van relationele naar niet-relationele database) moeten de microservices hiervoor ook aangepast worden. Daarnaast zullen er ook meer regressietesten nodig zijn.

Volgens Newman (2015) is het aangewezen om voor elke microservice een eigen database te voorzien om tegemoet te komen aan de concepten van *loose coupling* en *high cohesion* van de gepersisteerde data. Er moet dus vermeden worden dat services zelf de nodige informatie uit andere tabellen (van andere microservices) gaan halen. Een oplossing hierbij is om de data ter beschikking te stellen via *API's*. De onderliggende relaties tussen de data worden behandeld door de services en bevinden zich dus niet meer op database niveau.

SchemeSpy is alvast een tool die kan helpen om de database te documenteren.

Omtrent datamanagement zijn er volgens Newman (2015) nog enkele aspecten die mee in rekening gebracht kunnen worden bij de implementatie van microservices:

Shared data In sommige gevallen maken verschillende microservices gebruik van *shared data*. Een voorbeeld hiervan is *shared static data*. Dit is statische data waar geen (directe) wijzigingen in verwacht worden, zoals bijvoorbeeld landcodes. Dit soort van statische data kan ter beschikking gesteld worden voor de microservices door voor elke microservice de statische data in hun eigen database of in hun eigen code te voorzien. Echter kunnen beide voorzieningen van statische data voor eventuele inconsistentie zorgen. Mocht er toch ooit een wijziging nodig zijn, dan moet de statische data in alle database-tabellen of microservice code aangepast worden. De meest optimale oplossing voor het voorzien van statische data is deze data te voorzien via een aparte service. Hierdoor kan de nodige statische data door eender welke service opgevraagd worden aan de hand van de microservice architectuur.

Transacties Transacties zorgen ervoor dat als iets fout gaat bij een database-operatie, dat er een *rollback* kan gebeuren. Een *rollback* zorgt ervoor dat database-operaties *alles of niets* zijn waardoor de database consistent blijft. Ofwel wordt de database-operatie succesvol afgewerkt zonder problemen, ofwel treedt er een probleem op (bijvoorbeeld wegvallen van de internetverbinding) waardoor de database-operatie niet uitgevoerd wordt en eerdere database wijzigingen van dezelfde transactie terug hersteld worden.

Bij microservices is dit echter een uitdaging. Als twee services, elk verantwoordelijk voor hun eigen domein met hun eigen database, een bepaalde functionaliteit moeten uitvoeren waarbij verschillende database-tabellen moeten worden aangepast, hoe kunnen we transactie-principe zo goed als mogelijk proberen te integreren? Hiervoor zijn enkele mogelijkheden:

- **Try again later:** Hierbij wordt de mislukte database-operatie bijgehouden in een

logbestand om later opnieuw uit te voeren. Omdat hier geen garantie is op volledige consistentie, kan men dit beschouwen als eventuele consistentie (Eng.: *eventual consistency*).

- **Abort operation:** Wanneer een database-operatie niet lukt, kan men een compenserende transactie (Eng.: *compensating transaction*) opzetten. Hiermee kunnen de database-wijzigingen, die wel geslaagd zijn, terug ongedaan gemaakt worden. Het toevoegen van een lijn in de database wordt ongedaan gemaakt door de lijn in de database terug te verwijderen. Ook hier kan dit beschouwd worden als *eventual consistency* omdat er geen garantie is dat de *compensating transaction* ook effectief zal slagen.
- **Distributed transactions:** De transacties binnen de applicatie worden behandeld door een *transaction manager*. Deze heeft de verantwoordelijkheid om ervoor te zorgen dat de database consistent blijft. Meestal wordt hier gebruik gemaakt van de *two-phase commit*. In de eerste fase moet elke service stemmen om zijn lokale transactie te laten doorgaan. Als de *transaction manager* van alle services een *yes* heeft gekregen in de stemronde, worden hun commits uitgevoerd. Wanneer er één *no* is uit de stemronde, dan wordt er een *rollback* uitgevoerd op eerdere commits. Deze oplossing probeert het meeste aan te sluiten bij het modulair karakter van microservices en database-consistentie, maar heeft ook zijn eigen uitdagingen. Wat als de centrale *transaction manager* zelf (tijdelijk) niet beschikbaar is (*single point of failure*)? Wat als een service niet kan antwoorden (stemmen) in de stemronde? Wat als een commit niet slaagt na het stemmen?

Reporting Vele organisaties willen rapporten trekken uit de gepersteeerde data. Als de data verspreid zit over verschillende databases en services, dan kan dit een uitdaging zijn om deze data gezamenlijk te structureren voor het opstellen van rapporten. Enkele mogelijke oplossingen voor het opstellen van rapporten aan de hand van data verspreid over verschillende microservices:

- **Data ophalen via API's:** De data die nodig is voor het opstellen van rapporten kan opgehaald worden aan de hand van API's. Dit kan echter een trage operatie zijn indien er veel data voorhanden is. Daarbij komt dat de API's van de microservices eventueel niet bedoeld kunnen zijn om er rapporten mee samen te stellen. Sommige API's kunnen teveel informatie teruggeven die onnodig zijn voor het opstellen van de rapporten. Andere API's voorzien misschien te weinig informatie.
- **Data pumps:** In plaats van alle data op te halen (*pull*), wat een trage operatie kan zijn, kan de data door de verschillende services periodiek gestuurd worden (*push*) naar de *reporting service*. Echter zorgt deze oplossing voor extra network calls, alsook aangepaste API's voor het leveren van rapporteringsdata.
- **Event data pump:** Deze variant op de *data pump* zal enkel data versturen naar de *reporting service* bij bepaalde *events*. Hiervoor communiceren de microservices met de *reporting service* aan de hand van een *event-based* patroon. Meer hierover in sectie 2.2.2 - Uitdaging communicatie.
- **Backup data pump:** Hiervoor wordt een *back-up* database gebruikt waaruit de rapporteringsdata uitgehaald wordt. Hierdoor blijft de operationele database vrij van

verkeer voor rapporteringsdoeleinden.

Monitoring

Vaak willen we een systeem gaan opvolgen aan de hand van *monitoring*, waarbij onder andere *logging* en *metrics* (CPU-gebruik, RAM-gebruik,...) belangrijke elementen zijn. *Monitoring* laat ons enerzijds toe om de gezondheid van een systeem van microservices op te volgen en anderzijds om een inzicht te krijgen hoe een systeem gebruikt wordt door de eindgebruikers.

Bij een monolithische applicatie is de gezondheid van de applicatie makkelijker op te volgen aangezien er slechts één applicatie dient gemonitored te worden. Alsook zijn fouten makkelijker op te sporen binnenin één applicatie. Bij een applicatie gebaseerd op de microservice architectuur zijn fouten moeilijker op te sporen doordat het systeem bestaat uit verschillende microservices, meestal verspreid over verschillende *hosts*. De uitdaging wat betreft *monitoring* van dergelijke applicaties of services bestaat uit het centraliseren van monitoring-data van de verschillende services. Balalaie e.a. (2018) raadt hierbij aan om gebruik te maken van een centrale *monitoring-service*. Dit sluit enerzijds aan bij het microservice principe, waarbij de *monitoring-service* de verantwoordelijkheid (*Single Responsibility Principle* zie sectie 2.2) heeft om monitoring-data te verzamelen en weer te geven. Anderzijds zorgt een monitoring-service ervoor dat centrale monitoring van de verschillende services mogelijk wordt.

Er moet ook rekening gehouden worden met het feit dat één bepaalde microservice op verschillende servers kan gehost worden. Volgens Newman (2015) is het bij dergelijke opstelling belangrijk om ook de gezondheid van alle *hosts* goed op te volgen. Bepaalde monitoring-data kan aangeven dat het probleem aan de microservice ligt of eerder aan de *hosts*. Zo kan bijvoorbeeld een verhoogd CPU-verbruik dat voorkomt op alle *hosts* van een microservice een aanwijzing zijn dat er effectief een probleem is met de microservice dewelke het verhoogd CPU-verbruik veroorzaakt. Is er daarentegen enkel een verhoogd CPU-verbruik bij één enkele host, dan geeft dit aan dat het probleem eerder bij de host kan liggen. Daarnaast is het ook aangewezen om bij het gebruik van een *load balancer* (zie sectie 2.2.2 - Uitdaging *load balancer*), de *load balancer* zelf ook te gaan monitoren, gezien dit ook een *single point of failure* vormt.

Newman (2015) voegt er nog aan toe dat het belangrijk is om zelf *service metrics* te voorzien per microservice omdat deze informatie niet enkel belangrijk is om de gezondheid van een microservice op te volgen. Bepaalde *metrics* kunnen een bepaalde business waarde hebben waardoor ze een belangrijk inzicht kunnen geven over hoe het systeem gebruikt wordt door de eindgebruikers, hoe vaak bepaalde services (al dan niet) gebruikt worden,... . Zo kan het voor een organisatie belangrijk zijn om te weten hoeveel keer een gebruiker zijn vorige orders bijvoorbeeld bekijkt, hoe vaak er ingelogd wordt,... .

Bij een microservice applicatie gebeuren meestal verschillende *asynchrone calls* tussen de

verschillende services, voor het leveren van een bepaalde functionaliteit. Dit zorgt ervoor dat het opsporen van een fout ook hier moeilijker kan verlopen. Volgens Newman (2015) bestaat de oplossing eruit om gebruik te maken van *correlation IDs* waarmee de ketting van netwerk-calls tussen de verschillende services kan opgevolgd worden. Men kan dit vergelijken met een *stack trace*, waarmee men de locatie van een probleem kan terugvinden. Dit principe houdt in dat bij een reeks van netwerk-calls tussen verschillende microservices, bij de eerste netwerk-call een *Globally Unique Identifier (GUID)* toegevoegd wordt en deze doorgestuurd wordt naar de volgende microservice in de ketting van asynchrone netwerk-calls. Als er in een bepaalde service een probleem optreedt kan aan de hand van de *GUID* snel gedetecteerd worden waar, in de ketting van netwerk-calls, het probleem opgetreden is.

Er zijn heel wat tools beschikbaar inzake monitoring: *Collectd*, *Logstash* (logging), *ElasticSearch*, *Kibana*, *CodaHale* (Java metrics), *Nagios*, *Graphite*, *Zipkin* (tracing), *Dapper* (tracing), *ModSecurity*, Ook hier bieden *cloud* platformen (zie sectie 2.2.2 - Uitdaging *cloud computing*) reeds uitstekende oplossingen om monitoring te voorzien op applicaties, eventueel verspreid op een cluster.

Security

In een monolithische applicatie zit alle functionaliteit inzake authenticatie en autorisatie in de applicatie zelf. In een applicatie bestaande uit microservices kan dit echter een uitdaging zijn waarbij er vaak communicatie is tussen de verschillende microservices onderling. Newman (2015) raadt echter aan om een *Single Sign-On (SSO)* implementatie te voorzien, in de vorm van een *Single Sign-On Gateway*. Deze *gateway* vormt een centraal toegangspunt voor externe *requests* waardoor de microservices afgeschermd worden van de buitenwereld. Zie figuur 2.3 voor een voorbeeld van een microservices met gebruik van een *gateway*. Hierdoor kunnen zowel autorisatie als authenticatie via dit centraal punt afgehandeld worden. De *gateway* zal zelf instaan voor het authentifieren en autoriseren. Wanneer de gebruiker aangemeld is, zal de *gateway* de *request* doorsturen naar de desbetreffende microservice, waarbij de autorisatie en authenticatie gegevens toegevoegd kunnen worden aan de *headers* van de *requests*. Deze microservices kunnen op zich dan intern beslissen of de request van de gebruiker voldoet aan de beveiligingsstandaard van de microservice zelf.

Het is in ieder geval ook aan te raden om gebeurtenissen binnenin het systeem te gaan opvolgen en documenteren (zie sectie 2.2.2 - Uitdaging monitoring), waarbij verdacht gedrag of verdachte activiteiten sneller opgespoord kunnen worden. Daarnaast kunnen microservices, net als een monolithische applicatie, ook via een *firewalls* beveiligd worden door enkel bepaalde IP-adressen toe te laten of bepaalde IP-adressen uit te sluiten.

Er zijn enkele tools beschikbaar die kunnen helpen omtrent security: *Shibboleth* (identity management), *ModSecurity*,

Testing

Doordat microservices onafhankelijk zijn, kunnen individuele *unit testen* per microservices volgens Newman (2015) makkelijker zijn door de afgebakende verantwoordelijkheden van de microservices. Echter houdt dit ook in dat een gedistribueerd systeem moeilijker te testen is. Wanneer een microservice getest wordt, samenwerkend met andere microservices, kan men hier gebruik maken van *mocking*. Door *mocking* kan men een microservice zo configureren om vastgelegde responses te sturen naar onze microservice die getest wordt. *MounteBank* is alvast een tool die hierbij kan helpen. De grootste uitdaging bij het testen van microservices zijn de *End-to-end* testen. Dit zijn testen uitgevoerd op het algemene systeem en leveren een hoge graad van vertrouwen die garandeert dat de geteste code ook zal werken in productie. Voor *End-to-End* testen moeten we alle microservices samen *deployen* en worden ze samen getest. Het is echter niet aangeraden om voor elke individuele microservice een *End-to-End* test te voorzien. Indien men gebruik maakt van automatische *build-pipelines* (zie sectie 2.2.1), voorziet men dus beter geen *End-to-End* test op het einde van elke individuele *build-pipeline* van elke microservice. Men kan beter alle *build-pipelines* laten samenkomen in één *End-to-End* test *stage*. Hierdoor voorkomt men alvast dat dezelfde functionaliteiten meerdere keren getest wordt door de verschillende test-stages van alle microservices *build-pipelines* (Newman, 2015).

