KU LEUVEN

FACULTEIT WETENSCHAPPEN

Het balanceren van Servicecontinuïteit en Surgekost tijdens Continuous Deployment van Multi-tenant Applicaties

Arnout HOEBRECKX

Promotor: Prof. dr. ir. W. Joosen
DistriNet

Co-promotor: Dr. E. Truyen
DistriNet

Begeleider: Dr. E. Truyen

DistriNet

Assessoren:

Dr. B. Lagaisse

Prof. dr. D. Hughes

Proefschrift ingediend tot het
behalen van de graad van
Master of Science in Toegepaste Informatica

Academiejaar 2019-2020

Voorwoord

Na een bewogen jaar, afgesloten met een uniek semester, is de masterproef tot een goed einde gebracht. Het afleveren van zo een waardevol document waar veel bloed, zweet en tranen zijn ingekropen zou niet mogelijk geweest zijn zonder de mensen rondom mij. Daarom zou ik nog graag even de tijd nemen om enkele personen te bedanken.

De begeleiding van mijn promotor professor W. Joosen en mentor E. Truyen was van onschatbare waarde en cruciaal voor de aflevering van deze masterproef.

Daarnaast wil ik graag mijn ouders bedanken, zij hebben mij de kans gegeven om deze studies aan te gaan en mezelf te bewijzen. De opleiding was zeker niet altijd even makkelijk en zonder mijn vriendin was dit een zware klus geweest. Ze stond altijd klaar voor mij doorheen deze periode en heeft een doorslaggevende factor gespeeld in mijn succes. Als laatste wil ik mijn vrienden bedanken om tijdens deze periode toch voor de nodige steun en afleiding te zorgen.

Arnout Hoebreckx

Korte Samenvatting

Software as a Service (SaaS) is een software architectuur waarbij applicaties via een online verbinding aan klanten worden aangeboden. Tenants, een andere benaming voor klanten, zullen gebruik maken van de applicatie tegen een bepaalde kostprijs en verwachten hiervoor in ruil een welbepaald serviceniveau. Een kostenefficiënte manier voor het beheren van een SaaS-applicatie is het gebruik van multi-tenancy waarbij meerder tenants gebruik maken van dezelfde applicatie-instantie. Voor de beheerder van de applicatie is kostenefficiëntie een belangrijk aspect voor het bepalen van het aanbod en multi-tenancy kan hierbij een grote rol spelen. Tenants verwachten onderhoud en vernieuwing van de applicatie om mee te zijn met de recentste trends. De ontwikkelaar kan dit aanbieden a.d.h.v. continuous deployment, een strategie waarbij applicaties snel en frequent worden geüpdatet en beschikbaar worden gesteld aan de tenants. Om applicatieversies op regelmatige basis uit te rollen en ter beschikking te stellen aan de tenants kan er gebruik gemaakt worden van containertechnologie zoals Docker en Kubernetes die snel, licht en resource efficiënt zijn. Een Pod is een Kubernetes object dat bestaat uit één of meerdere containers en representeert een applicatie-instantie. Verder bezit Kubernetes functionaliteiten omtrent het samenplaatsen van Pods op dezelfde node, dit kan een interessante keuze zijn omdat containers met *shared libraries* onderling geheugen delen.

Tijdens continuous deployment van de applicatie is het belangrijk dat de tenant controle heeft over het upgradeproces en dat de SLA nageleefd wordt, hoge beschikbaarheid en service kwaliteit ingerekend. Een Pod verbruikt een bepaalde hoeveelheid resources om de applicatie op een correcte manier te laten werken. Wanneer een upgrade wordt uitgevoerd hoort de Pod over voldoende resources te beschikken om de nieuwe en oude tenants te behandelen volgens de SLA-voorwaarden. Om aan de hoge beschikbaarheid voorwaarde te voldoen tijdens een upgrade, wordt er vaak gebruik gemaakt van extra Pods om een *zero downtime* omgeving te produceren, dit wordt surge genoemd. Tenants die uit een versie migreren kunnen echter suboptimale Pods achterlaten die meer resources reserveren dan ze gebruiken. Om deze Pods optimaal te maken, moet er verticaal geschaald worden wat eveneens surge produceert. Verticaal schalen wijzigt de hoeveelheid resources van een Pod in tegenstelling tot het aantal Pods bij horizontaal schalen. Tijdens een upgrade moet er rekening gehouden worden met de hoeveelheid gereserveerde resources door de Pods en de surge om servicedegradatie te vermijden.

In deze thesis wordt er een middleware gepresenteerd voor het uitvoeren van verticale upgrades van multi-tenant SaaS-applicaties in Kubernetes met als doel om een kostenefficiënte oplossing te bieden zonder servicedegradatie. De verticale upgrades worden zonder servicedegradatie voorzien op basis van een hypothese omtrent surge. Deze stelt dat het gelijktijdig upgraden van tenants in een groep een lagere surgekost genereert dan het uitvoeren van een upgrade voor elke tenant individueel. Er werd uitgebreid onderzoek gevoerd naar de impact van surge tijdens een upgrade en er werd een upgradeproces gedefinieerd dat een *zero downtime* verticale upgrade van één of meerdere tenants voorziet. Er werd een prototype van de voorgestelde middleware ontwikkeld en geëvalueerd. De resultaten tonen aan dat de groepsmethode in totaliteit een lager resourcegebruik genereert dan de sequentiële methode en dit zonder inbreuk te plegen op de SLA.

Abstract

Software as a Service (SaaS) is a software architecture where applications are offered to customers over an online connection. Tenants, which is another name for customers, pay a certain price for use of this application and expect a certain level of service in return. A cost-efficient way of managing this type of application is multi-tenancy which uses the same application instance for multiple tenants. Cost-efficiency is an important factor for the administrator of the application as it determines the viability of the SaaS-offering and multi-tenancy provides this. Tenants expect maintenance and evolution of the application to keep up with the latest trends. The developer can provide this by using continuous deployment, this software deployment strategy allows fast and frequent updates and makes the application quickly available to the tenants. Rolling out these application versions can be done regularly by using container technology like Docker and Kubernetes which are fast, lightweight and resource efficient. Moreover, Kubernetes has functionalities around the concept of colocation of Pods on a node, this can be interesting because containers with shared libraries can share memory between themselves.

During continuous deployment of the application it's important that the tenant has control over the upgrade process and that the SLA is respected, high availability and service quality included. An application instance needs a certain amount of resources to function properly. When an upgrade is executed a Pod should have sufficient resources to serve old and new tenants according to the SLA-conditions. To provide high availability during upgrades, extra containers are used to realize a zero downtime upgrade, these extra containers are called surge. Tenants that migrate out of an application instance can leave suboptimal Pods which reserve more resources than they consume. To optimize these Pods, vertical scaling is used, this process also generates surge. Vertical scaling changes the amount of assigned resources to a Pod in contrast to the number of Pods with horizontal scaling. During an upgrade, the amount of resources reserved by the Pods and the amount of surge generated should be taken into account to avoid service degradation.

This thesis presents a middleware for execution of vertical upgrades for multi-tenant SaaS-applications in Kubernetes with the aim of providing a cost-efficient solution without service degradation. Vertical upgrades are performed without service degradation based on a hypothesis concerning surge which states that upgrading tenants simultaneously in a group generates less surge than executing upgrades for each tenant individually. Extensive research was performed regarding the impact of surge during an upgrade and an upgrade process was defined that provides zero downtime vertical upgrades of one or more tenants. A prototype of the proposed middleware was developed and evaluated. The results show that the group method produces a lower total resource usage than the sequential method without violating on the SLA.

I. Lijst van afkortingen en symbolen

SaaS	Software as a Service
aufs	Advanced multi-layered unification filesystem (Another Union File System)
GSL	GNU Scientific Library
CPU	Central Processing Unit
GB	Gigabyte
kB	Kilobyte
cgroups	Control Groups
CoW	Copy-on-Write
NIST	National Institute of Standard and Technology
SLA	Service Level Agreement
SOA	Service-oriented Architecture
m	Millicores

II. Lijst van figuren

III. Lijst van tabellen

Inhoudsopgave

1 Introductie

In dit hoofdstuk wordt het thesisonderwerp ingeleid. In het eerste onderdeel wordt de context rondom de thesis duidelijk gemaakt. Hierna volgt de probleemstelling en de doelen die bereikt horen te worden tegen het einde van deze thesis. Verder wordt het onderwerp gemotiveerd a.d.h.v. van een experiment en wordt de aanpak duidelijk gemaakt. Als laatste volgt er een tekstoverzicht dat de opbouw van de thesis verduidelijkt.

1.1 Context

Software is een belangrijk onderdeel van onze samenleving geworden, ieder van ons komt er bijna dagelijks mee in contact. De nood aan software zorgt er ook voor dat de applicatie ontwerpers hun bestaande programma's blijven verbeteren en oog hebben voor innovatie. Een modern paradigma in software design is dat van de online-applicatie. Dit is software waarnaar klanten, ook wel tenants genoemd, via het netwerk moeten communiceren. Naar de software kan door gebruikers verbonden worden a.d.h.v. zogenaamde *thin clients*, deze kunnen een browser of een lokale applicatie zijn. Langs de kant van de ontwikkelaar biedt zo een online-applicatie meer mogelijkheden op vlak van kostenefficiëntie, ontwikkeling en onderhoud aangezien op deze manier meerdere tenants gebruik kunnen maken van dezelfde applicatie-instantie. Dit type applicatie wordt ook wel een multi-tenant *Software as a Service* (SaaS) applicatie genoemd. [1]

Een proces dat overwegend voorkomt in deze online-applicaties is *continuous deployment*, [2] het wordt toegepast door ontwikkelaars en beheerders van een applicatie om zo snel en correct mogelijk nieuwe versies van de software beschikbaar te stellen aan de tenant. [3] Echter, een applicatie die typisch in *one-shot* een upgrade uitvoert, houdt geen rekening met tenant benodigdheden of eisen. [4] Het is belangrijk dat controle over deze upgrades niet enkel bij de SaaS-beheerder ligt maar dat deze ook beïnvloed kunnen worden door de tenant. Deze laatste heeft verder verwachtingen over de responsiviteit en beschikbaarheid van de applicatie en sluit een *Service Level Agreement* (SLA) af met de beheerder. In een SLA-contract worden afspraken tussen de tenant en beheerder vastgelegd om een welbepaald dienstniveau aan de tenant te leveren onder bepaalde voorwaarden. [5]

Een middel om continuous deployment te ondersteunen is het gebruik van containers waarin applicaties met al hun code en afhankelijkheden gebundeld worden. Verder zijn ze licht in vergelijking met virtuele machines en hierdoor starten ze sneller op, dit biedt de mogelijkheid voor snelle uitbreiding en verandering van de applicatie. [6] Op dit moment zijn er verscheidene containertechnologieën op de markt zoals Docker, CRI-O, LXC, containerd enzovoort. [7], [8], [9], [10]

Kubernetes is een container orchestratie platform dat het beheren van containers in een gedistribueerde omgeving ondersteunt. [11] Het biedt mogelijkheden aan tot het opleggen van restricties op resourcegebruik en bezit een uitgebreide functionaliteit voor het beheren van containers door zowel horizontale als verticale *autoscalers* en rolling upgrades. Het staat verder gebruik van Kubernetes-objecten toe voor het gemak van configuratie, zo is de Pod, die een eenheid van deployment voorstelt, de kleinst mogelijke uitvoerbare eenheid in Kubernetes.

Pods moeten altijd heropgestart worden om gewijzigde resources of een nieuwe image toegekend te krijgen. [12]

1.2 Probleemstelling

Om continuous deployment van een multi-tenant SaaS-applicatie te faciliteren zijn er overwegingen waar rekening mee gehouden moet worden. Het is belangrijk dat de tenants inspraak hebben in het upgradeproces en dat de SLA nageleefd wordt, voor de SaaS-beheerder staat de kostenefficiëntie van de operatie centraal.

Om te voldoen aan de SLA worden er tijdens upgrades altijd extra Pods voorzien om de nieuwe applicatie op te starten en operationeel te krijgen voordat de oude versie afgesloten wordt. In Kubernetes wordt dit extra aantal Pods surge genoemd, het is onvermijdbaar als hoge beschikbaarheid en optimaal resourcegebruik een significante rol spelen. Er moeten immers altijd voldoende Pods aanwezig zijn om servicedegradatie tegen te gaan en hoge beschikbaarheid te garanderen. Om te voldoen aan de beschikbaarheids-SLA moeten er minimaal N+1 Pods aanwezig zijn volgens het N+1 foutenmodel. [13] Verder zullen tenants die upgraden naar een nieuwe versie, Pods achterlaten die suboptimaal zijn. Een suboptimale Pod betekent in deze context dat een Pod meer resources reserveert dan nodig om aan de SLA te voldoen. Om deze Pods in optimale staat te krijgen moet er verticaal geschaald worden.

Echter, een probleem met verticaal schalen in Kubernetes is dat de containers altijd heropgestart moeten worden om een resourceconfiguratie aan te passen. Een upgrade die rekening houdt met de SLA zal hier ook surge genereren, maar indien er niet genoeg ruimte op de node is om deze extra Pod te plaatsen zal er servicedegradatie plaatsvinden.