Volgens Newman (2015) is er een alternatief voor *End-to-End*-tests: *Consumer-Driven tests*. Het uiteindelijke doel is te verzekeren dat wanneer we een nieuwe microservice *deployen* in productie, dat onze wijzigingen geen problemen veroorzaken in andere microservices die afhankelijk zijn van de gewijzigde microservice. Hierbij kunnen we gebruik maken van *consumer driven contracts* waar de verwachtingen van een afhankelijke microservice (de *consumer*) worden gedefinieerd van een bepaalde microservice waarvan hij afhankelijk is (de *producer*). Deze *consumer-driven contracts* worden best toegevoegd aan de *CI-build pipeline* van de *producer*.

De meeste testen gebeuren vooraleer de microservice(s) in productie gaan. Newman (2015) voegt hieraan toe dat er enkele mogelijkheden zijn wanneer microservices in productie gaan en getest moeten worden:

- ***Smoke test suite***: dit zijn testen die ontworpen zijn om microservices die juist *gedeployed* zijn te testen of het deployment gelukt is.
- ***Blue/Green deployment***: Hierbij worden twee versies van de microservice samen *gedeployed*, waarbij slechts één versie de echte *requests* ontvangt. Wanneer de *smoke tests* voor de nieuwe versie van de microservice geslaagd zijn, kan de *production traffic* (netwerkverkeer naar de microservice in productie) omgeleid worden naar de nieuwe versie van de microservice. Dit zorgt er ook voor dat de *downtijd* van een microservice geminimaliseerd wordt.
- ***Canary releasing***: deze variant van *blue/green deployment* waarbij zowel functionele als niet-functionele vereisten getest worden. Beide versies van de microservice ontvangen de *production traffic* zodat beide versies met elkaar vergeleken kunnen worden (*metrics*,...).

Performantie

Wanneer bij een monolithische applicatie een bepaalde functionaliteit moet afgewerkt worden, worden alle operaties intern in de applicatie afgehandeld. Wanneer een identieke applicatie in de vorm van microservices dezelfde functionaliteit moet voldoen, is er communicatie nodig tussen de microservices onderling in de vorm van verschillende (interne) *network calls*. Dit kan leiden tot een verminderde gebruikerservaring voor de eindgebruiker, doordat de gebruiker mogelijk langer moet wachten op het resultaat van de functionaliteit. De interne communicatie tussen de microservices onderling zou beperkt kunnen worden door *data sharing* (zie sectie 2.2.2 - Uitdaging datamanagement).

Fouttolerantie

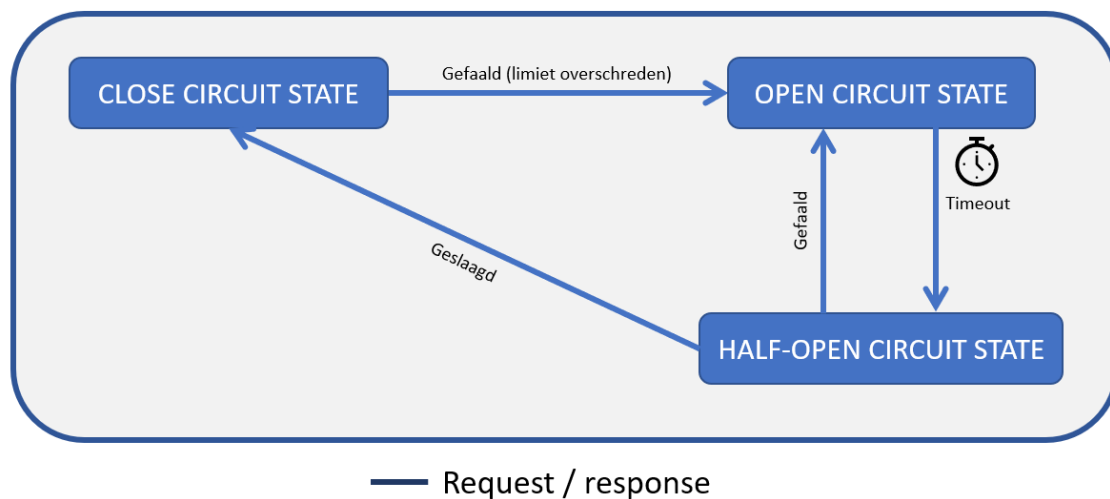
Microservices voorzien algemeen een hogere fouttolerantie, doordat de microservices individueel *gedeployed* worden. Een falende microservice heeft weinig of geen impact op de beschikbaarheid van andere microservices. Deze beschikbaarheid kan doorgaans gecontroleerd worden via een *service registry* (zie sectie 2.2.2 - Uitdaging *Service Discovery*). Maar het kan zijn dat een microservice zich succesvol geregistreerd heeft bij de *service registry*, maar uitvalt voor hij zijn periodieke hartslag gestuurd heeft naar de *service registry*. Om snelle feedback te hebben bij dergelijke situatie waar een microservice niet bereikbaar is, kan men kiezen om een *circuit breaker* te voorzien in de microservice (*client*) die de onbereikbare microservice (*provider*) wenst te bereiken. Deze *circuit breaker* kan drie mogelijke statussen hebben op basis van het opvolgen van recente *responses*:

- **Close circuit state:** Een provider is bereikbaar. *Requests* worden doorgestuurd naar de *provider*.
- **Open circuit state:** Een provider is niet bereikbaar. *Requests* worden niet doorgestuurd naar de *provider*. Deze status wordt bereikt wanneer een bepaalde limiet van gefaalde *responses* is overschreden.
- **Half-Open circuit state:** De *circuit breaker* zal na een bepaalde tijd (*timeout*) de *provider* terug proberen te contacteren. Bij slagen wordt de status van de *circuit breaker* veranderd naar *close circuit state*. Bij falen wordt de status opnieuw naar *open circuit state* veranderd.

Zie figuur 2.7 voor een voorstelling hoe de statussen veranderd worden bij een *circuit breaker*. Echter bieden *cloud* platformen (zie sectie 2.2.2 - Uitdaging *cloud computing*) automatische oplossingen om een hoge fouttolerantie en beschikbaarheid van de microservices te garanderen.

Containerization

Het gebruik van *containerization* is een gekend concept bij microservices. *Containers* dragen bij tot een hogere modulariteit en isolatie volgens Stubbs e.a. (2015). Ze zorgen ervoor dat microservices als individuele elementen makkelijk herbruikt en geschaald kunnen worden in een gedistribueerd systeem, vooral in combinatie met *cloud computing* (zie sectie 2.2.2 - Uitdaging *cloud computing*). *Containerization* is een mechanisme dat

Figuur 2.7: Verloop van statussen in een *circuit breaker*

virtualisatie voorziet op *operating system* niveau, waarbij het enkel geheugen gebruikt die het nodig heeft om zijn processen uit te voeren. Dit in tegenstelling tot *Virtual Machines* (VM).

Virtual Machines

Wanneer een applicatie in de *cloud* gebruikt moet worden, kan men deze applicatie in een *Virtual Machine* (VM) draaien. Deze doet zich voor als een fysieke computer waarop de applicatie draait. De VM emuleert de hardware van een computer waarop hij draait, inclusief een besturingssysteem en de uiteindelijke applicatie. Wanneer het verkeer op de applicatie stijgt, kunnen er meerdere VM instanties opgestart worden. Echter houdt dit in dat voor elke VM instantie heel wat rekenkracht nodig is voor de server waarop hij draait voor het emuleren van de hardware, het besturingssysteem en de applicatie (Aussems, 2018).

Containers

Containers bieden hier de oplossing. Een container bevat alles wat de applicatie nodig heeft om te draaien (code, *dependencies*,...). Het doel van deze containers is om geïsoleerde *units* te voorzien die kunnen gebruikt worden op eender welk platform. Containers onderscheiden zich van *Virtual Machines* doordat verschillende containers dezelfde besturingssysteem kernel delen. Hierdoor worden ze vaak gezien als de lichtere versies van VM's omdat bij deze containers geen besturingssysteem aanwezig is (Aussems, 2018). Dit heeft als gunstig effect dat ze kleiner zijn in grootte en dus ook sneller opgestart kunnen worden (IBM, 2020). Er zijn enkele interessante platformen die voorzien in *containerization*: *Docker*, *OpenVZ*, *Rocket*,... (Amaral e.a., 2015). Aangezien *Docker* het meeste wordt geassocieerd met de microservice-architectuur in de vakliteratuur en als standaard in de industrie omtrent *containerization* wordt beschreven door IBM (2020), wordt verder ingegaan op *Docker* in de volgende paragraaf. Echter geldt hetzelfde prin-

cipe van *containerization* voor elk beschikbaar platform. Het uiteindelijke resultaat van *containerization* bestaat uit een *containerized* applicatie die op eender welke machine kan uitgevoerd worden. Door het efficiënt gebruik van hardware-resources en hun makkelijke schaalbaarheid (kleinere grootte ten opzichte van VM's) is *containerization* een belangrijk concept bij microservices.

Docker

Docker is een platform dat gebruikt wordt om applicaties in *containers* te voorzien. Het zorgt ervoor dat alles wat nodig is om een applicatie te gebruiken (code, *dependencies*, ...), gebundeld wordt tot één bestand, nl. een *Docker image* (Stubbs e.a., 2015). Deze *image* zorgt er ook voor dat het op éénder welke machine kan gebruikt worden waar *Docker* is op geïnstalleerd of bij voorbaat in de *cloud* (zie sectie 2.2.2 - Uitdaging *cloud computing*). Een *Dockerfile* wordt gebruikt voor de configuratie voor het aanmaken, gebruiken en delen van containers. De applicatie (container) kan lokaal gebruikt worden of men kan deze ook delen op *Docker Hub* of andere *cloud* platformen waardoor de applicatie publiek gebruikt kan worden in de *cloud* (Docker, 2020).

Cloud computing

Volgens Toffetti e.a. (2015) houdt *cloud computing* in dat applicaties *continuously* beheerd kunnen worden. Dit houdt in dat enerzijds hun *resources* aangepast kunnen worden op basis van het inkomende verkeer en anderzijds veerkrachtig zijn bij falen door het aanmaken of het heropstarten van instanties. Het *continuously management* van dergelijke applicaties of microservices wordt gerealiseerd door het monitoren van de applicatie of microservice (zie sectie 2.2.2 - Uitdaging monitoring) en voorziet automatische reacties bij falen (*health management*) en/of een veranderende omgeving (*auto-scaling*) waarbij menselijke tussenkomst geminimaliseerd wordt.

De voordelen van *cloud computing* zijn:

- **Operationeel:** Algemeen wordt er gesteld dat er een verhoogde productiviteit is. Deze verhoogde productiviteit wordt gerealiseerd door:
 - Het automatisch toewijzen van resources.
 - Verminderde wachttijd voor het voorzien van verschillende omgevingen (*development, test, production*).
 - Verhoogde flexibiliteit en verminderde *time-to-market* bij *business* veranderingen
- **Economisch:** Er wordt bij *cloud-computing* voornamelijk gebruik gemaakt van het *pay-per-use* model, waarbij men enkel betaalt voor hetgeen men effectief verbruikt. Dit heeft volgende voordelen:
 - Geen investeringen in eigen IT infrastructuur nodig, inclusief het onderhoud ervan.
 - Door het verschuiven van de verantwoordelijkheden inzake de fysieke IT infrastructuur naar een extern bedrijf, kan het ontwikkelbedrijf zich ten volle concentreren op de ontwikkeling van de applicatie en/of microservices. Ook zijn hierdoor geen extra kosten nodig voor onderhoudspersoneel of dergelijke

voor IT infrastructuur.

Om de voordelen van *cloud computing* te kunnen genieten, alsook de bijhorende functionele voordelen zoals *monitoring*, *health management* en *auto-scaling*, kan er gebruik gemaakt worden van diensten van externe bedrijven inzake IT infrastructuur. Door het gebruik van dergelijke diensten bevindt de logica inzake *continuous management* geïsoleerd van de applicatie zelf. Volgens Stubbs e.a. (2015) zijn er verschillende platformen die dergelijke *cloud computing* infrastructuur aanbieden: *CoreOS*, *Mesos/Mesosphere*, *OpenShift*, *CloudFoundry*, *Kubernetes*, *Brooklyn/Clocker*, *Shipyards*, *Crane*,.... Bij dergelijke platformen wordt gebruik gemaakt van *containerization* (zie sectie 2.2.2 - Uitdaging containerization). Aangezien *Kubernetes* door verschillende bronnen uit de vakliteratuur omschreven wordt als de standaard binnen de industrie wat betreft gedistribueerde microservicesystemen, wordt hier verder op ingegaan in de volgende paragraaf.