1.3 Doelen

Het doel van de thesis is het ontwerpen van een upgrade planner voor multi-tenant SaaS-applicaties in Kubernetes. Deze voert upgrades uit door Pods verticaal te schalen en te voldoen aan *high availability* en *service continuity* (i.e. service op een continue kwaliteit) volgens de afgesproken SLA. De upgrades moeten rekening houden met de surge om een zero downtime omgeving te voorzien. Deze aspecten moeten verder op een zo optimaal mogelijke manier gebeuren zodat de beheerder kostenefficiënt te werk kan gaan, hiervoor wordt gebruik gemaakt van containers en moet surge maximaal benut worden.

Het doel is onderverdeeld in verschillende aspecten om te beantwoorden aan de noden van de tenant en de beheerder.

1. **Kostenefficiëntie**. Om kostenefficiëntie te bereiken wordt er de volgende hypothese voorgesteld: Het gelijktijdig upgraden van tenants in (grotere) groepen genereert een lagere kost van surge in vergelijking met het uitvoeren van een upgrade voor elke tenant individueel. Kubernetes wordt gebruikt om containers met gedeelde lagen samen te plaatsen om kostenefficiëntie in geheugen te behalen. [14]
2. **Upgrade planning**. Er wordt een upgrade planner ontworpen die het verticaal schalen van Pods faciliteert volgens bovenstaande kostenefficiëntie hypothese. Deze upgrade

planner moet verder de mogelijkheid geven aan tenants om aan te duiden wanneer een upgrade uitgevoerd moet worden.

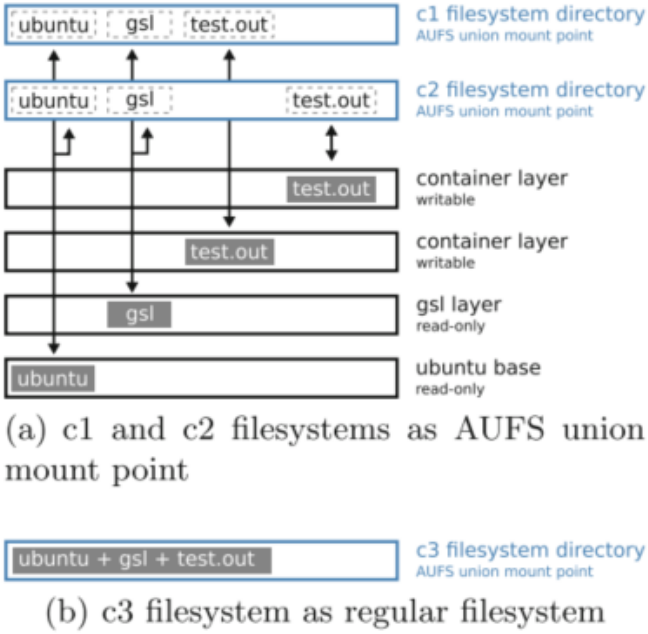
1.4 Motivatie

Aangezien Kubernetes steeds vereist dat een Pod opnieuw opgestart wordt is verticaal schalen niet altijd de beste optie. Horizontaal schalen is echter ook niet altijd een geldige optie. Dit is zeker zo bij *downscalen*. Er moeten immers steeds N+1 instanties van een bepaalde applicatie aanwezig zijn om hoge beschikbaarheid en fouttolerantie te garanderen. [13]

Het gebruik van containers vormt een kostenefficiënte oplossing voor continuous deployment door hun kleine images en snelheid omtrent het opstarten. Dit alleen maakt het een populair middel om continuous deployment te ondersteunen. Een ander voordeel is de mogelijkheid tot het delen van *shared libraries* in geheugen. In het artikel "*More Sharing, More Benefits? A Study of Library Sharing in Container-Based Infrastructures*" [14] wordt er een experiment uitgevoerd waarin wordt bewezen dat containers met overeenkomstige images geheugen delen tussen shared libraries. Het onderzoek werd uitgevoerd op een oudere containertechnologie LXC, het is echter interessant voor deze thesis want dit versterkt de nood om containers, die images hebben met shared libraries, op dezelfde node te plaatsen.

Het origineel experiment werd uitgevoerd op LXC containers en een aufs bestandslaag. Aufs is een implementatie van een *union filesystem*, en staat voor *advanced multi-layered unification filesystem*, waarbij een verenigd overzicht wordt gegeven van de bevattende bestanden. [15] In deze thesis wordt er gebruik gemaakt van Docker als containertechnologie en overlay2 [16] als modernere implementatie van aufs. Het doel van de reproductie is om te achterhalen of de functionaliteit ook aanwezig is in Docker en indien dit geldt, of deze ook aanwezig is in Kubernetes als er Pods gebruikt worden.

Het origineel experiment is als volgt geponeerd. Er zijn drie containers C1, C2 en C3 zoals getoond in Figuur 1. Voor C1 en C2 geldt dat hun bestandslagen een *aufs union mount point* zijn, bestaande uit verschillende lagen waaronder een shared library. C3 bestaat uit dezelfde componenten maar deze worden op een ext4 bestandssysteem geplaatst, dit is het standaard bestandssysteem voor meerdere Linux distributies. De resultaten wezen uit dat er tussen C1 en C2 libraries gedeeld werden terwijl C3 dit niet deed.



Figuur 1: Opstelling van het experiment (Overgenomen van Ferreira et al. [14])

De reproductie van dit experiment gebruikt een image die dezelfde lagen bevat als in het origineel experiment zijnde de GNU Scientific Library (GSL) en een Ubuntu basis. Zoals eerder al vermeld wordt er gebruik gemaakt van Docker en de Overlay2 *storage driver*. De geheugenmetingen gebeuren via /proc/<pid>/smaps, deze toon het private en gedeelde geheugen van het proces. In het eerste scenario wordt er één container opgestart en deze situatie vormt een referentie waarmee het volgend scenario vergeleken wordt. In het tweede scenario wordt een tweede identieke container opgestart. In Tabel 1 zijn de verwerkte metingen zichtbaar waarbij per proces het gedeelde en private geheugen getoond wordt in kilobyte (kB).

Tabel 1: Resultaten van reproductie voor Docker via smaps

	1 container		**2 containers**	
Proces	**Shared**	**Private**	**Shared**	**Private**
/a.out	0	12	4	8
/lib/x86_64-linux-gnu/libm-2.19.so	0	264	224	28
/lib/x86_64-linux-gnu/libc-2.19.so	904	64	924	24
/gnu/gsl/lib/libgslcblas.so.0.0.0	0	68	60	8
/gnu/gsl/lib/libgsl.so.19.3.0	0	492	396	84
/lib/x86_64-linux-gnu/ld-2.19.so	128	20	140	8
Totaal	**1032**	**920**	**1748**	**160**

Uit de vergelijking van de data blijkt dat in het tweede scenario meer geheugen gedeeld wordt dan in het eerste. Libraries die in het eerste scenario nog geen geheugen deelde zoals de GSL library, delen in het tweede scenario wel geheugen tussen de containers. Hieruit wordt afgeleid dat Docker net zoals LXC shared libraries in geheugen kan delen. In deze thesis wordt er niet enkel van Docker gebruik gemaakt, Kubernetes speelt ook een grote rol en voegt extra abstracties toe zoals Pods. Als volgt wordt dit experiment uitgevoerd in een Kubernetes omgeving, hier zal een Deployment gemaakt worden die enkel op één node draait. In het eerste scenario zal deze Pod maar één replica hebben en in het tweede scenario wordt er een tweede replica toegevoegd.

Tabel 2: Resultaten van reproductie voor Kubernetes via smaps

	1 Pod		**2 Pods**	
Proces	**Shared**	**Private**	**Shared**	**Private**
/a.out	0	12	4	8
/lib/x86_64-linux-gnu/libm-2.19.so	0	524	356	136
/lib/x86_64-linux-gnu/libc-2.19.so	956	136	1036	88
/gnu/gsl/lib/libgslcblas.so.0.0.0	0	72	64	8
/gnu/gsl/lib/libgsl.so.19.3.0	0	684	524	136
/lib/x86_64-linux-gnu/ld-2.19.so	140	8	140	8
Totaal	**1096**	**1436**	**2124**	**384**

Uit de analyse volgt dat er tussen de twee scenario's een uitgesproken verschil is in gedeeld geheugen zoals voorheen ook aangetoond in het Docker en origineel experiment. Dit resultaat bevestigt het voorgaand onderzoek en biedt de mogelijkheid om gebruik te maken van deze eigenschap.

1.5 Aanpak

De werkwijze van het onderzoek is gebaseerd op de volgende methodes:

- Literatuurstudie omtrent het upgraden van multi-tenant SaaS-applicaties en de vereisten voor dit soort middleware.
- Omzetten van de surge hypothese naar een kostenmodel om de belangrijkste metrieken te identificeren en deze laatste vervolgens omvormen naar een algoritme voor een maximaal aantal tenants in groep te migreren zonder de high availability en service continuity in het gedrang te brengen. Dit kostenmodel en algoritme worden in de rest van deze thesistekst de *groepsmethode* genoemd.
- Opstellen van een architectuur die gebruik maakt van Kubernetes om de doelstellingen van de thesis te bereiken. Er wordt ook een prototype van deze architectuur ontwikkeld om een evaluatie van de groepsmethode uit te kunnen voeren.

- Evaluatie van de groepsmethode a.d.h.v. experimenten met het prototype. Als base line wordt er tenant-per-tenant een upgrade uitgevoerd, terwijl de groepsmethode probeert maximale groepen van tenants samen te plaatsen om een upgrade uit te voeren.

1.6 Tekstoverzicht

De opbouw van de thesis is als volgt: In hoofdstuk 2 wordt er achtergrond informatie gegeven over multi-tenant SaaS-applicaties en Kubernetes. Vervolgens wordt in hoofdstuk 3 bestaand onderzoek besproken omtrent het faciliteren van upgrades in een multi-tenant SaaS-omgeving. Hoofdstuk 4 gaat over surge en definieert de groepsmethode voor het bepalen van maximale groepen van tenants. In het volgende hoofdstuk 5, wordt de architectuur van de middleware voorgesteld en besproken. Hierna volgt in hoofdstuk 6 de verwezenlijking van deze architectuur in een prototype en in hoofdstuk 7 wordt de evaluatie en resultaten besproken. Tot slot bevindt zich in hoofdstuk 8 de conclusie en een blik op het toekomstig werk omtrent dit onderwerp.

2 Achtergrond

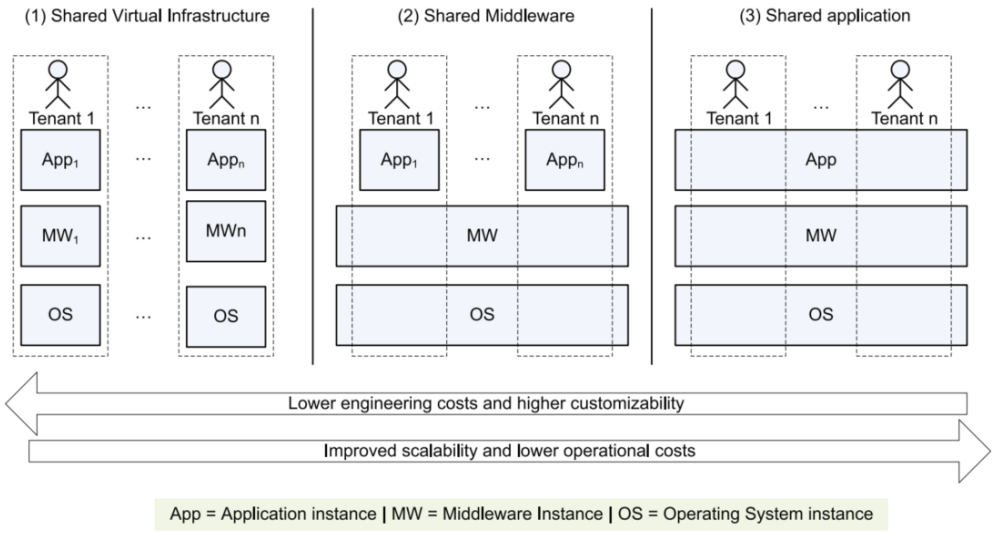
Dit hoofdstuk zal een overzicht aanbieden over de verschillende concepten en technologieën die aan bod komen in deze thesis. Er wordt eerst uitleg gegeven over wat een multi-tenant SaaS-applicatie is en zijn relevantie voor de thesis. Als volgende zal er dieper ingegaan worden op continuous deployment en wat het betekent voor multi-tenant SaaS-applicaties. Hierna zal er naar containers en Kubernetes gekeken worden, hoe werken deze technologieën en wat zijn de mogelijkheden. Als laatste worden verschillende deployment methodes uitgelegd en worden deze aan de thesis gekoppeld.

2.1 Multi-tenant SaaS-applicaties

“The capability provided to the consumer is to use the provider’s applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings.” [17]

Dit is de definitie van een SaaS-applicatie opgesteld door de National Institute of Standards and Technology (NIST). Het multi-tenant gedeelte van de term verwijst naar de verschillende klanten die de applicatie gebruiken, deze worden in het verder verloop van de thesis ook wel tenants genoemd.