Kubernetes

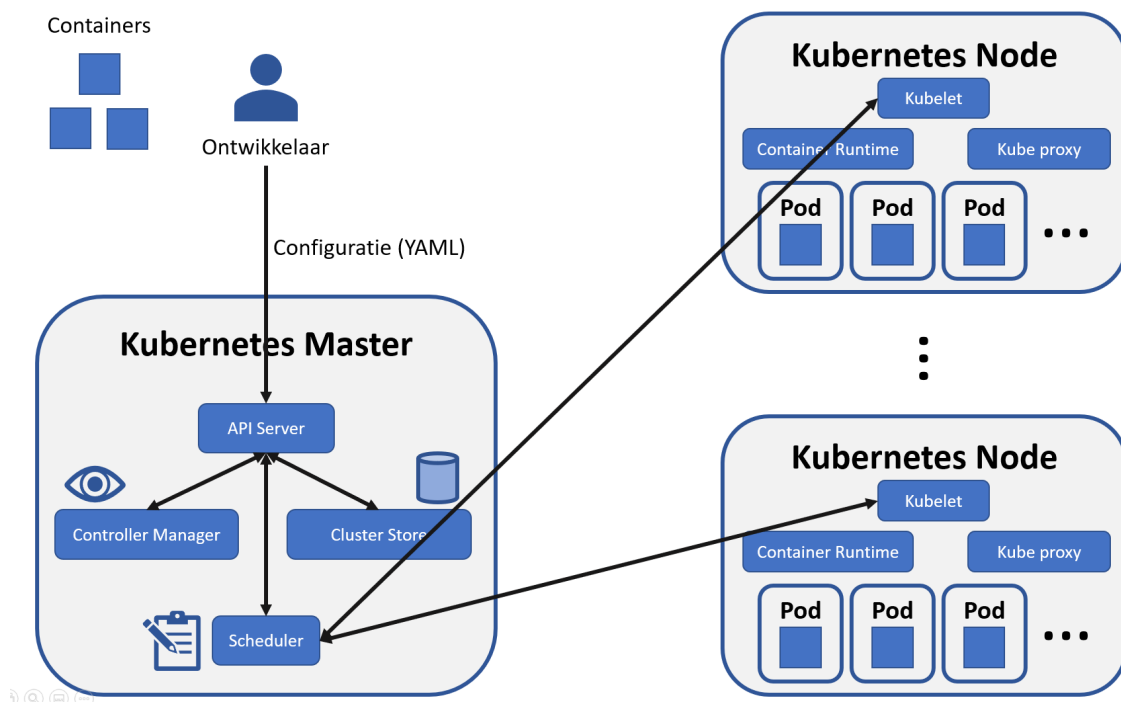
Volgens Victor e.a. (2015) is *Kubernetes* een zeer goede *open-source* cluster container manager, ontworpen door *Google*. *Kubernetes* beheert, *deployed* en schaaft groepen van containers (zie sectie 2.2.2 - Uitdaging containerization) naar clusters van machines. Dergelijke systemen zijn ontworpen om de nodige primitieven te voorzien voor microservice-architecturen. Daarnaast kan *Kubernetes* als een orchestrator (zie sectie 2.2.2 - Uitdaging communicatie) gezien worden bij *cloud* microservices applicaties (Poulton & Joglekar, 2019).

Kubernetes omvat volgende concepten (Kubernetes, 2020):

- **Definiëren van een 'gewenste staat':** Er kan een 'gewenste staat' geconfigureerd worden. Deze kan onder meer specificeren welke *container images* gebruikt moeten worden, het aantal *replica's* van een container, welke *resources* er gebruikt moeten worden,... . *Kubernetes* zal altijd trachten om deze 'gewenste staat' te garanderen. Deze configuratie wordt doorgaans in een *.yaml* bestand voorzien en doorgestuurd naar de *Kubernetes Master*.
- **Kubernetes Objecten:** *Kubernetes* voorziet enkele abstracties die de staat van een systeem weergeven: *deployed containerized applicaties*, *netwerk en schijf-resources* en andere informatie omtrent de cluster. Deze abstracties worden weergegeven aan de hand van volgende (basis) *Kubernetes objecten*:
 - **Pod:** Een *Pod* staat voor de basis-unit voor het uitvoeren van een *Kubernetes* applicatie. Een *Pod* bevat: één of meerdere containers, opslag-*resources*, een uniek netwerk identiteit (IP adres), alsook verschillende opties hoe containers moeten gebruikt worden binnen de cluster. Alle containers in één *Pod* delen ook dezelfde *resources*. Een *Pod* resulteert in één draaiende instantie van een bepaalde applicatie. Bij *auto-scaling* in *Kubernetes*, worden extra instanties voorzien van een applicatie door replicatie, waardoor extra *Pods* aangemaakt worden in de cluster volgens de configuratie in de 'gewenste staat' van de cluster.
 - **Service:** Een *Kubernetes Service* is een abstractie die een logische groep van verschillende *Pods* definieert, alsook hoe deze bereikt kunnen worden.

- **Namespace:** *Namespaces* kunnen gebruikt worden om de fysieke cluster verder in te delen in virtuele clusters.
- **Kubernetes Master:** In een cluster is steeds een *master* aanwezig die verantwoordelijk is voor verschillende processen zoals het configureren en valideren van objecten in de cluster, het garanderen van de geconfigureerde gewenste staat, monitoring, implementeren van veranderingen,... . Een *master* is dus volledig verantwoordelijk voor het management van de cluster en bevat volgende systeemp processen (Poulton & Joglekar, 2019):
 - **API server:** Alle communicatie, zowel intern als extern, verloopt via de *API server*. Als ontwikkelaar, kunnen wij onze *.yaml* configuratie doorsturen (*HTTPS POST*) doorsturen naar de *master*. De *API server* valideert deze configuratie, persisteert deze in de *cluster store* en zorgt dat deze uitgevoerd wordt op de cluster. Daarnaast is dit ook het enige component in *Kubernetes* waarmee we rechtstreeks interageren.
 - **Cluster store:** Deze service is verantwoordelijk voor het persisteren van de configuratie van de 'gewenste staat'.
 - **Controller manager:** Deze manager voert verschillende onafhankelijke controles uit op periodieke basis waarmee het de cluster observeert en reageert op bepaalde events om zo de 'gewenste staat' te kunnen blijven garanderen.
 - **Scheduler:** Deze service is verantwoordelijk voor het verdelen van nieuwe taken over de beschikbare en gezonde *nodes*.
- **Kubernetes Nodes:** Dit zijn de machines (*Virtual machines*, fysieke servers,...) waar de applicaties op draaien. Deze *Kubernetes Nodes* worden rechtstreeks beheerd door de *Kubernetes Master*. Deze *nodes* observeren de *API server* voor nieuwe taken. Ook communiceren ze met de *master* om zo feedback te geven over bepaalde operaties. Een *node* bevat volgende basiselementen (Poulton & Joglekar, 2019):
 - **Kubelet:** Elke *node* heeft een *kubelet*. Een *kubelet* is verantwoordelijk voor de registratie van een *node* bij een cluster. Daarnaast is deze service ook verantwoordelijk voor het observeren van taken beheerd door de *master* en geeft het feedback aan de *master* (bij falen,...). Wanneer een *kubelet* een taak niet kan uitvoeren, informeert het enkel de *master*. De *master* is op zijn beurt verantwoordelijk voor het toewijzen van de taak aan een nieuwe *node*.
 - **Container runtime:** Een *kubelet* heeft ook een *container runtime* nodig om container gerelateerde taken uit te voeren zoals: het opstarten van een container, het stoppen van een container,... . Deze maakt het ook mogelijk om in *Kubernetes* met containers te werken zoals *Docker*.
 - **Kube-proxy:** Deze service draait op elke *node* in de cluster en is verantwoordelijk voor netwerk configuratie. Het zorgt ervoor dat elke *node* zijn uniek IP adres krijgt, dat er load-balancing mogelijk is tussen de verschillende *pods* in de *node*,... .
- **Kubernetes DNS:** Elke *Kubernetes* cluster heeft een interne *Domain Name System*. Deze *DNS* service heeft zelf een statisch IP adres en zit gecodeerd in elke *pod* in de cluster, wat betekent dat alle containers en *pods* weten hoe de *DNS* service te bereiken. Daarnaast wordt elke nieuwe service automatisch geregistreerd via deze cluster *DNS* service, waardoor elke service kan teruggevonden worden door naam.

Om de vernoemde elementen duidelijker te maken, is er alvast een visuele weergave van de *Kubernetes*-architectuur weergegeven in figuur 2.8. Het voordeel aan *cloud-computing*, in combinatie met *containerization*, is dat veel van de vernoemde *Kubernetes*-elementen automatisch voorzien en beheerd worden door *Kubernetes* zelf. Voor ontwikkelaars volstaat het om enkel *containerized* applicaties te voorzien en de 'gewenste staat' van de cluster (in de vorm van een *.yaml* bestand of via een *user interface*) te versturen naar de *Kubernetes Master - API server*. Verder zijn *health management*, *auto-scaling*, *monitoring*,... services die automatisch beheerd worden door *Kubernetes* waardoor een hoge beschikbaarheid en fouttolerantie gegarandeerd worden (Poulton & Joglekar, 2019).



Figuur 2.8: Kubernetes architectuur

Kubernetes voorziet hierdoor volgende functionaliteiten:

- **Load Balancer** (zie sectie 2.2.2 - Uitdaging load balancing)
- **Automatisch beheer van de containers (cluster)**
- **Logging en monitoring van de containers** (zie sectie 2.2.2 - Uitdaging monitoring)
- **Auto-scaling** (zie sectie 2.2.1 - Voordeel schaalbaarheid)
- **Rolling updates:** Wanneer een applicatie is gewijzigd, worden de oude containers incrementeel vervangen door de nieuwe. Dit resulteert in minder *downtijd*.

Verskillende *cloud*-bedrijven bieden een platform aan gebaseerd op *Kubernetes*:

- **Google Kubernetes Engine**
- **Azure Kubernetes Service**
- **Amazon Elastic Kubernetes Service**
- **IBM Cloud Kubernetes Service**
- **CoreOS**

- **OpenShift Kubernetes Engine**
- **CloudFoundry**
- **Apache Brooklyn**
- **Clocker**

Daarnaast zijn er nog andere *cloud*-platformen beschikbaar:

- **Docker Swarm**
- **Apache Mesos**
- **Batch Shipyard (Azure)**
- **GFS Crane Cloud**

3. Methodologie

3.1 Inleiding

Dit onderzoek kan ingedeeld worden in 2 grote delen:

- Theoretisch onderzoek
- Proof-of-concept

3.2 Theoretisch onderzoek

Het theoretisch onderzoek werd gedaan om antwoorden te bieden op de verschillende onderzoeksvragen om te voldoen aan de onderzoeksdoelstelling. Dit werd gedaan in verschillende fases:

1. Fase 1: Een algemeen beeld vormen van zowel de monolithische als de microservicearchitectuur
2. Fase 2: Onderzoeken van de voordelen en uitdagingen van de microservicearchitectuur
3. Fase 3: Onderzoeken van de tools en oplossingen voor de uitdagingen van de microservicearchitectuur
4. Fase 4: Onderzoek naar containerization en cloud computing

3.2.1 Fase 1

De doelstelling van deze fase was een algemeen beeld te vormen van zowel de monolithische als de microservicearchitectuur. Deze informatie werd verzameld uit verschillende academische naslagwerken, artikels, conference papers, Hiervoor werden volgende zoektermen vastgelegd: *microservice(s)*, *microservice architectuur/architecture*, *monolith*, *monolithisch/monolithic*, *monolithische architectuur/monolithic architecture*. De bronnen werden gevonden aan de hand van de zoektermen.

3.2.2 Fase 2

De doelstelling van deze fase was de voordelen en de uitdagingen van de microservicearchitectuur te onderzoeken. Hoewel uit Fase 1 reeds voldoende voordelen en uitdagingen teruggevonden werden, werden de zoektermen uit de voorgaande fase verder uitgebreid met volgende zoektermen: *voordelen/advantages*, *uitdagingen/challenges*. De bronnen werden gevonden aan de hand van de zoektermen en hun eventuele combinaties onderling (bv. *microservices challenges*).

3.2.3 Fase 3

De doelstelling van deze fase was de beschikbare tools en oplossingen voor de uitdagingen van de microservicearchitectuur te onderzoeken. Hoewel uit Fase 1 en Fase 2 reeds enkele tools en oplossingen teruggevonden werden, werden de zoektermen uit de voorgaande fase verder uitgebreid met volgende zoektermen: *service discovery*, *load balancing*, *communicatie*,... . De bronnen werden gevonden aan de hand van de zoektermen en hun eventuele combinaties onderling (bv. *microservice service discovery*).

3.2.4 Fase 4

De doelstelling van deze fase was om een beeld te schetsen van wat *containerization* en *cloud computing* inhouden in relatie tot de microservice-architectuur. Hiervoor werden de zoektermen, op basis van vernoemingen in andere bronnen, uitgebreid met onder meer: *containerization*, *Docker*, *containers*, *cloud*, *cloud computing*, *Kubernetes*,... . Ook hier werden verschillende bronnen gevonden aan de hand van de zoektermen en hun eventuele relaties onderling (bv.: *microservice cloud*). Daarnaast werd ook bepaalde technische documentaties van bepaalde platformen, die in de literatuurstudie werden aangewezen als standaard binnen de sector, doorlopen.

Als alle onderzoeksfases doorlopen zijn, werden de resultaten hiervan als leidraad gebruikt doorheen de ontwikkeling van de proof-of-concept applicatie.

3.3 Proof-of-concept

Voor het samenstellen van de *proof-of-concept* werden verschillende elementen uit het literatuuronderzoek opgenomen. Echter bestaan deze meer uit technische verwijzingen, concepten, Voor een meer praktische toepassing van deze elementen werd verder doorgezocht naar de technische documentatie van bepaalde platformen om zo een zo correct mogelijk toepassing te kunnen garanderen van bepaalde technieken.

4. Proof-of-concept

4.1 Inleiding

In dit hoofdstuk worden bepaalde technieken die kunnen helpen bij het implementeren van de microservice-architectuur toegepast op een applicatie.