Multi-tenant applicaties kunnen architecturaal op verschillende niveaus voorzien worden namelijk op het virtualisatie-, middleware- of applicatieniveau zoals zichtbaar in Figuur 2. Elke aanpak verschilt in het vooropgestelde doel en maakt opofferingen om bepaalde aspecten te faciliteren. Zo bieden multi-tenant applicaties op virtualisatieniveau een hoge aanpasbaarheid voor de tenant maar dit gaat ten koste van de schaalbaarheid van de applicatie. Applicatieniveau architectuur bevindt zich aan de andere kant van dit spectrum en biedt hoge schaalbaarheid voor weinig tenant-specifieke aanpassingen. [18] In deze thesis wordt er enkel gekeken naar multi-tenancy op applicatieniveau. Concreet betekent dit dat de applicatie-instantie gedeeld wordt door meerdere tenants en deze softwarematig gescheiden worden.



Figuur 2: Verschillende niveaus van multi-tenant applicaties (Overgenomen van Walraven et al. [18])

Eén van de grootste voordelen van dit type architectuur is dat de kosten enorm verlaagd kunnen worden aangezien de infrastructuur en software resources efficiënt per tenant verdeeld kunnen worden. Dit buit het *economies of scale* effect uit dat verwijst naar het delen van vaste kosten over alle tenants en hierdoor worden de kosten per tenant verlaagd. [1] Het onderhoud verloopt eenvoudiger omdat de upgrade van een applicatie globaal kan worden uitgevoerd. De complexiteit ligt echter in het aanbieden van op maat gemaakte aanpassingen aan de applicatie in zo'n omgeving. [18]

In een multi-tenant SaaS-applicatie worden verschillende stakeholders geïdentificeerd. De SaaS-ontwikkelaar en operator/beheerder, deze zijn bezig met de ontwikkeling en uitrol van het SaaS-aanbod. De tenant administrator en eindgebruikers worden geassocieerd met de tenant organisatie. [19] Dit soort applicaties kan een enorme hoeveelheid tenants bedienen, het configureren en onderhouden van al deze tenants kan een monumentale taak zijn. De tenant administrator vervult deze rol en beheert de configuratie van de tenant door gebruik te maken van een self-service interface. De functionaliteit van dergelijke interfaces varieert maar standaard omvat die meestal het toevoegen of aanpassen van gebruikers en gebruikersrollen. [1] Verder is het belangrijk om een onderscheid te maken tussen een tenant en een gebruiker. Een tenant is een entiteit zoals een bedrijf of een organisatie, het is een verzamelnaam voor een set van gebruikers die onderdeel uitmaken van deze entiteit.

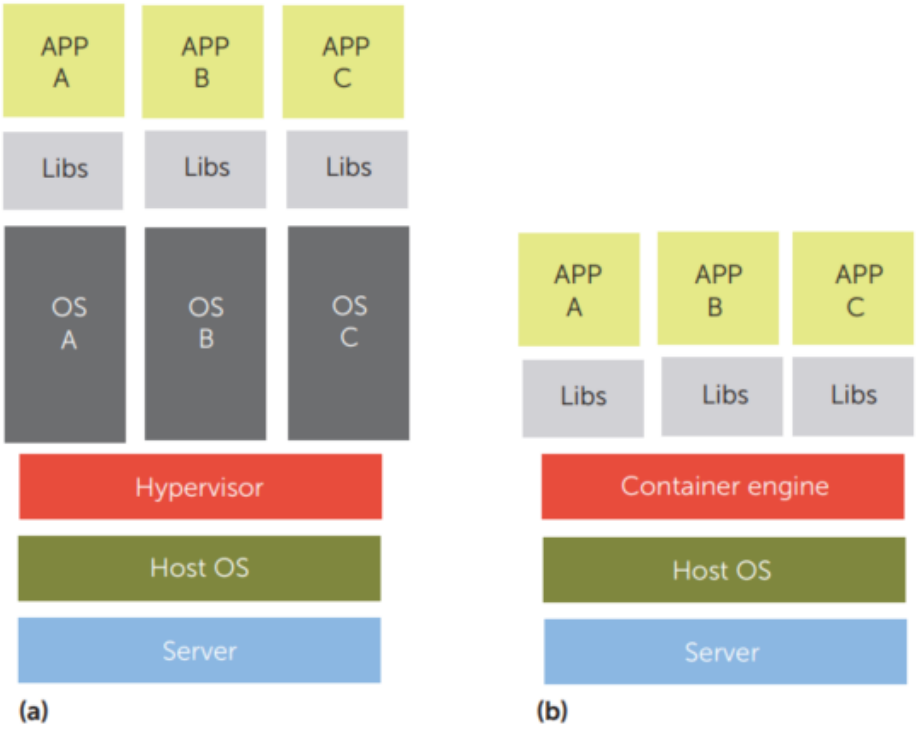
2.2 Continuous Deployment

De evolutie en onderhoud van multi-tenant SaaS-applicaties, verwijzend naar het upgraden en updaten, kan gebruik maken van een strategie genaamd continuous deployment. Dit wordt toegepast door de ontwikkelaar van de applicatie om zo snel en correct mogelijk nieuwe versies van de software beschikbaar te stellen aan de klant. [3] Bij continuous deployment wordt code automatisch gebouwd en getest, als deze testen slagen wordt de applicatie naar een productieomgeving uitgerold. De software wordt vaker geüpdatet via deze strategie en feedback van de klanten kan sneller toegepast worden. [20]

Bij continuous deployment voor multi-tenant SaaS-applicaties komen er enkele overwegingen aan bod namelijk (a) hoge beschikbaarheid en versie consistent gedrag, (b) service vermindering periodes en (c) SLA-naleving versus de aard van een upgrade. De eerste overweging verwijst hier naar de op voorhand afgesproken servicekwaliteiten d.m.v. een SLA-overeenkomst tussen de SaaS-beheerder en de tenants. Om beide eigenschappen te behouden tijdens een upgrade of update moeten lopende en nieuwe aanvragen van de tenants blijven behandeld worden. De volgende overweging, service vermindering periodes, kaart het probleem aan dat niet voor alle tenants dezelfde voorwaarden gelden voor toegelaten service vermindering. Hiermee wordt bedoeld dat alle tenants niet tijdens dezelfde tijdsloten service vermindering toestaan. In het geval van grotere veranderingen aan de applicatie hebben tenants ook verschillende tijdsperiodes waarin hun *business flow* aangepast wordt. Als laatste wordt de SLA-naleving tegen de aard van een upgrade aangehaald als overweging. Een te strikte naleving van een SLA is niet altijd haalbaar want een upgrade kan zeer ingrijpend zijn. Het kan verder ook onnodig zijn moest de tenant of SaaS-beheerder bereid zijn om de SLA tijdelijk op te offeren. [1]

2.3 Containers

Om continuous deployment te ondersteunen wordt er gebruik gemaakt van containers ook wel systeemniveau virtualisatie genoemd. Dit systeem is gebaseerd op een lichte virtualisatie techniek en laat het toe om een applicatie en zijn afhankelijkheden te bundelen in een container. Deze aanpak zorgt voor kleinere images, snellere opstarttijden en produceert in tegenstelling tot virtuele machines een hoger aantal applicatie-instanties op een node. Voor deze kleinere images is er echter een opoffering, containers delen de *kernel* met het besturingssysteem waarop ze draaien, dit limiteert ze in omgevingen en vormt een extra veiligheidsrisico. [6], [21] In Figuur 3 worden de lagen van een virtuele machine (a) en een container (b) getoond met het verschil zijnde de OS laag.



Figuur 3: Vergelijking tussen virtuele machines en containers (Overgenomen van bron [22])

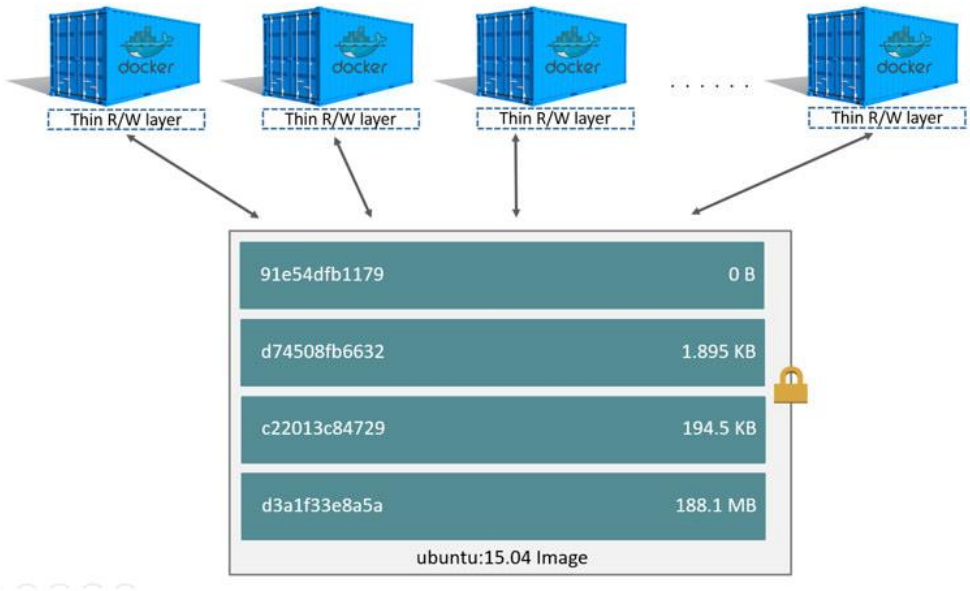
Containertechnologie wordt aangeboden door gebruik te maken van verscheidene Linux mechanismes om isolatie op verschillende niveaus te garanderen. *Namespaces* zorgen voor isolatie tussen de onderliggende host en de container om veiligheidsrisico's zoals *root access* te vermijden. Verder bieden *namespaces* nog andere functionaliteiten zoals het geven van een IP-adres aan elke container en het *mounten* van bestandssystemen zonder dat deze een impact op het hostsysteem heeft. [23] Control Groups (cgroups) is een *kernel* mechanisme voor het isoleren, limiteren en monitoren van resourcegebruik voor een groep van processen. Huidige containertechnologie is gebaseerd op dit mechanisme. [24]

Docker is één van de containertechnologieën die verder bouwt op deze basis. De Docker engine maakt gebruik van namespaces en cgroups om een basis container omgeving te voorzien maar breidt deze uit door gebruik te maken van Union File Systems om enkel de verschillen tussen container images op te slaan. Deze uitbreiding zorgt ervoor dat containers niet altijd hun volledige image opnieuw moeten opslaan maar enkel de lagen die verschillen met andere images. [25]

Een Docker image wordt opgebouwd uit een reeks van lagen, elke laag stelt een specifieke stap in het Docker image bouwproces voor. Elke laag in de image is *read-only* en dient als basis voor de creatie van een container. Het verschil tussen een container en een image is één extra *read/write* laag bovenop de *read-only* image, dit wordt geïllustreerd in Figuur 4.

Als containers opgestart worden op basis van images met identieke lagen dan dient Docker de laag maar éénmaal te voorzien en wordt deze hergebruikt voor alle containers met gemeenschappelijke lagen. Om zo efficiënt mogelijk gebruik te maken van opslagruimte wordt de Copy-on-Write (CoW) strategie toegepast. Deze strategie zorgt ervoor dat enkel bestanden die aangepast worden in de container naar de bovenste read/write laag gekopieerd worden. Als

dit niet het geval zou zijn, zou er tijdens elke activatie van een container een kopie van de hele image gemaakt moeten worden om aanpassingen toe te laten zonder dat het origineel gewijzigd wordt. De container-laag slaat geen persistente data op en bij vernietiging van de container gaan de bewerkte bestanden ook verloren tenzij er expliciet gebruik gemaakt wordt van middelen om deze op te slaan. [26]



Figuur 4: Docker Image-lagen en de container-laag (Overgenomen van bron [27])

Het delen van deze read-only lagen zorgt ervoor dat containers kleiner kunnen zijn en opslagruimte efficiënt gebruikt wordt. Vaak maken applicaties gebruik van shared libraries om delen van de code in geheugen te delen net zoals Docker opslagruimte via de lagen deelt. In sectie 1.4 werd bewezen dat Docker niet enkel de code deelt in opslagruimte maar ook het geheugen wordt gedeeld.

2.4 Kubernetes

Om containers op grote schaal te kunnen beheren kan er gebruik gemaakt worden van verschillende container-gebaseerde orchestratiesystemen zoals DC/OS, Docker Swarm en Kubernetes. [11] In deze thesis wordt enkel Kubernetes gebruikt en onderzocht.