4.2 Accounting applicatie

Als *proof-of-concept* wordt de *Accounting* applicatie gebruikt. De *Accounting* applicatie is een applicatie die volgende functionaliteiten biedt:

- het beheren van klanten
- het beheren van facturen
- het beheren van banktransacties

4.2.1 Beheren van klanten

In de *Accounting* applicatie kan men klanten toevoegen, wijzigen en verwijderen. Een client (*Client-model*) heeft volgende basisgegevens:

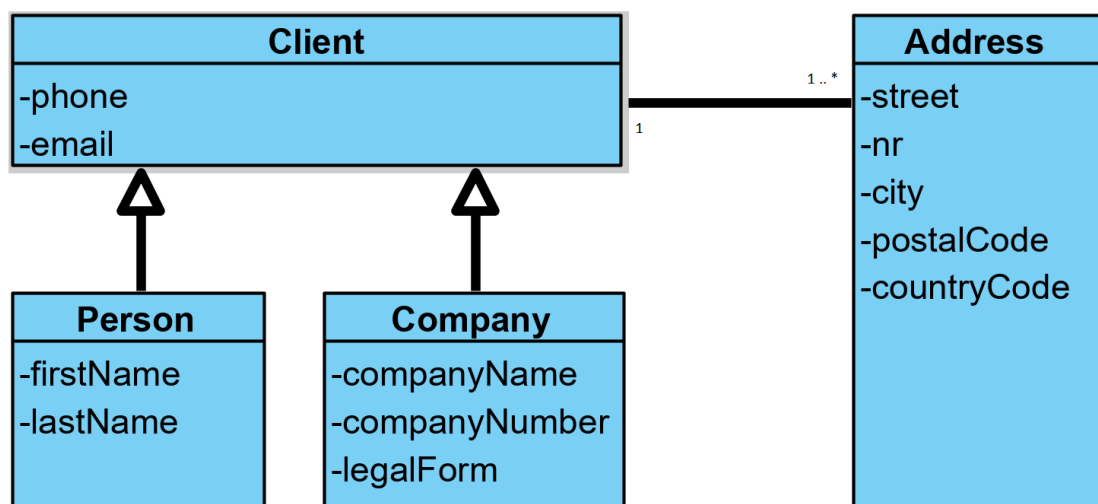
- telefoon (*phone-property*)
- email (*email-property*)
- adres (*address-property* - *Address-model*)
 - straat (*street-property*)

- nummer (*nr*-property)
- stad (*city*-property)
- postcode (*postalCode*-property)
- landcode (*countryCode*-property)

Een client kan daarnaast een persoon (*Person*-model) of een bedrijf (*Company*-model) zijn met elk volgende extra gegevens:

- *Person*:
 - voornaam (*firstName*-property)
 - achternaam (*lastName*-property)
- *Company*:
 - bedrijfsnaam (*companyName*-property)
 - bedrijfsnummer (*companyNumber*-property)
 - bedrijfsvorm (*legalForm*-property)

Zie figuur 4.1 voor een visuele representatie van het *Client*-model en bijhorend *Address*-model.



Figuur 4.1: Visuele representatie *Client*-model

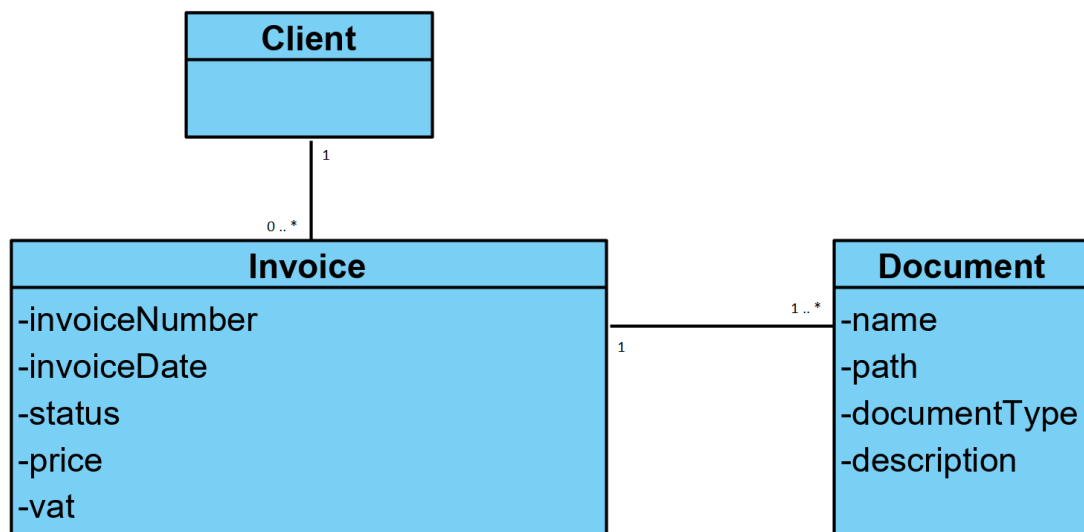
4.2.2 Beheren van facturen

In de *Accounting* applicatie kan men facturen toevoegen, wijzigen en verwijderen. Een factuur (*Invoice*-model) heeft volgende gegevens:

- referentie (*invoiceNumber*-property)
- datum (*invoiceDate*-property)
- client (*client*-property - *Client*-model (zie sectie 4.2.1))
- status (*status*-property)
- prijs (*price*-property)

- btw (*vat*-property)
- document (*document*-property *Document*-model)
 - naam (*name*-property)
 - pad (*path*-property)
 - type (*documentType*-property)
 - uitleg (*description*-property)

Zie figuur 4.2 voor een visuele representatie van het *Invoice*-model, bijhorend *Document*-model en het *Client*-model.



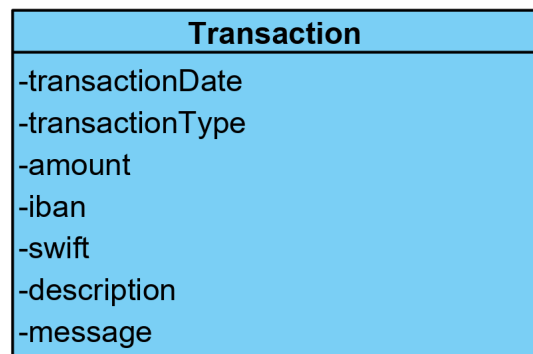
Figuur 4.2: Visuele representatie *Invoice*-model

4.2.3 Beheren van banktransacties

In de *Accounting* applicatie kan men transacties toevoegen, wijzigen en verwijderen. Een transactie (*Transaction*-model) heeft volgende gegevens:

- datum (*transactionDate*-property)
- type (*transactionType*-property)
- bedrag (*amount*-property)
- IBAN (*iban*-property)
- BIC (*swift*-property)
- btw (*vat*-property)
- uitleg (*description*-property)
- bericht (*message*-property)

Zie figuur 4.3 voor een visuele representatie van het *Transaction*-model.

Figuur 4.3: Visuele representatie *Transaction*-model

4.3 Microservice-architectuur

Het doel van deze *proof-of-concept* is het toepassen van de microservice-architectuur op de *Accounting* applicatie om op deze manier te kunnen genieten van de voordelen van deze architectuur: modulariteit, de vrije technologiekeuze per microservice en het individueel *deployen*/schalen van microservices. Hierbij zal een mogelijke implementatie van een ecosysteem van microservices worden aangetoond, waarbij getracht wordt de uitdagingen verbonden aan de microservice-architectuur te minimaliseren.

4.3.1 Bepalen van de microservices

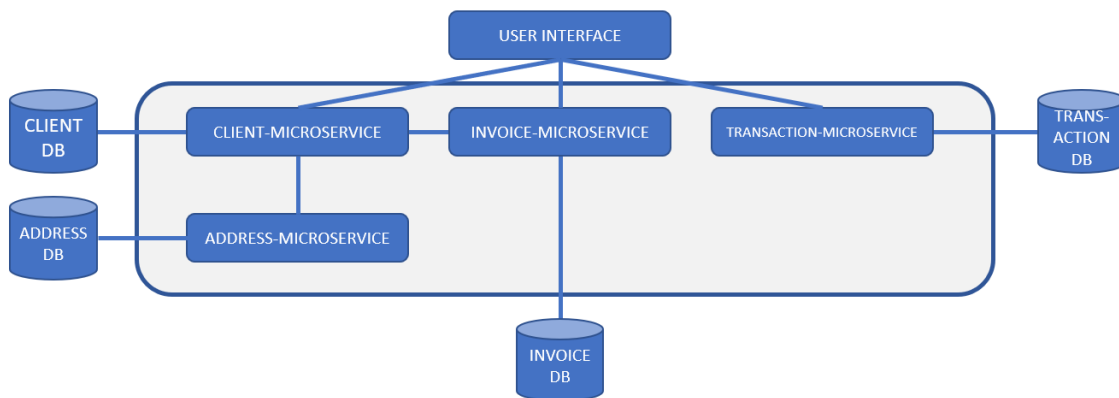
Als we kijken naar het domein van de algemene applicatie, kunnen we volgende *bounded contexts* beschouwen:

- Clienten (*Client*-microservice)
- Facturen (*Invoice*-microservice)
- Banktransacties (*Transaction*-microservice)

Omdat we voor toekomstige uitbreidingen van de applicatie rekening houden met het feit dat het *Address*-model en het *Document*-model herbruikt zou kunnen worden binnen andere toekomstige modellen, voorzien we ook volgende *bounded contexts* voor:

- Adressen (*Address*-microservice)
- Documenten (*Document*-microservice)

Een overzicht van bovengenoemde microservices is visueel weergegeven in figuur 4.4. Door hun eigen verantwoordelijkheden binnen het domein wordt op deze manier het modulaire karakter van microservices geïntanceerd.



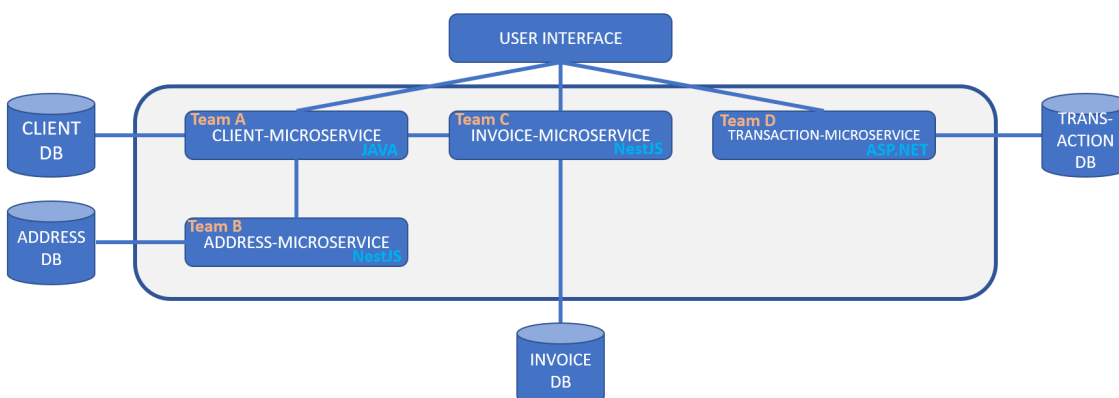
Figuur 4.4: Visuele representatie van de microservices in de *Accounting* applicatie

4.3.2 Ontwikkelen van de microservices

Doordat de grenzen van de microservices bepaald zijn op basis van hun verantwoordelijkheden binnen het domein, kunnen deze microservices onafhankelijk van elkaar ontwikkeld worden waarbij men een vrije technologiekeuze heeft per microservices. Zo is het mogelijk om de *Client*-microservice in *Java* te ontwikkelen, de *Address*-microservice in *NestJS* te ontwikkelen,...

Daarnaast heeft men ook de keuze om verschillende ontwikkelteams te verdelen over de verschillende microservices. Zo kan de *Client*-microservice ontwikkeld worden door *Ontwikkelteam A*, de *Address*-microservice ontwikkeld worden door *Ontwikkelteam B*,... . Dit sluit aan bij het modulaire karakter van microservices en zorgt er ook voor dat de individuele teams gespecialiseerd worden in hun microservice waardoor problemen sneller opgelost zullen raken. Daarnaast kunnen nieuwe teamleden zich ook sneller inwerken in één bepaalde microservice doordat door het afgebakende domein van de microservice de *codebase* beperkt is.

Een mogelijk voorbeeld wordt visueel weergegeven in figuur 4.5.



Figuur 4.5: Voorbeeld van verdeling van de ontwikkelteams en technologiekeuzes van microservices in de *Accounting* applicatie

4.3.3 Deployment van de microservices

Door het modulaire karakter van de microservices kunnen deze individueel en onafhankelijk van elkaar *gedeployed* worden. Wijzigingen in een bepaalde microservice mogen geen invloed hebben op andere microservices.

Verschillende *cloud* platformen, in combinatie met containerization en/of *service mesh*, bieden verschillende oplossingen voor diverse uitdagingen verbonden aan het deployen van microservices:

- Monitoring (observeren van de status van microservices, observeren van het netwerkverkeer tussen microservices,...)
- *Load balancing*
- Testing (*canary releasing*)
- Fouttolerantie (hoge beschikbaarheid door *auto-scaling*, gebruik van *circuit breakers*)

Omdat in de literatuurstudie verwezen wordt naar bepaalde platformen, worden volgende platformen geselecteerd voor het opstellen van het deployment van de microservices:

- Containerization: *Docker*
- Cloud platform: *Google Kubernetes Engine* (GKE)
- Service mesh: *Istio* in *Kubernetes*

Volgende paragrafen bevatten een stappenplan voor de configuratie. Om het overzicht in te bewaren worden de stappen abstract beschreven. Meer gedetailleerde stappenplannen zijn terug te vinden in hoofdstuk B. Er wordt reeds verondersteld dat voor zowel *Docker* als *Google Cloud Kubernetes* de nodige accounts zijn aangemaakt.

Docker Containerization

Nu de verschillende applicaties individueel ontwikkeld zijn, kunnen ze omgevormd worden tot *containerized* applicaties. Dit dient als voorbereiding op het deployment op GKE, waarvoor volgende stappen genomen dienen te worden (voor elke afzonderlijke microservice):

1. *Docker* installeren.
2. *Dockerfile* en eventueel *.dockerignore* toevoegen aan project.
3. *Docker-image* aanmaken.
4. (Optioneel) *Docker* container lokaal starten.
5. *Docker-image* online delen (*Google Cloud Container Registry*, *DockerHub*,...).