Kubernetes is een container-gebaseerd orchestratiesysteem dat een container van het begin tot het einde van zijn levensduur zal beheren en blootstellen aan gebruikers. Het wordt gebruikt om de uitrol en het beheer van applicaties op grote schaal te vergemakkelijken. Continuous deployment van containers binnen Kubernetes wordt gefaciliteerd door de Kubernetes API die gebruik maakt van Kubernetes-objecten en hun bijhorende functionaliteiten. De belangrijkste van deze objecten voor dit thesisonderwerp worden vervolgens uitgelegd. [28]

Pod: Een Pod is de kleinst mogelijke uitvoerbare eenheid in Kubernetes en stelt een eenheid van *deployment* voor. Containers worden gedefinieerd in een Pod configuratie, dit kan een set van containers zijn. De containers die tot dezelfde Pod behoren worden samen op een node geplaatst en delen onderling opslagruimte en een IP-adres. [29] [12]

Replication controller/ReplicaSet: Een replication controller wordt gebruikt om het aantal draaiende Pods te beheren. Wanneer er meer of minder Pods aanwezig zijn dan gedefinieerd in de controller zal Kubernetes het aantal Pods verhogen of verlagen afhankelijk van het origineel aantal. [30] Een ReplicaSet beoefent dezelfde functionaliteit als een replication controller maar omvat meer methodes om Pods die bij de set horen te identificeren. [31]

Deployment: Een Deployment is een Kubernetes object dat een Pod en een ReplicaSet verenigt in één Kubernetes declaratie. Het is een aggregaat dat de mogelijkheid biedt tot het updaten, terugrollen en schalen van een Pod. Dit maakt het geschikt voor het beheer van applicaties en/of testomgevingen. [32] [33]

Labels: Een label is een *key-value* paar dat gebruikt wordt om Kubernetes-objecten te categoriseren. Het biedt de mogelijkheid om objecten te filteren en aan te spreken op basis van zelf-ingestelde waardes, elke sleutel kan echter wel maar éénmaal gedefinieerd worden per object. Om Kubernetes-objecten met elkaar te verbinden via labels wordt er gebruikt gemaakt van label selectors. Deze zullen zoeken naar Kubernetes-objecten met de geselecteerde labels, dit kan gebeuren a.d.h.v. (a) *equality-based selector*s of (b) *set-based selectors*. *Equality-based* maakt gebruik van 'gelijk aan' of 'niet gelijk aan' om een groep objecten te selecteren. *Set-based* geeft de mogelijkheid om op een set van objecten te filteren door gebruik te maken van de operatoren "in", "notin" en "exists". [34]

Requests/Limits: Requests en limits worden gebruikt om restricties op resourcegebruik van containers te leggen. Limits stelt zoals de naam impliceert de maximale limiet van een resource voor. Requests komt overeen met het gereserveerd aantal resources voor een container, deze waarde wordt gebruikt door de Kubernetes Scheduler om te beslissen of een Pod geplaatst kan worden op een node of niet. Deze beslissing is afhankelijk van de som van alle request waarden op de node. Er kan enkel een restrictie ingesteld worden op CPU en geheugen. [35]

Resources: In Kubernetes worden er verschillende resource types aangeboden die overeenkomen met computer resources. CPU wordt gerepresenteerd in Kubernetes CPU en komt overeen met 1 vCPU/Core bij Cloud leveranciers en 1 hyperthread voor bare-metal Intel processoren, verder kan het in millicores toegekend worden aan Kubernetes-objecten. Geheugen wordt uitgedrukt in bytes. Momenteel zijn dit de enigste resource types in Kubernetes, er zijn echter plannen om in toekomst meerdere types toe te voegen zoals *Network Bandwidth, Storage Space, …* [36] [37]

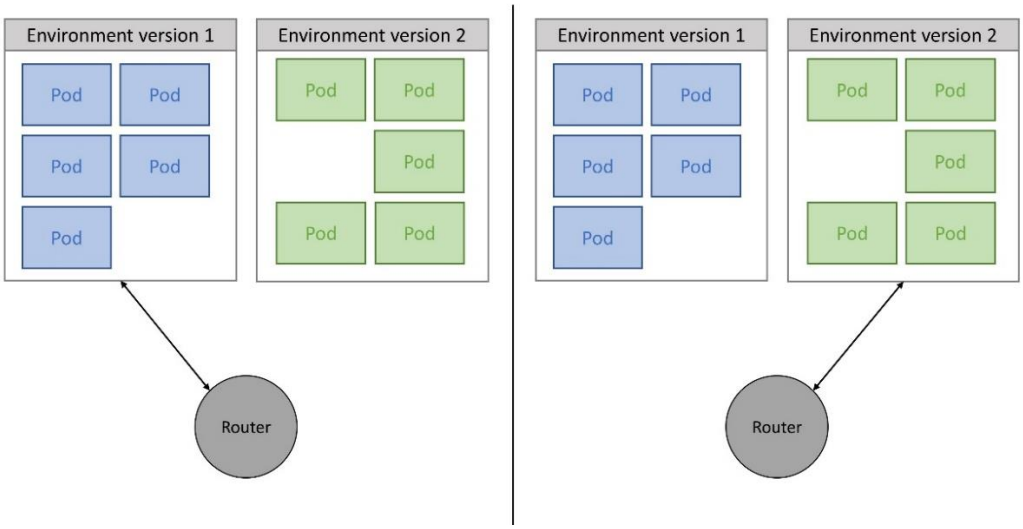
(Anti-)affinity: Colocatie en spreiding van Pods is een aspect dat al eerder aangehaald werd in deze thesis. Het kan voordeel bieden op vlak van geheugengebruik en beschikbaarheid. Dit gedrag kan bereikt worden door gebruik te maken van affinity en anti-affinity, het is een beperking die geplaatst kan worden tussen Pods en nodes en tussen Pods onderling. Concreet betekent dit dat affinity de mogelijkheid biedt aan Pods om aan te geven op welke nodes ze het liefst draaien en/of met welke Pods. Anti-affinity voorziet de omgekeerde werking door een voorkeur uit te drukken om een specifieke set Pods en/of nodes niet bij elkaar te plaatsen. Plaatsing van Pods wordt bepaald tijdens de *scheduling* fase, indien er op een later tijdstip een affinity regel verbroken wordt, zullen de Pods niet automatisch herplaatst worden. [38]

Readiness probes: Om zero downtime te garanderen tijdens updates van Pods wordt er gebruik gemaakt van Readiness Probes. Het zorgt ervoor dat Pods pas als *Ready* doorgegeven worden aan Kubernetes wanneer ze klaar zijn om verkeer te ontvangen. Zo wordt er vermeden dat er verkeer naar een Pod wordt verstuurd die deze nog niet kan afhandelen. [39]

2.5 Deployment Methodes

De multi-tenant applicatie en zijn verschillende versies wordt volgens *continuous deployment* aangeboden aan de tenants. Deployment van de applicatie-instantie wordt via Kubernetes en containers gefaciliteerd maar er zijn verschillende methodes om dit te doen zoals blue-green deployments en rolling upgrades.

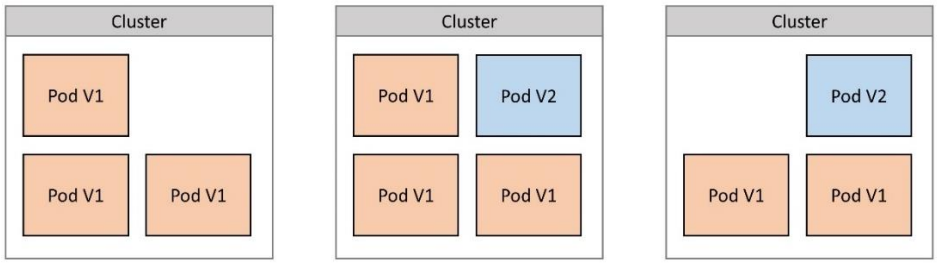
Blue-green deployments, zichtbaar in Figuur 5, maken gebruik van twee verschillende omgevingen om voor zero downtime upgrades te zorgen. De blauwe of ook wel huidige omgeving genoemd wordt door tenants gebruikt, de groene of nieuwe omgeving wordt opgezet om de nieuwe applicatieversie op te draaien. Zodra de groene omgeving klaar is om verkeer te ontvangen kunnen klanten omgeschakeld worden naar deze versie. [40]



Figuur 5: Blue-green deployments

Rolling updates/upgrades worden in Figuur 6 getoond en zorgen voor een graduele overgang van de oude naar de nieuwe applicatieversie. Het doel van deze upgrade-methode is net zoals bij blue-green deployments om zero downtime van de applicatie te garanderen. Om dit te bereiken zullen applicatie-instanties gradueel vervangen worden door de nieuwe versie door eerst de nieuwe instantie op te starten en hierna de oude instantie af te sluiten. Het aantal Pods dat vervangen wordt per rolling upgrade stap wordt bepaald door de beheerder van de SaaS-applicatie zodat er zich geen breuk van de SLA voordoet. [41], [42]

Een nadeel van rolling upgrades is het gevaar van compatibiliteitsproblemen tussen applicatieversies. Het aanpassen van database schema's of protocollen kan een impact hebben op andere versies van de software die op hetzelfde moment nog aanwezig zijn. [42] Verder kunnen er ook onzichtbare dependencies tussen versies verbroken worden zoals protocol verschillen, *dynamically linked libraries* en database schema's. Voor containers is het library probleem echter niet van toepassing omdat de libraries van de code in de container gebundeld worden. [41]



Figuur 6: Rolling updates/upgrades

De opstarttijd van applicatie-instanties en de overhead van de upgrade operatie kunnen voor een tijdelijk verlies van verwerkingscapaciteit zorgen. Hierdoor is er een gevaar voor prestatiedegradatie bij rolling upgrades. [43]

In deze thesis wordt er gebruik gemaakt van een omgeving waar slechts één Pod per applicatieversie aanwezig is. Een upgrade waarbij eerst de nieuwe versie opgezet wordt en vervolgens de oude instantie verwijderd wordt, komt overeen met zowel blue-green deployments als rolling upgrades in een enkele Pod omgeving. Er wordt echter gebruik gemaakt van Kubernetes en deze biedt een rolling upgrade functionaliteit aan om automatisch Pods volgens deze methode te upgraden wanneer een wijziging in Pod configuratie zich voordoet.

3 Bestaand onderzoek

Dit hoofdstuk werpt een blik op reeds bestaand onderzoek dat als inspiratie en ondersteuning dient voor dit thesisonderwerp. In sectie 3.1 wordt er gekeken naar eerder onderzoek omtrent de planning van upgrades in multi-tenant SaaS-applicaties en worden de verschillen met het thesisonderzoek geschetst.

3.1 Evolutie van Multi-tenant SaaS-applicaties

Multi-tenant SaaS-applicaties zijn voor de ontwikkelaar interessant door het voordeel van kostenvermindering. Dit wordt bereikt doordat resources efficiënt verdeeld worden over verschillende tenants en het *economies-of-scale* effect. Het brengt echter ook het probleem van isolatie tussen de tenants met zich mee en vermoeilijkt het upgradeproces als het vertrouwen van de tenants in het aanbod een primaire vereiste is. Een upgrade die impact heeft op alle tenants kan immers voor complicaties zorgen of tijdsverlies veroorzaken afhankelijk van de aard van de upgrade. Het is dus belangrijk dat de tenant een beslissingsrol heeft in de upgrade.

In de paper *"Continuous Evolution of Multi-tenant SaaS Applications: A Customizable Dynamic Adaptation Approach"* [4] werden initieel een aantal vereisten vastgesteld voor het aanpassen van upgrades op per-tenant basis.

- **Tenant upgrade isolatie:** zodat enkel tenants die willen upgraden dit ook doen.
- **Ondersteuning voor servicecontinuïteit:** hiervoor wordt een mechanisme voorzien dat de mogelijkheid biedt om een afweging te maken tussen *harvest* en *yield* [44] tijdens een upgrade.
- **Stakeholder control:** verschillende stakeholders hebben verschillende rollen tijdens dit proces
- **High automation:** hiermee ligt de focus op het verlagen van de operationele kosten van de applicatie.

Dit werk stelde verder een architectuur voor die voldoet aan deze vereisten. Dit kan echter niet geverifieerd worden aangezien er geen implementatie en evaluatie in de paper aanwezig zijn. De focus ligt dan eerder op het opstellen van de vereisten en het aanbieden van een benadering om alle stakeholders in het upgradeproces te integreren.

Een volgend artikel *"Middleware for Customizable Multi-staged Dynamic Upgrades of Multi-tenant SaaS Applications"* [45] past de vereisten aan maar komt op dezelfde concepten uit. In deze paper wordt er echter wel een implementatie en evaluatie voorzien van de voorgestelde middleware-oplossing. De evaluatie wordt uitgevoerd op basis van een document processing applicatie waarvoor verschillende upgrade scenario's met een verschillende impact op de applicatie opgesteld werden. Het prototype is gebouwd op basis van *Service Oriented Architecture* (SOA) die veelvoorkomend is in multi-tenant SaaS-applicaties en voorziet een component genaamd DSlookup die voor iedere tenant de correcte service teruggeeft wanneer deze een versie specifieke service aanroept. Als configuratiemogelijkheden heeft de tenant de keuze over de tijd van de upgrade activatie en het activatiemechanisme. Deze laatste bestaat uit primitieven die voor een compromis tussen yield en harvest [44] zorgen.

Het derde artikel "*Middleware for Dynamic Upgrade Activation and Compensations in Multi-tenant SaaS*" [19] bouwt verder op het voorgaand werk en zorgt als uitbreiding voor compensaties. Dit mechanisme biedt verschillende faciliteiten aan voor elke afweging tussen harvest en yield bestaande uit oplossingen die deze afwegingen compenseren voor de tenant. De compensaties bestaan net zoals bij de activatiescripts uit primitieven en biedt dus de mogelijkheid om extra gedrag te voorzien om servicedegradatie op te vangen.

Het doctoraal proefschrift van Fatih Gey [1] gaat dieper in op de uitdagingen en vereisten van dit type applicaties. Dit gebeurt op basis van een uitgebreide studie over de opportuniteiten van Software as a Service (SaaS). Verder wordt er onderzoek voorgelegd omtrent continuous delivery en wordt er een upgrade development proces opgesteld. De architectuur omvat *customization*, *upgrade enactment customization*, *compensations* en *reactive upgrade enactment.* Deze laatste is een zelf aanpassend systeem op basis van een *feedback-driven control loop* die gebruikersdata zal verzamelen en afhankelijk hiervan het overgangsgedrag aanpast. Verder wordt er een prototype opgesteld en een evaluatie uitgevoerd met Harvest en Yield als belangrijkste metrieken. Het proefschrift biedt een uitgebreide oplossing voor het plannen van upgrades op basis van een Service Oriented Architecture.

Discussie. Het onderzoek naar het ondersteunen van evolutie voor multi-tenant SaaS-applicaties is al redelijk uitgebreid. De voorgaande proefschriften leggen de focus op het behoud van vertrouwen van de tenant in het aanbod. Om dit te doen worden verscheidene faciliteiten geïntroduceerd zoals compensaties en *reactive upgrade enactment*. Deze componenten zijn ontworpen om afwegingen in *harvest* en *yield* te maken wat verder ook de belangrijkste metriek is van de evaluatie in het voorgaand onderzoek. Verder is de architectuur van de middlewarelaag ontworpen voor applicaties op basis van losstaande services en voert het een integrale rol uit door *user requests* te herleiden naar de juiste services. Op deze manier worden upgrades gelijkgesteld aan het verplaatsen van requests naar de nieuwe applicatieversie.

In deze thesis heeft de middlewarelaag geen controle over de tenant *requests*, terwijl SLA-naleving een integraal onderdeel van de thesis is, worden er geen extra faciliteiten voorzien om servicedegradatie op te vangen. Het doel is het ontwikkelen van een kostenefficiënte manier om applicatie-upgrades uit te voeren die geen SLA-breuk produceert. Hierdoor is manipulatie van de *user requests* niet nodig en kan de middleware los van de applicatie uitgerold worden.

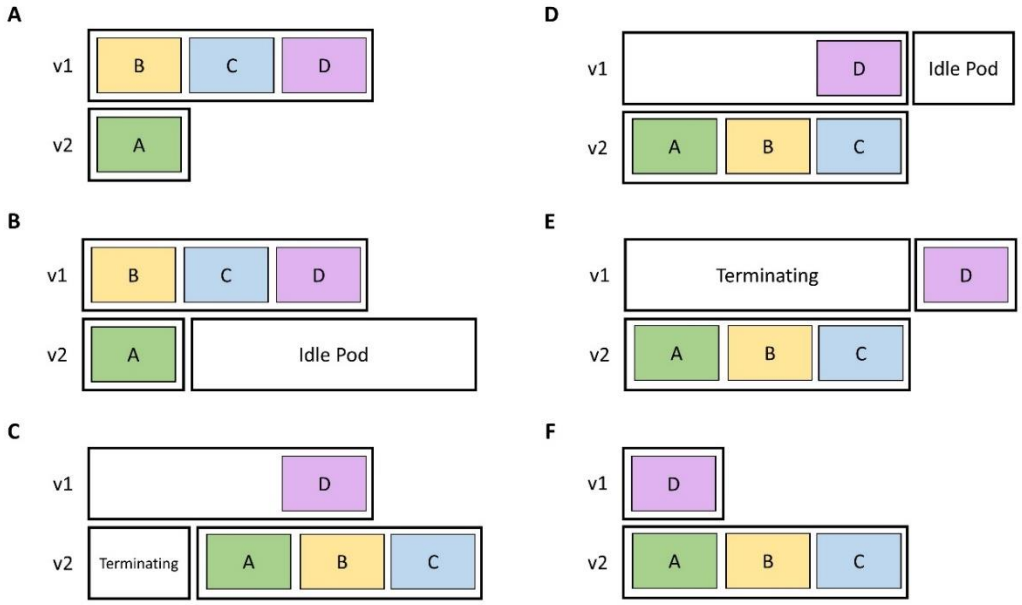
4 Groepsmethode: kostmodel en algoritme

In dit onderdeel wordt er dieper ingegaan op het concept van surge. Wanneer een upgrade uitgevoerd wordt, zullen er extra Pods opgestart worden om downtime te vermijden, deze bijkomend Pods worden als surge gedefinieerd tijdens het upgradeproces. Elke Pod komt overeen met een hoeveelheid gereserveerde resources. In dit hoofdstuk worden er algoritmes en een kostenmodel opgesteld om de hoeveelheid gereserveerde resources te minimaliseren. In deze thesis wordt enkel rekening gehouden met de CPU-resources.

Het upgradeproces voor deze thesis wordt geïntroduceerd in sectie 4.1 en maakt gebruik van rolling upgrades om multi-tenant applicaties zonder downtime te upgraden. Hierna wordt er in sectie 4.2 een kostenmodel opgesteld om de primaire CPU-kosten van surge te bepalen. Als volgende worden in sectie 4.3 de algoritmes voorgesteld die gebruikt kunnen worden om het optimaal aantal tenants voor de upgrade te bepalen. Afsluitend wordt er in sectie 4.4 een simulatie uitgevoerd die een voorbeeld geeft van de algoritmes en hoe deze gebruikt kunnen worden.

4.1 Upgradeproces

Rolling upgrades voeren gradueel een upgrade uit door eerst een nieuwe Pod te voorzien alvorens de oude wordt afgesloten om zo zero downtime en minimale servicedegradatie te bereiken. Om multi-tenant applicaties naar containers om te zetten wordt elke tenant bekeken als een vaste hoeveelheid resources in de Pod, wat ervoor zorgt dat de Pod lineair verandert in zijn hoeveelheid resources. Wanneer een rolling upgrade wordt dan uitgevoerd op de Pod resources om te herschalen met zero downtime.



Figuur 7: Upgradeproces voor multi-tenant applicaties

Het volledige upgradeproces in deze thesis bestaat uit zes stappen zoals aangetoond in Figuur 7. Deze figuur is verder onderverdeeld in twee grotere stappen die elk een rolling upgradeproces doorlopen. Stappen A, B en C behoren tot de *ScaleUp* fase waarin de nieuwere versie meer resources toegekend krijgt. Stappen D, E en F behoren tot de *ScaleDown* fase van het upgradeproces waarin de Pod van de oude versie minder resources krijgt. Deze twee fases zorgen ervoor dat de Pods altijd een optimaal aantal resources hebben om de hoeveelheid tenants waarover ze beschikken te bedienen.

De initiële stap A is de startsituatie waarin de applicatieversies zich bevinden net voordat het upgradeproces van start gaat. De volgende stap B voegt een extra Pod toe van de nieuwe versie waarin genoeg resources worden voorzien om het origineel aantal tenants en de nieuwe migrerende tenants in op te vangen. Deze extra Pod is de eerste kennismaking met surge in het upgradeproces, zoals eerder vermeld wordt deze opgestart om ervoor te zorgen dat er geen downtime is voor de tenants die gebruik maken van de nieuwe applicatieversie. Stap C is de laatste stap van de *ScaleUp* fase en introduceert geen nieuwe surge maar behoudt dezelfde situatie als in stap B. Tijdens deze stap is de nieuwe applicatie-instantie klaar om verkeer te ontvangen en zullen tenants hun nieuwe verbindingen naar deze instantie verzonden worden, de initiële Pod wordt ondertussen afgesloten. Nadat de Pod is afgesloten komt de *ScaleUp* fase ten einde en kunnen de migrerende tenants gebruik maken van de nieuwe applicatieversie zonder dat dit een impact heeft op de andere tenants. Er is echter nog het probleem van de suboptimale Pod in de originele versie die meer ruimte opeist dan er door de tenants gebruikt zal worden.

Dit wordt opgelost door de *ScaleDown* fase waar deze Pod geoptimaliseerd zal worden voor het aantal tenants dat het bevat. De stappen D, E en F verlopen identiek aan de eerste drie stappen maar dan op de Pod van de originele applicatieversie, ze verlopen dus ook volgens het rolling upgrade model om downtime te vermijden voor de tenants die overblijven in de originele versie. Wanneer stap F bereikt wordt is het upgradeproces voltooid en kunnen nieuwe upgrades plaatsvinden.

Uit Figuur 7 is het duidelijk dat er vier stappen (B, C, D, F) zijn waarin er een verhoogd aantal Pods aanwezig is. Stappen B en C hebben identiek dezelfde hoeveelheid surge, hetzelfde geldt voor stappen D en F. Deze vier stappen zullen dus een bepalende factor vormen om te beslissen of een upgrade volgens de bovenstaande stappen mogelijk is of niet.

4.2 Kostenmodel

In de vorige sectie werden de stappen met surge geïdentificeerd. Het doel van deze sectie is het opstellen van een kostenmodel dat gebruikt wordt om de metrieken te bepalen die impactvol zijn om surgekost te kunnen minimaliseren. De formules die hier uit voortvloeien kunnen gebruikt worden om de gestelde hypothese te valideren en algoritmes te definiëren die bepalen of een upgrade mogelijk is en met hoeveel tenants deze uitgevoerd kan worden.

Deze hypothese is als volgt: Rolling upgrades produceren surge om zero downtime te garanderen, dit is onafhankelijk van het aantal tenants dat upgradet naar de nieuwe versie. Hieruit volgt de assumptie dat het gelijktijdig upgraden van meerdere tenants in één upgrade zal leiden tot een lagere totale kost in vergelijking met het individueel tenant per tenant uitvoeren van een upgrade.

De kosten kunnen onderverdeeld worden in requestkost, zijnde het aantal resources dat tijdens een upgrade gereserveerd wordt en de gebruikskost, het aantal resources dat gebruikt wordt tijdens een upgrade. De formules in de volgende onderdelen zijn opgesteld voor een omgeving met één node.

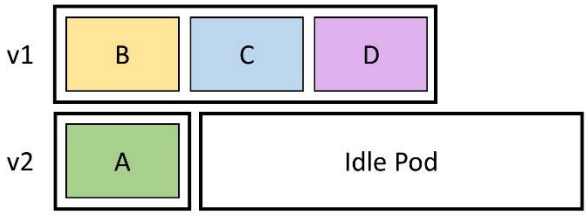
4.2.1 Requestkost

De requestkost is de totale kost van de gevraagde resources tijdens de verschillende fases van de upgrade. Zoals hierboven reeds aangestipt, wordt CPU als de primaire gestreste resource voor de applicatie genomen en worden secundaire gestreste resources buiten beschouwing gelaten. De multi-tenancy aard van de applicatie vraagt een CPU-waarde per tenant om de kost te berekenen, in deze thesis wordt de assumptie gemaakt dat dit een constante waarde is. De requestkost komt overeen met het gereserveerd aantal resources dat een Pod aanvraagt, dit vormt een limiterende factor binnen Kubernetes want als deze groter is dan de beschikbare ruimte op de node kan een Pod niet geplaatst worden. [46]

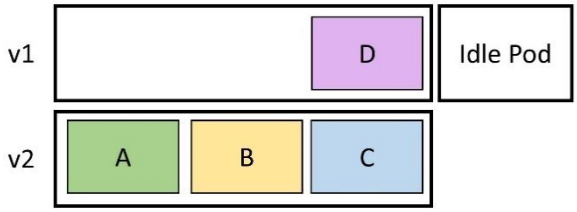
Het bepalen van de constante waarde die overeenkomt met de per tenant CPU-waarde gebeurt op basis van de afgesproken SLA. Deze zal beschikken over een minimum *latency* in ms waaraan de applicatie moet voldoen voor een afgesproken aantal gebruikers per seconde (users/s)[1]. Concreet komt dit neer op een overeenkomst van de vorm X latency voor Y users/s. Op basis van de afgesproken latency kan een resourcematrix opgesteld worden die verschillende limieten voor een verscheiden aantal user/s bevat. Deze matrix wordt gebruik om een maximaal aantal user/s per resourceconfiguratie te bepalen en dit kan omgevormd worden naar een per tenant constante waarde die voor elk mogelijk aantal user/s voldoet aan de SLA. De request waarde is gelijk aan het gemiddeld gebruik van de tenants en wordt in de rest van de thesis op 2/3 van de limiet geplaatst. Dit produceert een lineaire benadering van de reële oplossing.

De eerste stap van het upgradeproces is de startsituatie waarin de applicatie zich bevindt net voor de upgrade, de variabelen die gebruikt worden in de volgende formules zijn te vinden in Tabel 3. Tijdens de eerste stap bevinden de meeste tenants zich in applicatieversie *p* en zullen ze willen upgraden naar versie *p+i* waarbij *i* een positief geheel getal is. De volgende stap, zichtbaar in Figuur 8, omvat het vergroten van de nieuwe applicatieversie *p+i* en deze Pod is de surge die voorkomt in deze fase. De tweede fase waar surge een impact heeft is tijdens het optimaliseren van de suboptimale Pod van de oude applicatieversie zoals zichtbaar in Figuur 9.

[1] We veronderstellen dat user/s = request/s door de denktijd van elke gebruiker tussen 2 requests gelijk te stellen aan 1000 ms.