Zie bijlage B.1 voor een meer gedetailleerd stappenplan. Stap 2 tot en met 4 wordt herhaald voor alle applicaties die we in deze proof-of-concept als microservice wensen te gebruiken: *UI* (frontend), *Client*-microservice, *Address*-microservice, *Invoice*-microservice en *Transaction*-microservice. Het uiteindelijke resultaat bestaat uit een verzameling van *docker-images*. Zonder stap 5 zou de container enkel lokaal kunnen gebruikt worden. Omdat we gebruik wensen te maken *cloud computing* en onze applicatie online willen hebben, moeten we onze images ook online zetten. Aangezien we gebruik maken van

Google Kubernetes Engine, delen we onze images hiervoor op *Google Cloud Container Registry*.

Google Kubernetes Engine

Nu de verschillende applicaties *containerized* zijn, kunnen we ze uiteindelijk gebruiken om onze applicaties individueel te draaien. Echter is het niet gebruiksgemakkelijk om voor elke *docker-image* een *run* command te gaan uitvoeren. Omwille van deze reden maken we gebruik van *Kubernetes*. We hoeven enkel onze cluster te configureren (.yaml bestand) waarmee we *Kubernetes* informeren hoe onze cluster eruit moet zien. *Kubernetes* zelf is verantwoordelijk voor het beheer van alle *Pods*, *Nodes*,... . Dus als ergens een fout oploopt in één *Pod*, *Node*,... dan zal *Kubernetes* zelf instaan voor het garanderen van de beschikbaarheid die wij opgegeven hebben in de configuratie.

1. Een nieuw project aanmaken in *Google Cloud Console: accounting*
2. *Kubernetes Engine API* toevoegen aan *accounting* project
3. Cluster aanmaken voor het project
4. Container toevoegen aan de cluster
5. (Optioneel) Container verder configureren
6. Service (*Load balancer*) toevoegen
7. (Optioneel) *Kubernetes* cluster monitoring
8. (Optioneel) *Kubernetes* cluster logging
9. (Optioneel) *Kubernetes* cluster tracing

Zie bijlage B.2 voor een meer gedetailleerd stappenplan. Elke container die we toevoegen aan onze cluster is niet toegankelijk voor de buitenwereld. Omwille van deze reden voorzien we een *Kubernetes service* in de vorm van een *load balancer*. Stap 3 tot en met 6 wordt herhaald voor elke container (microservice).

Istio service mesh

Microservices communiceren doorgaans over het netwerk. Gezien de complexiteit van het netwerk zijn er heel wat elementen waar men rekening mee dient te houden zoals:

- Afhandelen van requests (routing)
- Hoe omgaan met fouten op het netwerk?
- Implementeren van *circuit breaker*,...
- Authenticatie over het netwerk
- ...

Hiervoor biedt een *service mesh* uitstekende oplossingen zonder iets te implementeren in de code zelf. We blijven in het *Google* territorium en leggen de keuze bij *Istio*. *Istio* is een *open-source service mesh* ontwikkeld door *Google*. *Istio* voorziet daarnaast nog extra features omtrent *tracing*, *logging* en monitoring waardoor de gezondheid van een systeem beter kan geobserveerd worden. *Istio* lijkt veel op *Kubernetes*, daar waar de configuraties ook via .yaml bestanden gedaan wordt en dat commando's uitgevoerd worden via *istioctl*.

Het enige waar de ontwikkelaar zich moet op concentreren is het correct configureren van *Istio* via *.yaml* bestanden.

1. *Istio* toevoegen aan het project.
2. *Istio service mesh* injectie activeren.
3. (Optioneel) Configureren van *egress* regels: configureren welke microservice wel met de externe *API*'s mogen communiceren.
4. (Optioneel) Configureren van routings tussen verschillende microservices.
5. Consulteren van metrics, logging, *mesh visualisatie* door verschillende add-ons van *Istio*.

Zie bijlage B.3 voor een meer gedetailleerd stappenplan.

5. Conclusie

De microservice-architectuur houdt enkele voordelen in voor zowel ontwikkelbedrijven als voor de eindgebruikers. Door hun modulaire karakter kan deze architectuurvorm voor ontwikkelbedrijven voordelig zijn doordat er beter omgegaan kan worden met veranderingen waarbij de focus makkelijker kan gelegd worden op ontwikkeling van de individuele microservices. Ook bieden microservices een zeer hoge fouttolerantie, beschikbaarheid en kunnen doorgaans beter geschaald worden. Echter zal de keuze voor deze architectuurvorm eerder afhangen van de grootte van het project, gezien de vele uitdagingen die verbonden zijn met deze architectuur. Communicatie tussen de verschillende microservices, het voorzien van *load balancers*, *service discovery*, hoe omgaan met data, het monitoren van de verschillende microservices, *end-to-end* testing, performantie, implementeren van elementen die een hogere fouttolerantie kunnen bieden,... zijn onderwerpen die, vooral in vergelijking met de monolithische architectuur, uitdagingen vormen waarmee rekening dient gehouden te worden om de microservice-architectuur toe te passen. Echter bleek in dit onderzoek dat vooral *cloud computing* hierbij kan helpen door verschillende van deze problemen op een geautomatiseerde manier te verhelpen. Kubernetes wordt als standaard gezien binnen de sector voor gedistribueerde microservice-systemen en biedt verschillende geautomatiseerde oplossingen inzake maximale fouttolerantie en beschikbaarheid (*auto-scaling*, *rolling updates*), *load balancing*, monitoring, netwerkcontrole,... . Het is dus duidelijk dat de microservice-architectuur, vooral bij grote projecten, voordelig kan zijn vooral in combinatie dergelijke IT infrastructuur. Echter zijn er nog enkele thema's omtrent microservices die nog verder onderzocht kunnen worden binnen het onderzoeksdomein zoals *security* en testing. Hoewel de thema's in het literatuuronderzoek vernoemd werden als uitdagingen verbonden aan microservices, werd er nog geen directe oplossingen gevonden in dit onderzoek.

A. Onderzoeksvoorstel

Het onderwerp van deze bachelorproef is gebaseerd op een onderzoeksvoorstel dat vooraf werd beoordeeld door de promotor. Dat voorstel is opgenomen in deze bijlage.

A.1 Introductie

Het doel van dit onderzoek is om een zo'n duidelijk mogelijk beeld te geven van de microservice architectuur waarbij daarnaast een verklarende uitleg zal gegeven worden rond de uitdagingen en de reeds bestaande oplossingen voor deze uitdagingen. Hiermee wordt getracht ontwikkelaars zo goed mogelijk te informeren rond deze architectuur en een hulptool te bieden bij de praktische implementatie van deze architectuur.

Volgende onderzoeksvragen zullen beantwoord worden in het onderzoek:

- Wat is de microservice architectuur (in vergelijking met de traditionele monolitische architectuur)?
- Wat zijn de voor- en nadelen van de microservice architectuur?
- Welke frameworks bieden oplossingen voor de uitdagingen verbonden aan de microservice architectuur?

A.2 Literatuurstudie

Westerlinck (2019) onderzoekt de verschillen tussen microservice en monolitische architectuur bij applicatieontwikkeling. Beide architecturen worden beschreven, alsook hun voor- en nadelen. Daarnaast werd er ook een onderzoek gedaan in de verschillen in performantie

voor beide architecturen waarbij de monolitische architectuur echter voordeliger bleek te zijn tot op een bepaald keerpunt (aantal gelijktijdige gebruikers).

Singh (2018), Senior IoT Tempus ontwikkelaar, beschrijft in zijn artikel wat microservices inhouden en wat de voordelen zijn. Daarnaast worden de meest voorkomende uitdagingen rond deze architectuur beschreven en frameworks die oplossingen bieden voor deze uitdagingen.

Götza e.a. (2018) beschrijven de verschillen tussen de monolitische en microservice architectuur. Daarnaast wordt er ook meegegeven waarmee men rekening moet houden bij de implementatie van de microservice architectuur. Tot slot werd een praktische implementatie van deze architectuur als voorbeeld beschreven.

In het onderzoek van Edling en Östergren (2017) worden kort de monolitische en microservice architectuur besproken. Van hieruit werd ook onderzocht welke frameworks er beschikbaar zijn voor de implementatie van een microservice architectuur. Hierbij hebben de onderzoekers een monolitische applicatie omgevormd tot een microservice applicatie gebruikmakend van de twee meest belovende frameworks omtrent microservices.

A.3 Methodologie

Het theoretisch gedeelte van dit onderzoek voor het leveren van de vergelijkende studie tussen monolitische en microservice architectuur zal gebeuren aan de hand van de analyse van verschillende papers, thesissen, artikels, Het doel is een zo een volledig mogelijk beeld te vormen van de microservice architectuur.

Het praktisch gedeelte van dit onderzoek omvat het ontwikkelen van een microservice. Deze microservice zal functionaliteiten bevatten die in verschillende projecten terugkomen. Door het toepassen van de microservice architectuur zal deze microservice gebruikt kunnen worden binnen de verschillende lopende of toekomstige projecten. De frameworks die uit het theoretisch gedeelte naar voor komen, zullen gebruikt worden in de praktische gedeelte om tegemoet te komen aan uitdagingen verbonden aan de microservice architectuur.

A.4 Verwachte resultaten

Het verwachte resultaat van het onderzoek bestaat eruit een microservice te ontwikkelen die hergebruikt kan worden in verschillende projecten. Dit resultaat zal voor ontwikkelaars een verhoogde efficiëntie betekenen in:

- de implementatie van microservices
- het onderhoud van de microservices
- Continuous Integration/Continuous Delivery
- de kennis omtrent frameworks voor microservices

A.5 Verwachte conclusies

Microservices brengen heel wat uitdagingen met zich mee, vooral in vergelijking met monolitische applicaties. Voor bedrijven kan het juist interessant zijn om te kiezen voor de implementatie van microservices omdat deze een meerwaarde kunnen bieden voor ontwikkelaars of ontwikkelbedrijven. Dit onderzoek zal ontwikkelaars technisch en praktisch informeren om de microservice architectuur toe te passen.

B. Bijlagen

B.1 Docker Containerization

Hier is een meer gedetailleerd stappenplan terug te vinden met extra informatie of afbeeldingen om *Docker-images* aan te maken.

Stap 1: Docker installeren

Docker kan hier gedownload worden. Nadat *Docker* geïnstalleerd is, kan de installatie gecontroleerd worden door volgend commando (B.1):

```
nickg@LAPTOP-PKD5CQLE MINGW64 ~/Documents/H0-Gent/Cursussen/Stage/AdvantITge/Stageverloop/Projecten/Accounting/Web
$ docker --version
Docker version 19.03.1, build 74b1e89e8a
```

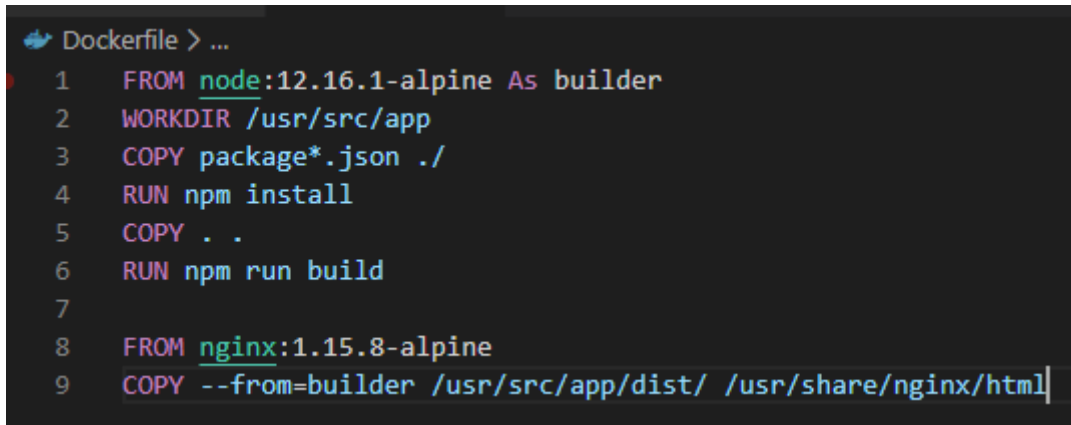
Figuur B.1: Docker installatie - versie controle

Stap 2: Dockerfile en (eventueel) .dockerignore bestanden toevoegen

Zowel *Dockerfile* als *.dockerignore* worden in de *root* van het project toegevoegd (figuur B.2).

```
nickg@LAPTOP-PKD5CQLE MINGW64 ~/Documents/H0-Gent/Cursussen/Stage/AdvantITge/Stageverloop/Projecten/Accounting/Web
$ touch Dockerfile .dockerignore
```

Figuur B.2: Docker - Dockerfile en .dockerignore toevoegen



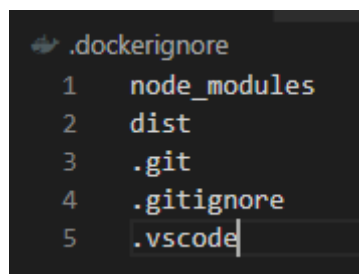
```
Dockerfile > ...
1 FROM node:12.16.1-alpine As builder
2 WORKDIR /usr/src/app
3 COPY package*.json ./
4 RUN npm install
5 COPY . .
6 RUN npm run build
7
8 FROM nginx:1.15.8-alpine
9 COPY --from=builder /usr/src/app/dist/ /usr/share/nginx/html
```

Figuur B.3: Docker - Dockerfile voorbeeld

Dockerfile We voegen een *Dockerfile* toe die de configuratie bevat hoe de *docker-image* moet aangemaakt worden. Het volledig beschrijven van alle mogelijke configuraties ligt buiten de scope van dit onderzoek. Dockerfile in figuur B.3 zal volgende stappen ondernemen:

1. Basis image gebruiken voor de nodige configuratie. Aangezien we *Angular* gebruiken voor frontend, gebruiken we hier *node* als basis-image.
2. Een *working directory* voorzien in de container.
3. *package.json* en *package-lock.json* kopiëren van het project naar de container.
4. Installeren van de nodige *libraries*, *dependencies* van het project (aan de hand van de *package.json* en *package-lock.json* bestanden).
5. Alles applicatiebestanden van het project kopiëren naar de container.
6. *npm run build* uitvoeren voor het compileren van de applicatie in de container.
7. *nginx* wordt meestal gebruikt voor het weergeven van grafische inhoud.
8. Kopieëren van het gecompileerd project naar de nodige *nginx* folder.