Figuur 8: Update applicatieversie p+i



Figuur 9: Versie p optimaliseren

De volgende formules representeren deze stappen en gelden voor één iteratie van het upgradeproces, na deze iteratie horen variabele parameters, zoals het aantal tenants in de elke versie, herbekeken te worden. Laat T_{Vp} het aantal tenants voor een applicatieversie *p* zijn op het begin van het upgradeproces, n is het aantal tenants dat upgrade naar de nieuwe applicatieversie, R_{Vp} de CPU-request waarde per tenant van applicatieversie *p*, $RC_{Vp+i}(n)$ is de totale requestkost tijdens een upgrade naar versie *p+i* en $RC_{Vp}(n)$ is de totale requestkost van de optimalisatie van versie *p.*

$$RC_{Vp+i}(n) = \left(R_{Vp} * T_{Vp}\right) + \left(R_{Vp+i} * T_{Vp+i}\right) + \left(R_{Vp+i} * (T_{Vp+i} + n)\right)$$

$$RC_{Vp}(n) = \left(R_{Vp} * T_{Vp}\right) + \left(R_{Vp+i} * (T_{Vp+i} + n)\right) + \left(R_{Vp} * (T_{Vp} - n)\right)$$

Deze formules komen overeen met de stappen van een rolling upgrade waar surge gegenereerd wordt. De eerste formule hoort bij de *ScaleUp* fase en representeert stap B in Figuur 7 waarin de nieuwe versie vergroot wordt. De tweede formule komt overeen met de *ScaleDown* fase en stap D in Figuur 7.

Wanneer het aantal migrerende tenants (n) gelijk is aan 1 wordt een groep van één tenant geüpgraded, dit komt overeen met de sequentiële methode en betekend dat de request van beide methodes identiek zijn wanneer het aantal migrerende tenants op zijn laagst is. Om de n-de iteratie, gelijk aan de n-de upgrade van uit de startwaarden, van de sequentiële methode te berekenen worden de volgende formules gebruikt:

$$RC_{Vp+i}(n) = \left(R_{Vp} * (T_{Vp} - (n-1))\right) + \left(R_{Vp+i} * (T_{Vp+i} + (n-1))\right) + \left(R_{Vp+i} * (T_{Vp+i} + n)\right)$$

$$RC_{Vp}(n) = \left(R_{Vp} * (T_{Vp} - (n-1))\right) + \left(R_{Vp+i} * (T_{Vp+i} + n)\right) + \left(R_{Vp} * (T_{Vp} - n)\right)$$

Een beperking voor de requestkost die zich voordoet tijdens de rolling upgrade is dat de hoeveelheid beschikbare CPU van de node groter moet zijn dan de hoogste waarde van deze

twee formules wanneer één tenant upgradet. De beschikbare capaciteit van een node komt overeen met de som van de niet-gereserveerde CPU en de request waardes van de applicaties versies die de upgrade uitvoeren.

$$CPU_{node} > \max\left(RC_{Vp+i}(1), RC_{Vp}(1)\right)$$

$$CPU_{node} = CPU_{free} + \left(R_{Vp} * T_{Vp}\right) + \left(R_{Vp+i} * T_{Vp+i}\right)$$

Deze beperking is een belangrijke limiet voor een upgrade van deze aard en het bepaalt het maximaal aantal tenants dat gebruik kan maken van de applicatie in een N+1 fouttolerantie omgeving. De impact van de verschillende variabelen ligt bij beide formules op een andere plaats. Om dit duidelijk te maken zijn de voorgaande formules herleid tot de volgende vorm:

Groepsmethode:

$$RC_{Vp+i}(n) = \left(R_{Vp} * T_{Vp}\right) + 2 * \left(R_{Vp+i} * T_{Vp+i}\right) + \left(R_{Vp+i} * n\right)$$

$$RC_{Vp}(n) = 2 * \left(R_{Vp} * T_{Vp}\right) + \left(R_{Vp+i} * T_{Vp+i}\right) + \left(R_{Vp+i} * n\right) - \left(R_{Vp} * n\right)$$

Sequentiële methode:

$$RC_{Vp+i}(n) = \left(R_{Vp} * T_{Vp}\right) + 2 * \left(R_{Vp+i} * T_{Vp+i}\right) + \left(R_{Vp+i} * n\right) + \left(\left(R_{Vp+i} - R_{Vp}\right) * (n-1)\right)$$

$$RC_{Vp}(n) = 2 * \left(R_{Vp} * T_{Vp}\right) + \left(R_{Vp+i} * T_{Vp+i}\right) + \left(R_{Vp+i} * n\right) - \left(R_{Vp} * n\right) - \left(R_{Vp} * (n-1)\right)$$

Analyse. Onder de assumptie dat een applicatie een stijgende request waarde krijgt naarmate deze evolueert door de toevoeging van extra functionaliteiten wordt de volgende analyse opgesteld van de bovenstaande formules. Bij de eerste fase (*ScaleUp*) ligt de grootste impact bij T_{Vp+i}, de sequentiële methode produceert bij elke iteratie een hogere kost t.o.v. de groepsmethode doordat T_{Vp+i} elke iteratie stijgt. De waarde van deze stijging is gelijk aan het verschil tussen R_{Vp+i} en R_{Vp} en zorgt ervoor dat, in de sequentiële methode, de CPU_{node} waarde sneller en bij een lager aantal tenants bereikt zal worden.

Onder dezelfde assumpties als voorheen ligt bij de tweede fase (*ScaleDown*) de grootste impact bij T_{Vp} en de sequentiële methode produceert per iteratie een lagere waarde voor $RC_{Vp}(n)$ doordat T_{Vp} daalt bij elke sequentiële iteratie. De formule $RC_{Vp}(n)$ produceert wanneer $R_{Vp} = R_{Vp+i}$ een constante waarde voor de groepsmethode, dit komt omdat de twee berekeningen waar de variabele n een impact heeft in deze formule elkaar opheffen in dit geval.

4.2.2 Gebruikskost

De gebruikskost is de opsomming van de gebruikte resources tijdens het upgradeproces, voor deze kost zijn tijd en CPU de grote drijfveren. De hoeveelheid CPU die gebruikt wordt is afhankelijk van de werklading die de tenants uitoefenen op de applicatie tijdens de upgrade.

De totale gebruikskost is in tegenstelling tot de requestkost wel afhankelijk van alle vier de intermediaire fases van het upgradeproces, zijnde B, C, D en E in Figuur 7. Voor de volgende formules geldt dat UT_{Vp} de tijd is die een applicatie nodig heeft om van versie *p* naar *p+i* te

upgraden en ST is de tijd die nodig is om tenants van applicatieversies te wisselen. Deze laatste zal vaak een constante waarde zijn die afhankelijk is van de tijd nodig om alle belangrijke services op de hoogte te brengen van de wijziging. U_{ten} is de hoeveelheid CPU gebruikt per tenant en U_{idle} de hoeveelheid door een Pod zonder tenants, UC_{Vp} komt overeen met de totale gebruikskost geproduceerd tijdens de aanpassing van de Pod behorende tot applicatieversie *p*, UCS_{Vp+i} representeert de gebruikskost geproduceerd tijdens het wisselen van tenants naar applicatieversie *p+i*. De totale resource kost is gelijk aan de som van de onderstaande formules, die respectievelijk overheen komen met stap B, C, D en E.

$$UC_{Vp+i}(UT_{Vp+i}) = UT_{Vp+i} * \left[\left(U_{ten} * T_{Vp}\right) + \left(U_{ten} * T_{Vp+i}\right) + U_{idle}\right]$$

$$UCS_{Vp+i}(n, ST) = ST * \left[\left(U_{ten} * \left(T_{Vp} - n\right)\right) + \left(U_{ten} * \left(T_{Vp+i} + n\right)\right) + U_{idle}\right]$$

$$UC_{Vp}(n, UT_{Vp}) = UT_{Vp} * \left[\left(U_{ten} * \left(T_{Vp} - n\right)\right) + \left(U_{ten} * \left(T_{Vp+i} + n\right)\right) + U_{idle}\right]$$

$$UCS_{Vp}(n, ST) = ST * \left[\left(U_{ten} * \left(T_{Vp} - n\right)\right) + \left(U_{ten} * \left(T_{Vp+i} + n\right)\right) + U_{idle}\right]$$

Tijd speelt hier een grote factor om de kost van het gebruik te meten. Een upgrade die langer duurt zal een veel grotere gebruikskost produceren. Omdat een sequentiële upgrade overeenkomt met een groepsupgrade van één tenant zal de tijd en het aantal tenants dat upgrade de beslissende factor zijn. Beide methodes zullen op dezelfde waarde starten maar de sequentiële methode zal al snel lineair stijgen in kost door de extra iteraties die uitgevoerd moeten worden. Dit ondersteunt de assumptie gemaakt in de thesis zijnde dat een groepsupgrade minder kosten produceert dan een sequentiële upgrade.

Tabel 3: Symbolen formules

Variabelen	**Uitleg**
R_{Vp}	De hoeveelheid CPU-resources gereserveerd per tenant om te voldoen aan de SLA voor applicatieversie *p*. Dit komt overeen met het Request-veld in Kubernetes configuratiebestand.
T_{Vp}	Het aantal tenants dat door versie *p* van de applicatie behandeld wordt.
U_{idle}	De hoeveelheid gebruikte CPU-resources voor een Pod zonder tenants.
U_{ten}	De hoeveelheid gebruikte CPU-resources per tenant.
UT_{Vp}	De tijd die nodig is om applicatieversie *p* te updaten. Concreet komt dit overeen met de tijd die nodig is om een Pod aan te passen en operationeel te krijgen.
ST	De tijd die nodig is om tenants van Pod te laten wisselen om zero downtime te garanderen.

4.3 Algoritme

Het kostenmodel heeft de impactvolle parameters in de formules geïdentificeerd. In dit onderdeel worden algoritmes opgesteld die gebruikt worden om te bepalen of een omgeving geschikt is om upgrades uit te voeren volgens dit model en wat de maximale grootte van een groep is voor de huidige parameters. Dit onderdeel zal beide doelen bespreken, de upgrade omgeving in sectie 4.3.1 en optimale groepsbepaling in sectie 4.3.2.

4.3.1 Upgrade omgeving

Het eerste doel van het algoritme is valideren of een upgrade mogelijk is in een omgeving op basis van de parameters. De bovenstaande request formules vormen het fundament voor het opzetten van meerdere omgevingen en deze te kunnen verifiëren. De minimale vereiste voor de uitvoer van een upgrade volgens dit model is het upgraden van één tenant. Hiervoor geldt dat de requestkost van de formules kleiner moet zijn dan de beschikbare hoeveelheid CPU op de node.

In de volgende formules geldt dat de waardes R_{Vp} en R_{Vp+i} op voorhand bepaald zijn afhankelijk van de applicatie en genoteerd worden als C1 en C2. De hoeveelheid tenants per versie T_{Vp} en T_{Vp+i} zijn respectievelijk x en y, en CPU_{node} is de beschikbare hoeveelheid CPU op de node. Voor de functie $RC_{Vp}(x)$ wordt er onderscheid gemaakt tussen het geval waar C1 = C2 en C1 ≠ C2.

$$RC_{Vp+i}(1) = (C1 * x) + 2 * (C2 * y) + (C2 * 1) \leq CPU_{node}$$

$$Geval\ C1 = C2;\ RC_{Vp}(1) = 2 * (C1 * x) + (C2 * y) \leq CPU_{node}$$

$$Geval\ C1\ \neq C2;\ RC_{Vp}(1) = 2 * (C1 * x) + (C2 * y) + (C2 * 1) - (C1 * 1) \leq CPU_{node}$$

Deze set van lineaire vergelijkingen vormt een beperking, als deze voldaan zijn, betekent het dat zeker één tenant upgrade uitgevoerd kan worden in de omgeving. Echter dit is geen garantie dat alle tenants in deze omgeving gaan kunnen upgraden rekening houdend met de toegevoegde surge.

Indien deze beperkingen niet voldaan zijn, kan een upgrade niet uitgevoerd worden in de huidige omgeving en kan een optimale omgeving bepaald worden. Deze omgeving wordt opgesteld met het doel om zoveel mogelijk tenants van versie *p* naar versie *p+i* te upgraden. Dit kan gedefinieerd worden door de objectief functie:

$$f(x, y) = x + 0y$$

Om de x-waarde te bepalen worden de formules omgevormd. Net zoals eerder wordt er hier onderscheid gemaakt tussen wanneer C1 = C2 en wanneer niet. Bij de *ScaleDown* fase correspondeert de eerste formule met de situatie waarbij C1 = C2 en de tweede met C1 ≠ C2.

$$Fase\ ScaleUp = x\ \leq \frac{CPU_{node} - \mathrm{C2x} - (2 * \mathrm{C2y})}{C1}$$

$$Fase\ ScaleDown = x\ \leq \frac{CPU_{node} - C2y}{2C1}; x\ \leq \frac{CPU_{node} - C2 + C1 - C2y}{2C1}$$

Deze formules produceren altijd een omgeving waarvoor de ingevoerde parameters geldt dat alle tenants geüpgraded kunnen worden in één groep.

4.3.2 Optimale groepen

Wanneer een omgeving voldoet aan de eerste voorwaarde, zijnde dat een upgrade van één tenant mogelijk is, kan er vervolgens bepaald worden hoeveel tenants in een groep maximaal geüpdatet kunnen worden. In de volgende formules is *n* het aantal tenants dat in één keer gemigreerd kan worden, x is het aantal tenants van versie *p* en y het aantal tenants van versie *p+i*. De eerste formule waarbij C1 = C2 produceert een constante waarde en wordt niet beïnvloed door het aantal tenants dat zich verplaatst van versie. Voor deze formule geldt enkel dat de waarde groter moet zijn dan of gelijk is aan 0 om een upgrade te kunnen uitvoeren.

$$Fase\ ScaleUp = n\ \leq \frac{CPU_{node} - \mathrm{C1x} - (2 * \mathrm{C2y})}{C2}$$

$$Fase\ ScaleDown = 0 \leq\ CPU_{node} - 2C1x\ ; n \leq\ \frac{CPU_{node} - (2 * C1x) - C2y}{(C2 - C1)}$$

De formules produceren het maximaal aantal tenants per fase en per upgradeproces iteratie. Wanneer de waardes van de fases verschillen is de kleinste waarde degene die overeenkomt met het maximum aantal tenants. Deze fase zal meer surge produceren en is de limiterende factor.

4.4 Simulatie

Een simulatie van de bovenstaande formules wordt gebruikt om een voorbeeldsituatie weer te geven. Voor deze simulatie wordt er gebruik gemaakt van realistische waarden, er geldt dat C1 en C2 respectievelijk gelijk zijn aan 65 en 90 CPU-millicores. De CPU_{node} is de maximale beschikbare ruimte voor een node, hier wordt op een node gerekend die enkel de applicatie bedient maar mogelijk andere Pods heeft draaien voor de werking van de cluster. Voor deze simulatie wordt er een capaciteit van 1750 millicores genomen.

$$RC_{Vp+i}(x) = (65 * x) + 2 * (90 * y) + 90x \leq 1750$$

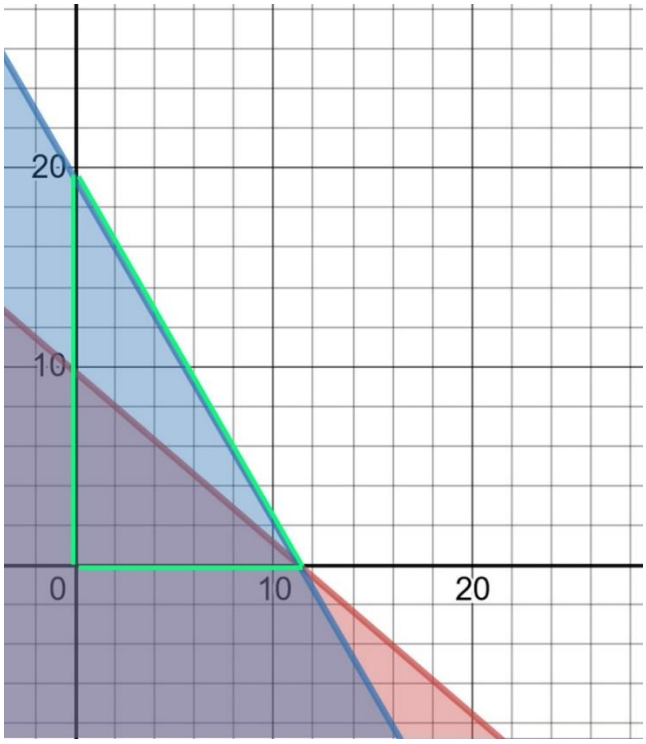
$$RC_{Vp}(x) = 2 * (65 * x) + (90 * y) + 90x - 65x \leq 1750$$

Samen leveren deze formules een grafiek op met een polytoop waarin een gebied bestaat met alle mogelijke oplossingen zoals zichtbaar op Figuur 10. Op basis van de grafiek kan dan de meeste optimale oplossing bepaald worden door de coördinaten van de hoekpunten op de figuur in te geven in de objectief functie. Omdat, in ons geval, de interesse ligt bij een situatie waarbij y = 0, kunnen deze functies berekend worden.

$$(65 * x) + 2 * (90 * 0) + 90x \leq 1750\ \rightarrow 135x\ \leq 1750 \rightarrow x \leq \frac{1750}{155} \rightarrow\ x \leq 11{,}29$$

$$2 * (65 * x) + (90 * 0) + 90x - 65x \leq 1750 \rightarrow 130x + 25x \leq 1750\ \ \rightarrow x \leq \frac{1750}{155} \rightarrow\ x \leq 11{,}29$$

Deze resultaten duiden ten eerste aan dat een upgrade mogelijk is omdat x een positieve waarde is en ten tweede hier verder uit afgeleid worden dat er 11 tenants geplaatst kunnen worden in de eerste applicatieversie op een node met 2 cores om een zero downtime upgrade naar applicatieversie 2 uit te kunnen voeren.



Figuur 10: Oplossingsgebied simulatie

Uit de bovenstaande resultaten wordt er een omgeving opgezet bestaande uit 11 tenants in applicatieversie *p* en 0 tenants in versie *p+i.* Omdat deze omgeving bepaald is via de formules voor een optimale omgeving wordt er verwacht dat de volgende formules aangeven dat er een groep van 11 tenants geüpgraded kan worden.

$$z \leq \frac{1750 - (65 * 11)}{90}; z \leq \frac{1750 - 2(65 * 11)}{(90 - 65)}$$

$$z \leq 11{,}5; z \leq 12{,}8$$

Als conclusie kan men stellen dat in deze omgeving 11 tenants in één keer geüpgraded kunnen worden. Dit voldoet aan de verwachtingen gesteld voor de berekening en ondersteunt de formules voor het opbouwen van een optimale omgeving. Tijdens een sequentiële upgrade van deze omgeving kan het mogelijk zijn dat niet dezelfde hoeveelheid tenants geüpgraded gaan kunnen worden. Dit wordt berekend op basis van de onderstaande formules:

$$RC_{Vp+i}(1) = (65 * 11) - (90 * 0) + 2 * (90 * 0) + (90 * 0) + 90) = 715 + 90 = 805$$

$$RC_{Vp}(1) = 2 * (65 * 11) - (90 * 0) + (90 * 0) + (90) - (65)) = 1430 + 25 = 1455$$

Deze formules moeten verder geïtereerd worden tot de surge over de beschikbare hoeveelheid CPU-resources uitgaat. De verdere berekening van deze formules is zichtbaar in de

onderstaande tabel en duidt aan dat maar 9 tenants volgens de sequentiële methode geüpgraded kunnen worden dit in tegenstelling tot 11 tenants bij de groepsmethode.

Tabel 4: Hoeveelheid requested CPU voor de sequentiële methode

Request CPU			
Tenants	**UpdateV2**	**UpdateV1**	**Totaal**
1	805	1455	2260
2	920	1415	2335
3	1035	1375	2410
4	1150	1335	2485
5	1265	1295	2560
6	1380	1255	2635
7	1495	1215	2710
8	1610	1175	2785
9	1725	1135	2860
10	1840	1095	2935
11	1955	1055	3010

5 Middleware voor Cost-efficient Continuous Deployment Van Multi-tenant SaaS-applicaties

5.1 Introductie

Door het gebruik van containers en multi-tenancy op applicatieniveau zullen de kosten voor een SaaS-beheerder dalen. Containers zijn van nature kleine images en kunnen hun shared libraries delen met elkaar om geheugenefficiënt te werk te gaan zoals eerder aangetoond in sectie 1.4. Multi-tenancy drukt de kosten door de applicatie-instantie te delen onder verschillende klanten, dit zorgt ervoor dat een applicatie maximaal gebruikt kan worden en geen onbenutte resources inneemt.

In dit hoofdstuk wordt de algemene architectuur van de middlewarelaag voorgesteld. Deze laat het plannen van upgrades voor multi-tenant SaaS-applicaties toe en schaalt de applicatie verticaal rekening houdend met de geproduceerde surge. De opbouw van deze middleware is gebaseerd op eerder werk omtrent *continuous evolution* van multi-tenant SaaS-applicaties en biedt ondersteuning voor tenant-specifieke configuratie van upgrades. [1]

Eerst wordt er een vereistenanalyse opgesteld omtrent de middleware die het plannen van upgrades voor multi-tenant applicaties faciliteert. Dit gebeurt op basis van de literatuurstudie en hier worden verder ook de belangrijkste vereisten voor de architectuur bepaald. Hierna wordt de algemene architectuur voorgesteld gevolgd door uitleg over de werking van de middleware voor ontwikkelaars en beheerders. Als laatste wordt er dieper ingegaan op de belangrijke componenten.

5.2 Vereistenanalyse door literatuurstudie

5.2.1 Achtergrond

In dit onderdeel wordt er eerst gefocust op de doelen van dit type middleware, hierna worden deze omgezet naar vereisten. Deze voorwaarden werden verzameld uit verschillende papers over upgrade planning van multi-tenant SaaS-applicaties en verder aangepast door persoonlijke toevoegingen. In deze opsomming wordt ook aangeduid welke onderdelen uitvoerig besproken zullen worden in deze thesis. [1] [47]

Allereerst worden de doelstellingen van deze middlewarelaag opgesteld:

- **D1:** Hoge beschikbaarheid en versie-consistent gedrag zijn van belang.
- **D2:** De planning van de upgrades moet verlopen in tijdsloten die rekening houden met de preferenties van de tenant.
- **D3:** SLA-degradatie moet toelaatbaar zijn
- **D4:** Het upgradeproces moet kostenefficiënt verlopen
- **D5:** De middlewarelaag moet herbruikbaar zijn voor verschillende soorten applicaties.

Deze vijf doelstellingen worden vervolgens uitgebreider geformuleerd om een concreter beeld te geven over hoe ze bereikt en geïmplementeerd kunnen worden.

- **V1:** Hoge beschikbaarheid moet gerealiseerd worden door de middlewarelaag om het vertrouwen van tenants te behouden.
- **V2:** Versie-consistent gedrag zorgt ervoor dat het voor een tenant lijkt alsof er maar één versie van het systeem operationeel is ook al zijn er meerdere tegelijk operationeel. [48]
- **V3:** Planning van upgrades moet rekening houden met preferenties van de tenant. Deze vereiste impliceert dat tenants de mogelijkheid moeten hebben om hun voorkeur uit te drukken wanneer een upgrade uitgevoerd moet worden.
- **V4:** Tenant upgrade isolatie zorgt ervoor dat een upgrade enkel impact heeft op de tenant of de groep van tenants die een upgrade uitvoert
- **V5:** Bereiken van kostenefficiëntie door op basis van de groepsmethode upgrades efficiënt te plannen.
- **V6:** Herbruikbaarheid van de middlewarelaag voor verschillende soorten multi-tenant SaaS-applicaties.

Aangezien de scope van deze thesis beperkt is, zullen niet alle opgesomde vereisten gerealiseerd worden in dit proefschrift. Zo zal er enkel dieper ingegaan worden op hoge beschikbaarheid, planning op basis van preferenties en kostenefficiëntie tijdens upgrades.

5.2.2 Onderzoeksuitdagingen

Onderstaande vereisten vormen de focus van dit onderzoek en zullen de primaire kwaliteiten van de middlewarelaag definiëren.

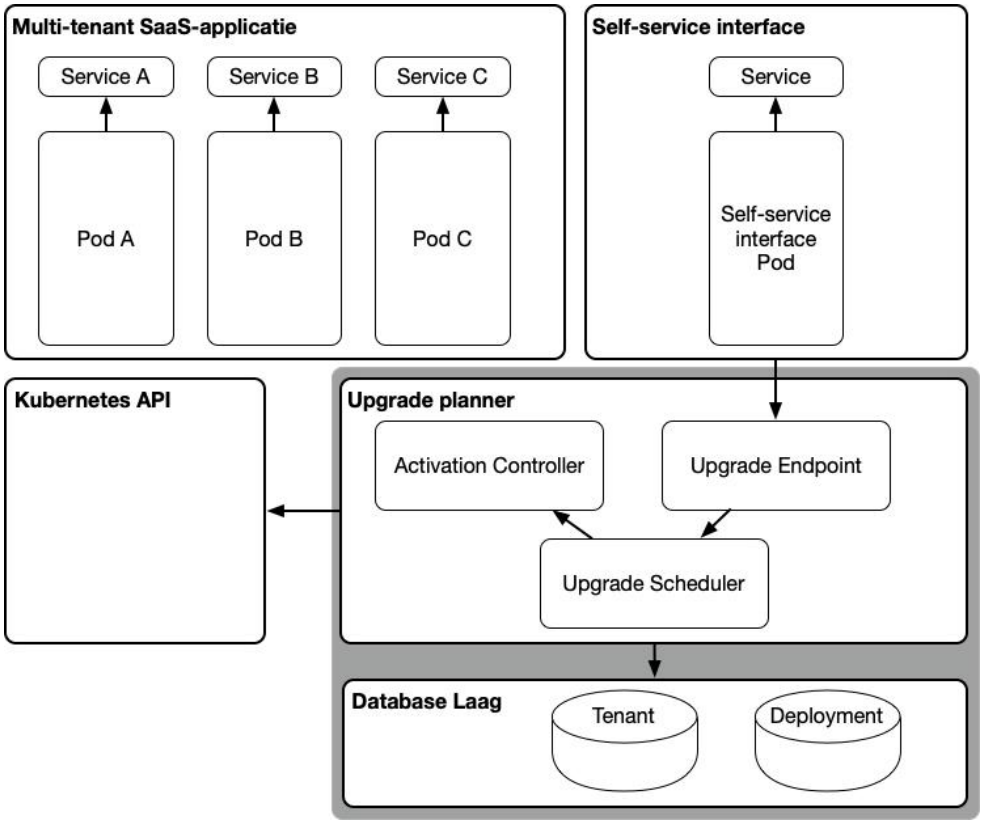
- **O1:** Hoge beschikbaarheid wordt voorzien door migratie van resources te baseren op rolling upgrades zoals beschreven in sectie 4.1.
- **O2:** Planning van tenant upgrades rekening houdend met preferenties van de tenant kan door een self-service interface aangeboden worden aan tenants.
- **O3:** Kostenefficiëntie op basis van de groepsmethode zal ervoor zorgen dat een upgrade altijd zo optimaal mogelijk uitgevoerd wordt door zoveel mogelijk tenants op hetzelfde moment te upgraden.