Standaard wordt een *port* (:80) voorzien voor elke container. Meer informatie omtrent *Dockerfile* is hier terug te vinden.



```
.dockerignore
1 node_modules
2 dist
3 .git
4 .gitignore
5 .vscode
```

Figuur B.4: Docker - .dockerignore voorbeeld

.dockerignore Er kan ook eventueel een *.dockerignore* bestand voorzien worden in het project waarmee we kunnen specificeren welke bestanden / mappen niet opgenomen moeten worden in de container. *.dockerignore* bestand in figuur B.4 zal de geïnstalleerde packages

in het project (node-modules) niet kopiëren aangezien deze geïnstalleerd worden via *Dockerfile* (npm install). Hetzelfde geldt voor de *dist* map aangezien het gecompileerd project wordt reeds toegevoegd aan de container (npm run build). GIT-configuraties, IDE-configuraties,... zijn daarnaast ook niet nodig voor de container.

Meer informatie omtrent *Dockerfile* is hier terug te vinden.

Stap 3: Docker-image aanmaken

Met de nodige configuraties (*Dockerfile* en *.dockerignore*) kan nu een *docker-image* aangemaakt worden. Hierbij wordt ook een *tag* opgegeven waarmee er nadien makkelijker kan gerefereerd worden. Alle stappen die zijn geconfigureerd in de *Dockerfile* worden

```
nickg@LAPTOP-PKD5CQLE MINGW64 ~/Documents/H0-Gent/Cursussen/Stage/AdvantITge/Stageverloop/Projecten/Accounting/Web
$ docker build -t accounting-web .
Sending build context to Docker daemon 1.097MB
Step 1/10 : FROM node:12.16.1-alpine As builder
--> f77abbe89ac1
Step 2/10 : WORKDIR /usr/src/app
--> Using cache
--> d00a072bce94
Step 3/10 : COPY package*.json ./
--> Using cache
--> dab8ad7dd848
Step 4/10 : RUN npm install
--> Using cache
--> a352790f02c2
Step 5/10 : COPY . .
--> 5f6555abc3ec
Step 6/10 : RUN npm run build
--> Running in 2ab571c39c0d
```

Figuur B.5: Docker - Docker-image aanmaken

doorlopen en resulteert uiteindelijk in een *docker-image*.

De *docker-image*, alsook de geïnstalleerde basis *images* (*node* en *nginx*) zijn nu terug te vinden in *Docker* op de lokale computer. Dit is ook terug te vinden in het overzicht van alle *docker-images* (zie figuur B.6).

```
nickg@LAPTOP-PKD5CQLE MINGW64 ~/Documents/H0-Gent/Cursussen/Stage/AdvantITge/Stageverloop/Projecten/Accounting/Web
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
accounting-web	latest	fbdb2d545a5a	2 hours ago	68.4MB
node	12.16.1-alpine	f77abbe89ac1	2 months ago	88.1MB
nginx	1.15.8-alpine	b411e34b4606	16 months ago	16.1MB

Figuur B.6: Docker - Docker-images overzicht

(Optioneel) Stap 4: Lokaal starten van de container aan de hand van de *docker-image*

De *docker-image* kan nu gebruikt worden om een container op te starten. Deze kan alvast lokaal opgestart worden met het commando in figuur B.7. Wanneer de container gestart is, is de *containerized* applicatie beschikbaar op <http://192.168.99.100:5000/>. Figuur B.8 toont hoe een overzicht van actieve containers kan gegeven worden.

```
nickg@LAPTOP-PKD5CQLE MINGW64 ~/Documents/H0-Gent/Cursussen/Stage/AdvantITge/Stageverloop/Projecten/Accounting/Web
$ docker run -p 5000:80 accounting-web
```

Figuur B.7: Docker - Docker container starten

```
nickg@LAPTOP-PKD5CQLE MINGW64 ~/Documents/H0-Gent/Cursussen/Stage/AdvantITge/Stageverloop/Projecten/Accounting/Web (feature/#58jyqg)
$ docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
8d5d2391eeb8        accounting-web      "nginx -g 'daemon of..." 37 seconds ago      Up 37 seconds      0.0.0.0:5000->80/tcp  elastic_dewdney
```

Figuur B.8: Docker - Docker actieve containers overzicht

Stap 5 - Docker-image plaatsen op Google Cloud Container Registry

In deze stap wordt de *docker-image* voorbereid om gedeeld te worden met *Google Cloud Container Registry*. Er wordt verondersteld dat reeds een project is aangemaakt op *Google Cloud* en dat *Container Registry API* is toegevoegd aan het project. Om deze online opslagplaats voor containers te kunnen gebruiken, moeten we onze *docker-image* voorzien van een *tag* ([locatie GC CR]/[GC project-id]/[image-naam]:[image-tag]). Wanneer de *tag* is voorzien, kan deze gedeeld worden op GC Container Registry. Beide commando's zijn weergegeven in figuur B.9. Deze image is hierna zichtbaar in het Container Registry op

```
nickg@LAPTOP-PKD5CQLE MINGW64 ~/Documents/H0-Gent/Cursussen/Stage/AdvantITge/Stageverloop/Projecten/Accounting/Web
$ docker tag accounting-web eu.gcr.io/accounting-278513/accounting-web

nickg@LAPTOP-PKD5CQLE MINGW64 ~/Documents/H0-Gent/Cursussen/Stage/AdvantITge/Stageverloop/Projecten/Accounting/Web
$ docker push eu.gcr.io/accounting-278513/accounting-web
The push refers to repository [eu.gcr.io/accounting-278513/accounting-web]
12fdd360138c: Pushed
979531bcfa2b: Pushed
8d36c62f099e: Pushed
4b735058ece4: Pushed
503e53e365f3: Layer already exists
latest: digest: sha256:f1cc011fc2fce4452e16db0d857039f672e2bdda4e412fc4160bb5d24fef0b4c size: 1365
```

Figuur B.9: Docker - Docker-image naar GC Container Registry sturen

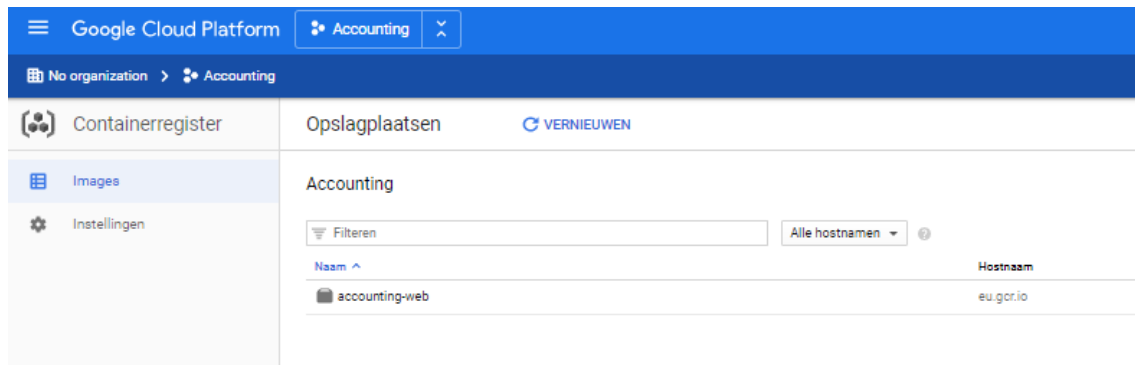
Google Cloud (figuur B.10).

B.2 Google Kubernetes Engine

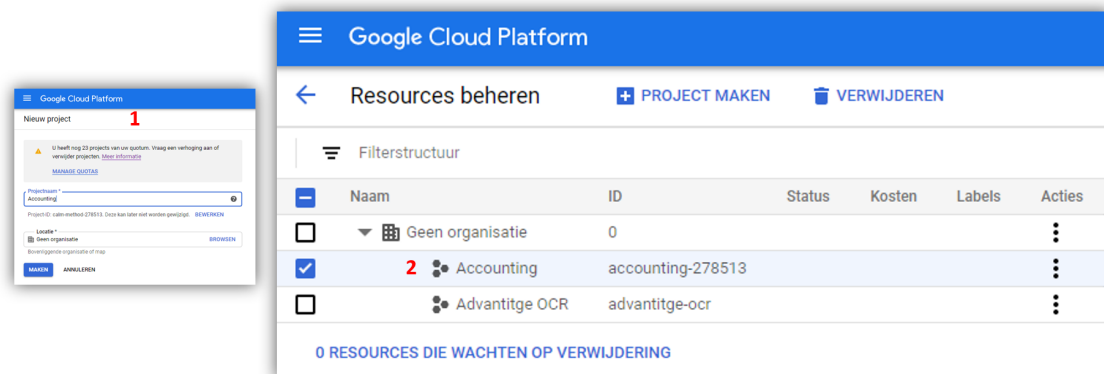
Het gebruik van *Google Kubernetes Engine* (GKE) heeft als voordeel dat vele operaties, configuraties,... via een UI kunnen verlopen. Daarnaast kan men ook commando's geven via *kubectl*. In dit stappenplan zijn beide manieren om configuraties te voorzien in *Kubernetes* gebruikt.

Stap 1: Nieuw project maken in *Google Cloud* (Console)

Eerst wordt er een nieuw project aangemaakt in *Google Cloud*. Na het toevoegen van het project in *Google Cloud* (zie figuur B.11) is het project terug te vinden in het projectoverzicht (zie figuur B.12).



Figuur B.10: Docker - Docker-image in GC Container Registry overzicht



Figuur B.11: Google Cloud - Project aanmaken

Stap 2: Kubernetes API toevoegen aan project

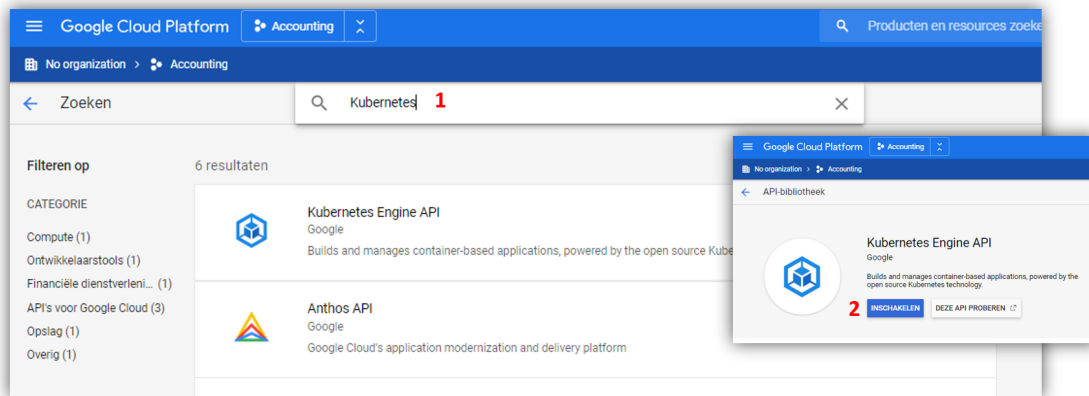
Om gebruik te kunnen maken van *Kubernetes* in *Google Cloud* moeten we *Kubernetes API* toevoegen aan het project (zie figuur B.12). Na het toevoegen van *Kubernetes API* is deze ook terug te vinden in het *Google Cloud Dashboard* (overzicht van actieve API's).

Stap 3: Kubernetes cluster aanmaken voor het project

Voor het aanmaken van een *Kubernetes cluster*, zie figuur B.13.

Stap 4: Toevoegen van een container aan de cluster

De volgende stap is het toevoegen van de container aan de cluster. Aangezien we deze container op *Google Cloud Container Registry* hebben geplaatst, kan deze makkelijk opgehaald worden door *Google Kubernetes Engine*. Zie figuur B.14.



Figuur B.12: Google Cloud - Kubernetes API toevoegen aan project

Stap 5: (Optioneel) Container verder configureren

De toegevoegde container kan verder geconfigureerd worden aan de hand van *UI tools*, *kubectl*,... . Hierbij kunnen configuraties zoals *auto-scaling*, *rolling updates*,... verder gespecificeerd worden (zie figuur B.15).

Stap 6: *Load balancer* toevoegen

Een container kan lokaal draaien zonder *load balancer* omdat hier doorgaans maar één IP adres is voorzien. Echter moet er een toegangspunt zijn voor onze container in de *Kubernetes* cluster. Praktisch komt het erop neer dat verschillende *containers* op verschillende *nodes*. Deze *nodes* zijn fysieke of virtuele hardware die onze container draait. Dus verschillende instanties van één container zitten verspreid over meerdere *nodes*. *Kubernetes* zal inkomende *requests* behandelen en verdelen over de verschillende instanties om zo het netwerkverkeer te verdelen. Hiervoor maken we in *Kubernetes* een *service* (*load balancer*) aan. Zie figuur B.16. Hierdoor wordt door *Kubernetes* een *load balancer* die het verkeer van en naar de verschillende instanties van onze container gaat verdelen. Deze krijgt een publiek IP adres toegewezen door *Google*, waardoor onze applicatie publiek toegankelijk is via dit IP-adres.

(Optioneel) Kubernetes cluster monitoring

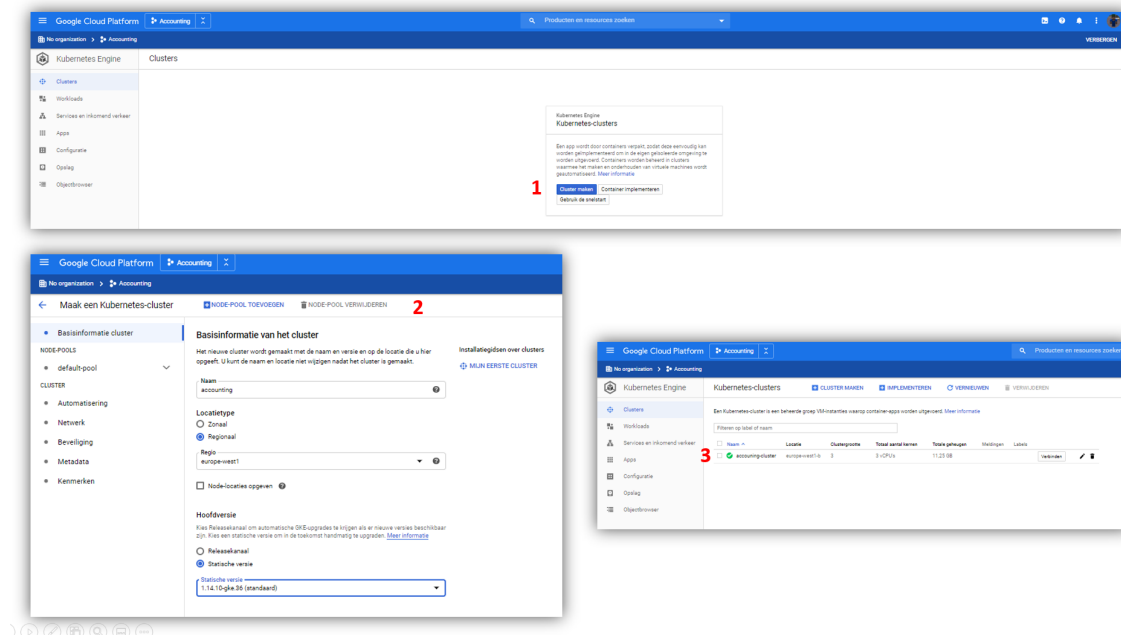
Zie figuur B.17.

(Optioneel) Kubernetes cluster logging

Zie figuur B.18.

(Optioneel) Kubernetes cluster tracing

Zie figuur B.19.



Figuur B.13: Kubernetes - Cluster aanmaken

B.3 Istio

Stap 1: Istio toevoegen aan project

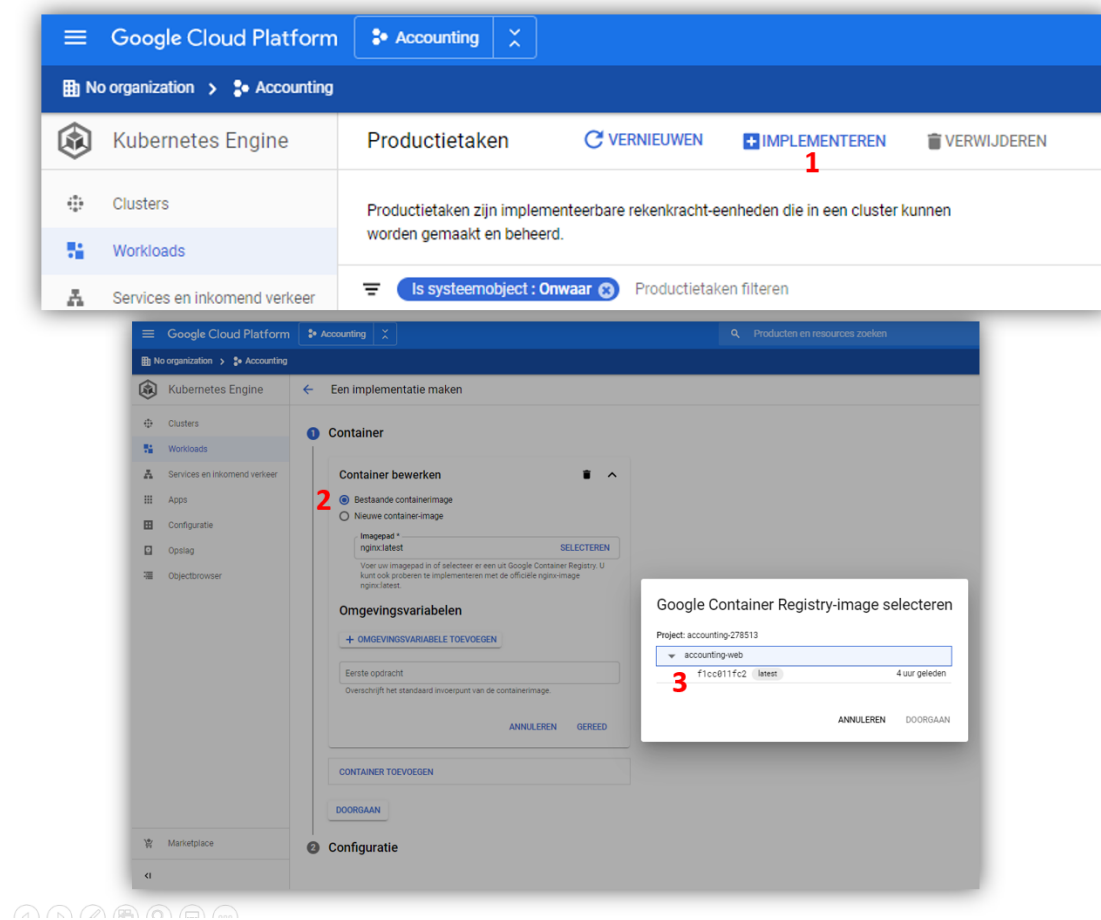
We voegen *Istio* toe aan het cluster-project op *Google Kubernetes Engine*. Na het toevoegen kunnen we *Istio* terug vinden in de lijst met beschikbare *Kubernetes services*. Zie figuur B.20.

Stap 2: Istio service mesh injectie activeren

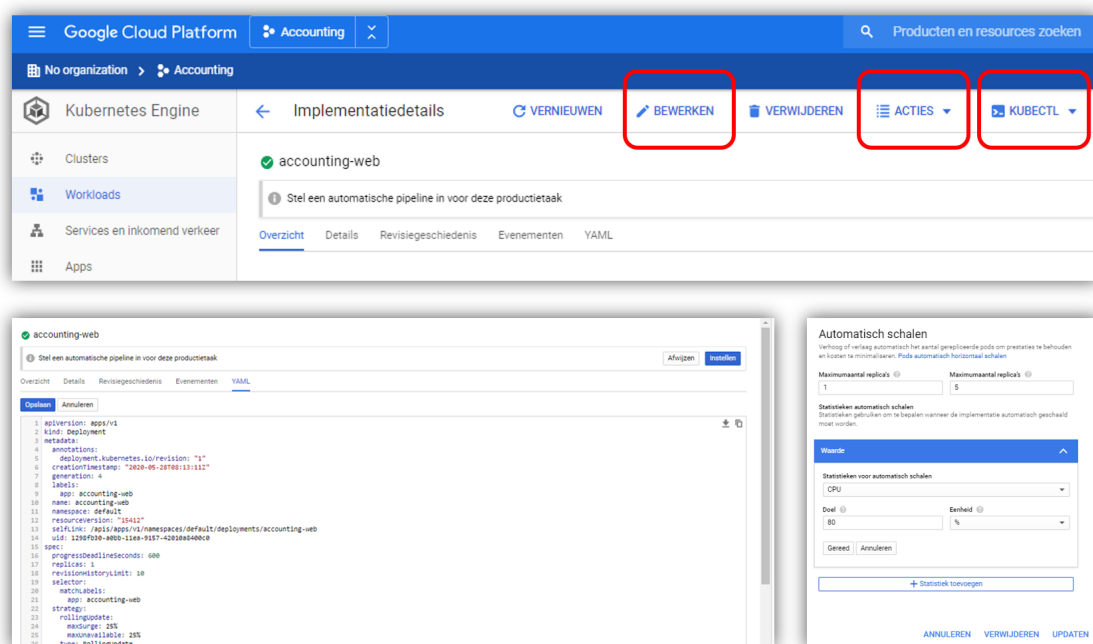
Service mesh injectie moet geactiveerd worden. Zie figuur B.21.

Stap 3: Monitoring met Grafana

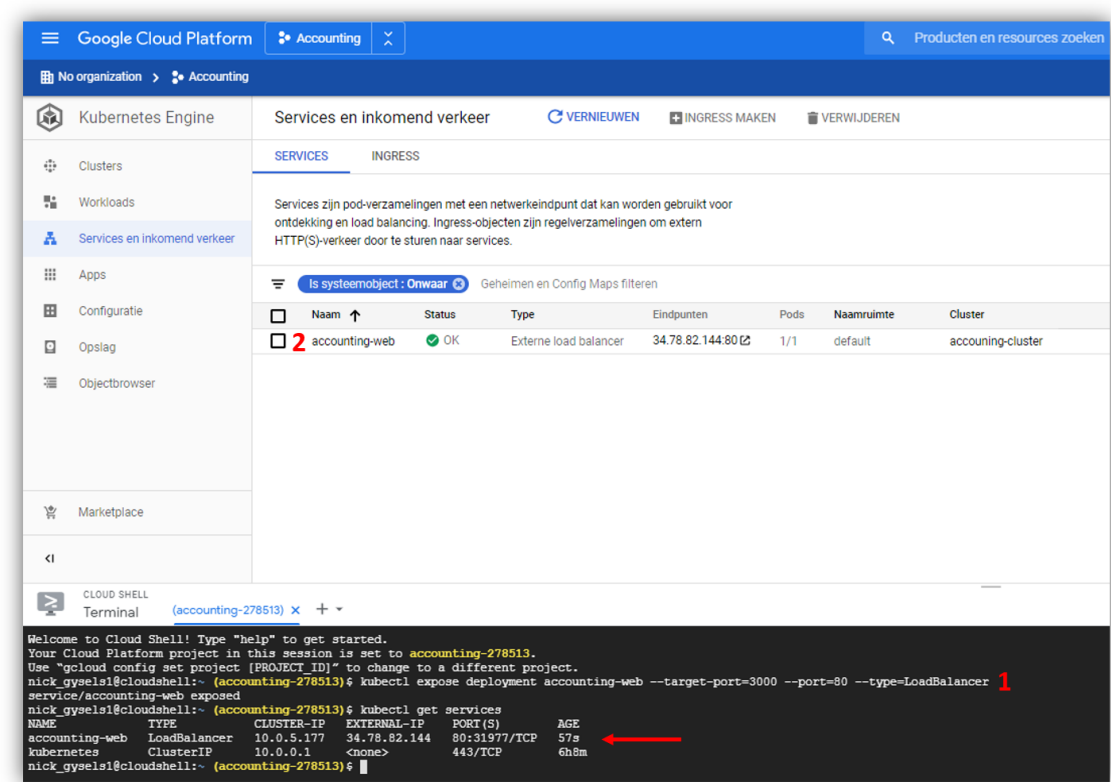
Wanneer we met de browser de Grafana monitoring UI (url: <http://localhost:3000/dashboard/db/istio-mesh-dashboard>) raadplegen kunnen we *cluster services*, *workloads (containers)* en *mesh* gedetailleerd monitoren. Zie figuur B.22



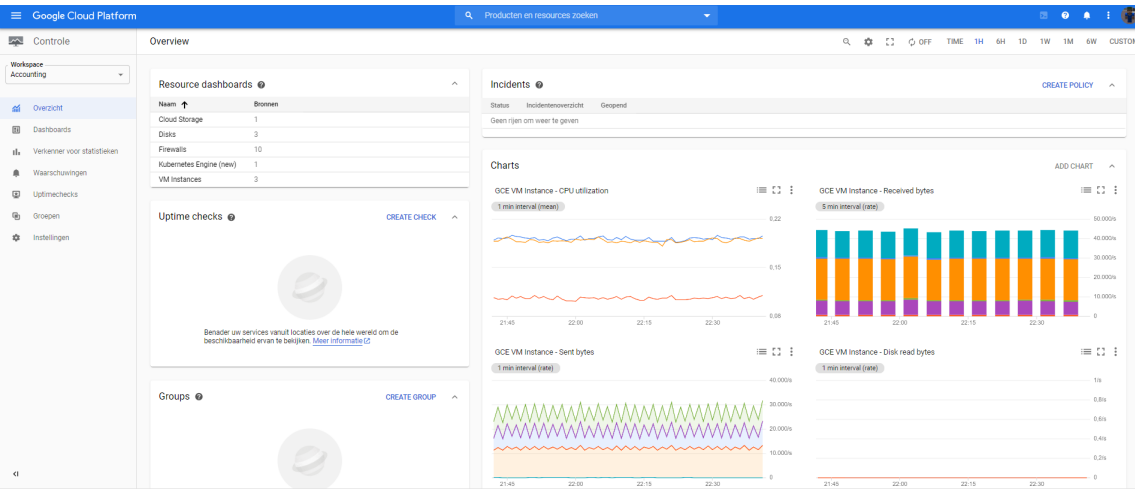
Figuur B.14: Kubernetes - Container toevoegen



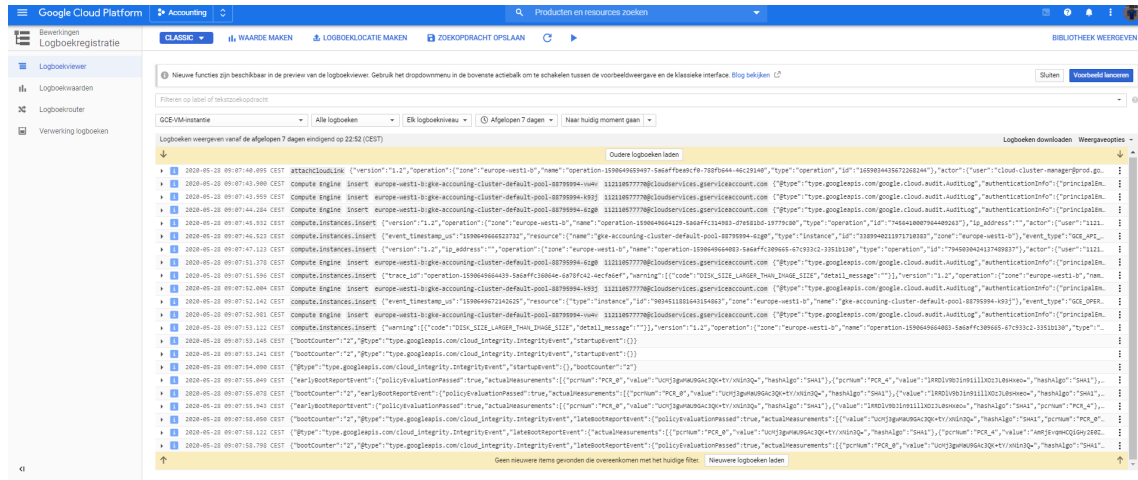
Figuur B.15: Kubernetes - Container configureren



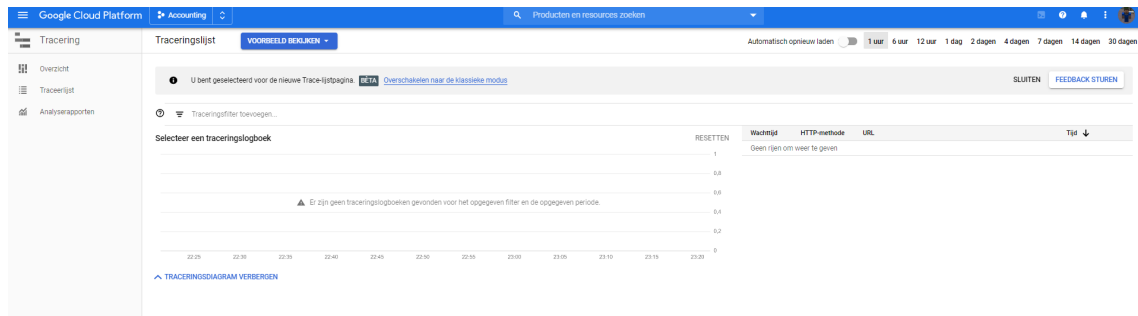
Figuur B.16: Kubernetes - Load balancer toevoegen



Figuur B.17: Google Cloud - Cluster monitoring



Figuur B.18: Google Cloud - Cluster logging



Figuur B.19: Google Cloud - Cluster tracing

```

nick_gysels1@cloudshell:~ (accounting-278513) $ gcloud beta container clusters update accounting-cluster --update-addons-Istio=ENABLED --istio-config-auth-MTLS_STRICT --region=europe-west-1-b
Updating accounting-cluster...done.
Updated [https://container.googleapis.com/v1beta1/projects/accounting-278513/zones/europe-west-1-b/clusters/accounting-cluster].
To inspect the contents of your cluster, go to: https://console.cloud.google.com/kubernetes/workload/gcloud/europe-west-1-b/accounting-cluster/project-accounting-278513

nick_gysels1@cloudshell:~ (accounting-278513) $ kubectl get service -n istio-system
NAME                TYPE        CLUSTER-IP      EXTERNAL-IP      PORT(S)
istio-citadel        ClusterIP    10.0.0.12        <none>            8060/TCP,15014/TCP
istio-galley         ClusterIP    10.0.0.243       <none>            443/TCP,15014/TCP,9090/TCP
istio-ingressgateway LoadBalancer 10.0.0.196       35.205.134.70    15020:30465/TCP,80:30465/TCP,443:32472/TCP,31400:30165/TCP,15029:31773/TCP,15030:32171/TCP,15031:31461/TCP,15032:32110/TCP,15443:31068/TCP
istio-pilot          ClusterIP    10.0.0.11        <none>            15010/TCP,15011/TCP,8080/TCP,15014/TCP
istio-policy          ClusterIP    10.0.0.312       <none>            9091/TCP,15004/TCP,15014/TCP
istio-sidecar-injector ClusterIP     10.0.0.195       <none>            443/TCP
istio-telemetry       ClusterIP    10.0.0.179       <none>            9091/TCP,15004/TCP,15014/TCP,42422/TCP
prometheus            ClusterIP    10.0.0.241       <none>            9090/TCP

```

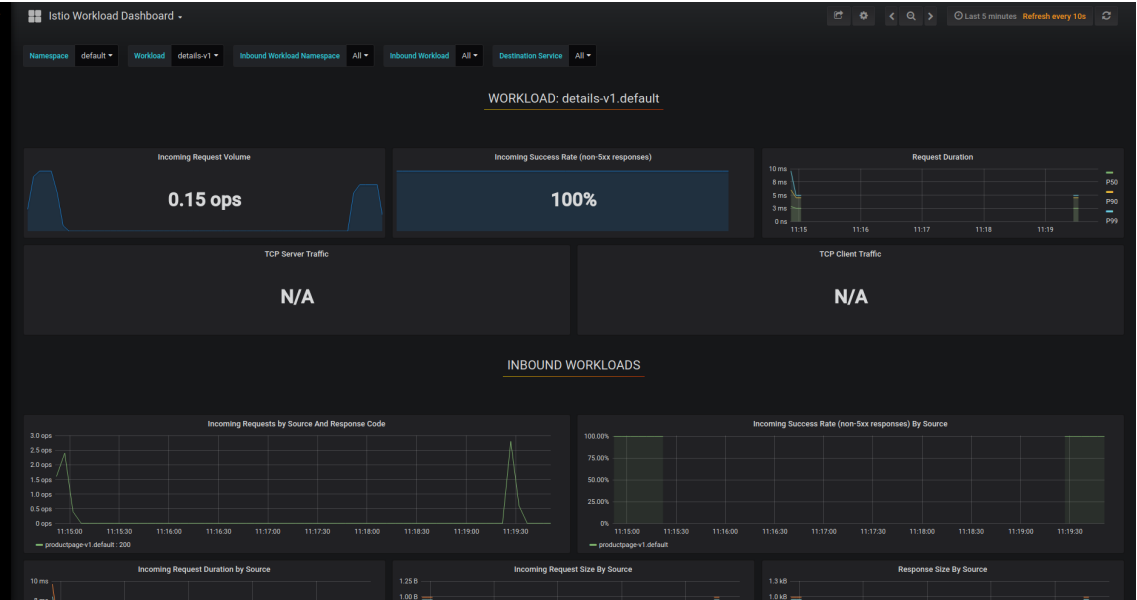
Figuur B.20: Istio - Toevoegen aan cluster

```

nick_gysels1@cloudshell:~ (accounting-278513) $ kubectl label namespace default istio-injection=enabled
namespace/default labeled

```

Figuur B.21: Istio - Service mesh injectie activeren



Figuur B.22: Istio - Grafana monitoring UI

Bron: Istios.io - Grafana UI

Bibliografie

- Alliance, A. (2020, april 19). *What is Agile?* <https://www.agilealliance.org/agile101/>
- Alshuqayran, N., Ali, N. & Evans, R. (2016). A Systematic Mapping Study in Microservice Architecture, In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. <https://doi.org/10.1109/SOCA.2016.15>
- Amaral, M., Polo, J., Carrera, D., Mohomed, I., Unuvar, M. & Steinder, M. (2015). Performance Evaluation of Microservices Architectures Using Containers, In *2015 IEEE 14th International Symposium on Network Computing and Applications*.
- Aussems, M. (2018, juli 25). *Kubernetes, wat is het en waarom verover het in snel tempo de wereld?* <https://www.techzine.be/blogs/cloud/16382/kubernetes-wat-is-het-en-waarom-verover-het-in-snel-tempo-de-wereld/>
- Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, D. A. & Lynn, T. (2018). Microservices migration patterns. *Software: Practice and Experience*, 48(11), <https://onlinelibrary.wiley.com/doi/10.1002/spe.2608>
- Braun, J. (2020, april 19). *Proof of Concept*. https://afdelingbuitengewonezaken.nl/tag/proof-of-concept?_locale=nl
- Docker. (2020, mei 12). *Docker documentation*. <https://docs.docker.com/>
- Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R. & Safina, L. (2017). Microservices: Yesterday, Today, and Tomorrow. In M. Mazzara & B. Meyer (Red.), *Present and Ulterior Software Engineering* (pp. 195–216). Cham, Springer International Publishing. https://doi.org/10.1007/978-3-319-67425-4_12
- Dragoni, N., Lanese, I., Larsen, S. T., Mazzara, M., Mustafin, R. & Safina, L. (2018). Microservices: How To Make Your Application Scale (A. K. Petrenko & A. Voronkov, Red.). In A. K. Petrenko & A. Voronkov (Red.), *Perspectives of System Informatics*, Cham, Springer International Publishing. https://doi.org/10.1007/978-3-319-74313-4_8

- Edling, E. & Östergren, E. (2017). *An analysis of microservice frameworks* (onderzoeksrap.). Linköping University.
- Forcepoint.com. (2020, april 26). *What is a firewall?* <https://www.forcepoint.com/cyber-edu/firewall>
- GitHub.com. (2020a, april 19). *About branches*. <https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/about-branches>
- GitHub.com. (2020b, april 19). *Committing and reviewing changes to your project*. <https://help.github.com/en/desktop/contributing-to-projects/committing-and-reviewing-changes-to-your-project>
- Götza, B., Schela, D., Bauera, D., Henkela, P., C. and Einbergera & Bauernhansla, T. (2018, maart 1). Challenges of production microservices, In *Procedia CIRP*, Vol.67.
- Hoogenraad, W. (2017, november 1). *Wat betekent Best Practice?* <https://www.itpedia.nl/2017/11/01/wat-betekent-best-practice/>
- IBM. (2020, mei 24). *Containerization*. <https://www.ibm.com/cloud/learn/containerization>
- Kalske, M., Mäkitalo, N. & Mikkonen, T. (2018). Challenges When Moving from Monolith to Microservice Architecture (I. Garrigós & M. Wimmer, Red.). In I. Garrigós & M. Wimmer (Red.), *Current Trends in Web Engineering*, Cham, Springer International Publishing. https://doi.org/https://doi.org/10.1007/978-3-319-74433-9_3
- Kubernetes. (2020, mei 24). *Kubernetes documentation*. <https://kubernetes.io/docs>
- Li, X., Xi, Y., Zhu, H., Ling, J. & Zhang, Q. (2020). Infrastructure Smart Service System Based on Microservice Architecture (A. G. Correia, J. Tinoco, P. Cortez & L. Lamas, Red.). In A. G. Correia, J. Tinoco, P. Cortez & L. Lamas (Red.), *Information Technology in Geo-Engineering*, Cham, Springer International Publishing.
- Malfait, I. (2015). *Analyse 1 - Use cases*.
- MuleSoft.com. (2020, april 26). *What is an API?* <https://www.mulesoft.com/resources/api/what-is-an-api>
- Namiot, D. & Sneps-Sneppe, M. (2014). On Micro-services Architecture. In *International Journal of Open Information Technologies* (9de ed.).
- Newman, S. (2015). *Building Microservices* (1st). O'Reilly Media, Inc.
- Poulton, N. & Joglekar, P. (2019). *The Kubernetes Book: The fastest way to get your head around Kubernetes*. Packt Publishing. <https://books.google.be/books?id=0jqgDwAAQBAJ>
- Silva, M. V. (2019, oktober 15). *Deployment vs Release*. <https://medium.com/code-thoughts/deployment-vs-release-543fe9f26272>
- Singh, J. (2018, juni 7). *The What, Why, and How of a Microservices Architecture*. <https://medium.com/hashmapinc/the-what-why-and-how-of-a-microservices-architecture-4179579423a9>
- Stubbs, J., Moreira, W. & Dooley, R. (2015). Distributed Systems of Microservices Using Docker and Serfnode, In *Proceedings of the 2015 7th International Workshop on Science Gateways*, USA, IEEE Computer Society. <https://doi.org/10.1109/IWSG.2015.16>
- Techopedia. (2020, april 19). *Build*. <https://www.techopedia.com/definition/3759/build>
- Toffetti, G., Brunner, S., Blöchlinger, M., Dudouet, F. & Edmonds, A. (2015). An Architecture for Self-Managing Microservices, In *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*, Bordeaux, France, Association for Computing Machinery. <https://doi.org/10.1145/2747470.2747474>

- van Veen, P. A. F. & van der Sijs, N. (1997). *Etymologisch woordenboek: de herkomst van onze woorden* (2de ed.). Antwerpen, Utrecht, Van Dale Lexicografie. <http://www.etymologiebank.nl/trefwoord/monoliet>
- Victor, M., Rohit, J. & Tim, H. (2015, februari 14). Networking in Containers and Container Clusters, In *netdev 0.1*.
- Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R. & Gil, S. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud, In *2015 10th Computing Colombian Conference (10CCC)*. <https://doi.org/10.1109/ColumbianCC.2015.7333476>
- Wang, Y. & Ma, D. (2019). *Developing a Process in Architecting Microservice Infrastructure with Docker, Kubernetes, and Istio*.
- Westerlinck, N. (2019). *Een vergelijkende studie over de performantie van monolith en microservices*. (onderzoeksrap.). Realdolmen.
- Zhang, W. (2020, mei 5). *Improving Microservice Reliability with Istio* (onderzoeksrap.). University of British Columbia, Vancouver, Canada.