5.3 Overzicht

De architectuur van onze middleware ondersteunt hoge beschikbaarheid, tenant-specifieke configuratie en kostenefficiëntie van de multi-tenant SaaS-applicaties tijdens het upgradeproces. In Figuur 11 wordt een overzicht van de omgeving weergegeven, bestaand uit verschillende componenten.

Kubernetes. De basis van de architectuur is een Kubernetes cluster. Dit is het platform waarop alle onderdelen zullen draaien en het oefent verder een grote invloed uit op de ontwerpkeuzes die gemaakt zullen worden. Kubernetes is een uitgebreid orchestratie platform met veel functionaliteiten die continuous deployment van de applicatie vergemakkelijken zoals scheduling en de communicatie met de applicatie. De container orchestratie laag biedt de mogelijkheid om functionaliteit van de voorheen complexere middlewarelaag over te nemen. [29]

Kubernetes API. De middlewarelaag kan gebruik maken van deze complexe functionaliteiten dankzij de Kubernetes API. Deze component zorgt voor de communicatie in de cluster en biedt een *interface* aan om de cluster aan te spreken. De middleware maakt gebruik van de API om optimale groepen van tenants te bepalen door de benodigde informatie uit de Deployments te halen. Verder zal dit aanspreekpunt gebruikt worden om de Deployments aan te passen naar de nieuwe resourceconfiguraties en een upgrade uit te voeren volgens het rolling upgrade model. Verdere configuratie zorgt ervoor dat Pods pas aangeven dat ze klaar zijn wanneer de server verkeer kan ontvangen en dat ze samen geplaatst worden om het delen van geheugen tussen containers uit te buiten.



Figuur 11: Overzicht middleware-architectuur

Multi-tenant SaaS-applicatie. Dit onderdeel wordt door de ontwikkelaar behandeld en heeft geen rechtstreekse impact op de middleware. Het ontwerp van de applicatie is irrelevant zolang deze op een Kubernetes cluster uitgerold kan worden, dit is een voordeel omdat de middleware universeel geplaatst kan worden voor verscheidene multi-tenant applicaties. Verder geeft dit de ontwikkelaar vrijheid tijdens de ontwikkeling omdat er geen rechtstreekse verbinding nodig is tussen de middleware en de applicatie. De uitrol van de multi-tenant applicatie hoort over N+1 nodes te gebeuren om te voldoen aan het redundantie model en hoge beschikbaarheid te garanderen. [4]

Self-service interface. De self-service interface is een webinterface die door de ontwikkelaar voorzien wordt om tenant-specifieke configuraties door de tenants zelf te laten uitvoeren. Zo wordt het traditioneel gebruikt voor de configuratie van gebruikers en hun rollen [1]. In deze architectuur zal het ook als communicatiepunt met de Upgrade Planner fungeren, dit bouwt verder op het idee dat de self-service interface tenant-specifieke configuraties aanbiedt en

hierdoor krijgen tenants de vrijheid om zelf een invloed uit te oefenen op het upgradeversie en het upgrademoment.

Upgrade planner. De Upgrade Planner bestaat uit drie hoofdonderdelen: de Upgrade Endpoint, de Upgrade Scheduler en de Activation Controller. De Upgrade Endpoint verzorgt de communicatie met de tenants en ontwikkelaars door verschillende API aanspreekpunten te presenteren waaronder het aanbieden van nieuwe applicatieversies en het plannen van upgrades. Dit onderdeel wordt in detail besproken in sectie 5.4. De Upgrade Scheduler, behandeld in sectie 5.5, is de belangrijkste toevoeging aan de middleware want deze bepaalt de maximale grootte van de tenant groepen door gebruik te maken van de surge algoritmes. Als laatste zorgt de Activation Controller ervoor dat de uitvoering van upgrades verloopt volgens het surge model, dit onderdeel wordt verder uitgelegd in sectie 5.6.

De multi-tenant SaaS-applicatie, de self-service interface, de upgrade planner en de database laag worden allemaal in Pods uitgerold. De multi-tenant applicatie wordt voorzien op nodes waar geen andere Pods draaien, deze keuze zorgt ervoor dat het algoritme op N+1 nodes een uniform verloop en hoge beschikbaarheid kan garanderen. Multi-tenancy in deze architectuur komt overeen met de verdeling van resources over Pods en, zoals in sectie 4 aangehaald, heeft elke applicatieversie een constante waarde per tenant. Zo draagt elke tenant die aanwezig is in de versie bij aan de uiteindelijke resourceconfiguratie van de applicatie.

5.3.1 Multi-tenancy in Deployments

Om multi-tenancy in Kubernetes te voorzien zijn er meerdere mogelijkheden zoals tenant namespaces, gedeelde container en namespace per SLA. [29] Zoals eerder aangehaald wordt er in deze thesis gebruik gemaakt van een gedeelde container strategie. Een Deployment is het Kubernetes object dat gebruikt wordt om tenants te groeperen. Multi-tenant SaaS-applicaties bevatten echter vaak tenants van verschillende niveaus op basis van de afgesloten SLA, deze niveaus worden vertaald naar resources waar een betere SLA-overeenkomt met meer resources per tenant. Om deze mogelijkheid te faciliteren is een Deployment gebonden aan een specifieke applicatieversie en SLA-klasse. Zo is het mogelijk om per Deployment een constante waarde te definiëren die de per tenant verandering in requests en limits weergeeft. Deze waarde wordt op voorhand bepaald door het opstellen van een resourcematrix van verschillende resourceconfiguraties en een constante waarde te berekenen die voldoet aan de SLA

In de Deployment worden echter nog andere specifieke waardes voorzien die gebruikt worden door de Upgrade Planner middleware om upgrades te faciliteren. Figuur 12 toont een voorbeeldconfiguratie van een Deployment. Zoals hierboven vermeld wordt een Deployment verbonden met een applicatie, versie en SLA-klasse, dit gebeurt respectievelijk door de app, version en sla labels.

```yaml
kind: Deployment
metadata:
  name: mt-api-v1
  labels:
    app: mt-api
    version: "1"
    sla: bronze
    requests: "67"
    limits: "100"

strategy:
    type: "RollingUpdate"
    rollingUpdate:
      maxUnavailable: 0%
      maxSurge: 100%


readinessProbe:
          httpGet:
            path: /api/info/
            port: 8080
```

```yaml
affinity:
        podAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 50
            podAffinityTerm:
              labelSelector:
                matchLabels:
                  app: mt-api
              topologyKey: "kubernetes.io/hostname"
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
            - weight: 80
              podAffinityTerm:
                labelSelector:
                  matchLabels:
                    version: "1"
                    sla: bronze
                topologyKey: "kubernetes.io/hostname"
```

Figuur 12: Deployment configuratie

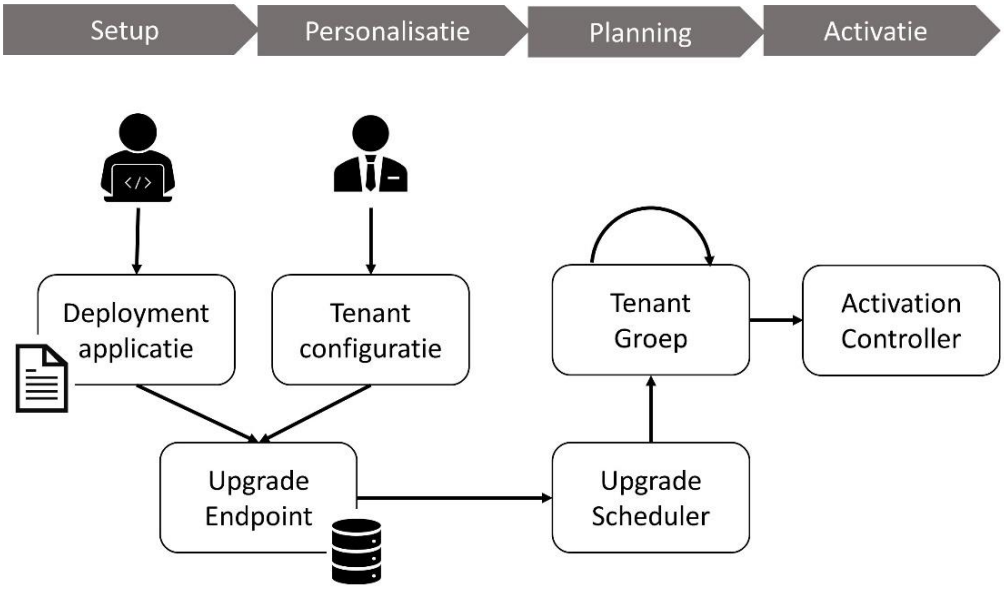
De upgrade strategie wordt zodanig gedefinieerd dat Kubernetes het updaten van de Deployment behandelt volgens de “RollingUpdate” optie. Deze strategie, samen met de ingestelde parameters, zorgt ervoor dat in een omgeving waar er een applicatie Pod per node is, dat eerst de Pod heropgestart wordt alvorens de oude versie wordt afgesloten.

Om zero downtime te garanderen wordt er gebruik gemaakt van een ReadinessProbe. Tijdens het opstarten van een Pod zal deze controleren of de Pod klaar is om gebruikersverkeer te ontvangen, pas wanneer dit het geval is zal de Pod verkeer toelaten.

Om verder te voldoen aan het N+1 foutenmodel en het delen van geheugen tussen de applicatieversies uit te buiten wordt er gebruik gemaakt van het affinity en anti-affinity concept tussen Pods. Zo zullen Pods van dezelfde versie op verschillende nodes verspreid worden om N fouten op te vangen en kunnen Pods van dezelfde applicatie maar verschillende versies op dezelfde node geplaatst worden om de overeenkomstige shared libraries tussen de versies te delen in geheugen.

5.3.2 Verloop van een upgrade

Een upgrade bestaat uit verscheidene elementen die door verschillende stakeholders bepaald worden. In dit onderdeel zal er een overzicht gegeven worden over de verschillende stappen die nodig zijn om een upgrade tot van het begin tot het einde te voorzien. In Figuur 13 wordt het overzicht geïllustreerd.



Figuur 13: Overzicht van het verloop van een upgrade

De eerste stap van het upgrade verloop is de *Setup* fase waarin de nieuwe applicatieversie uitgerold wordt op de cluster door de ontwikkelaar of beheerder. Deze applicatie wordt via een Deployment uitgerold volgens de template voorgesteld in de vorige sectie. De uitrol van de Deployment op de cluster wordt in non-actieve staat gedaan door geen replica's te voorzien. Nadat deze is uitgerold kan de ontwikkelaar of beheerder de Deployment toevoegen aan de planner en kan de upgrade aan de tenants voorgesteld worden.

De *Personalisatie* fase is een stap die voor elke tenant uitgevoerd zal worden en deze omvat de configuratie naar gelang de voorkeuren van de tenant. Een self-service interface biedt de mogelijkheid om deze informatie door te geven aan de middleware. De tenants krijgen hier de keuze over de versie waarnaar ze willen upgraden en het tijdstip van uitvoer.

Hierna wordt de *Planning* fase gestart waarbij tenants in een groep worden geplaatst. Dit proces omvat het koppelen van gelijkaardige tenants in een groep, tenants worden samen in een groep geplaatst als ze dezelfde SLA, huidige versie, upgrade versie en applicatie hebben. Wanneer de deadline verloopt in een groep wordt deze doorgestuurd naar de Activatie fase waar het upgradeproces in gang gezet wordt, dit verloopt zoals beschreven in sectie 4.1.

5.4 Upgrade Endpoint

De Upgrade Endpoint dient als communicatiepunt tussen de stakeholders en de middleware. Het biedt een REST API interface aan die door de SaaS-ontwikkelaar geïntegreerd kan worden in de self-service interface. Dit zorgt voor uniforme configuratie voor meerdere soorten applicaties en garandeert de correcte werking van de planning- en activatiecomponenten. De interface biedt de mogelijkheid tot het configureren van Deployments, tenants en hun upgrade configuratie en het verkrijgen van informatie omtrent upgrades.

Deployments. In sectie 5.3.1 werd al besproken hoe Deployments opgesteld moeten worden om alle parameters te omvatten die nodig zijn voor een correcte werking van het upgradeproces. Deployments kunnen non-actief in de cluster opgestart worden en de ontwikkelaar of beheerder van de applicatie kan dan de applicatieversie beschikbaar maken door de naam van de Deployment aan de Endpoint door te geven. Deze kan dan door gebruik te maken van de Kubernetes API al de benodigde informatie ophalen uit de Deployment configuratie.

Tenants. Omdat de middleware onafhankelijk van de applicatie draait, zal de planner zelf tenant data bijhouden. Tenants onderscheiden zich onderling door middel van een tenant *identifier* (tenant-ID). Deze hoort door de ontwikkelaar van de applicatie toegekend te zijn aan elke unieke tenant en wordt verwacht als identificatie methode tijdens elk verzoek van de tenant. Er wordt aangenomen dat van elke tenant de huidige versie, SLA-klasse en geassocieerde applicatienaam gekend zijn. Tijdens de configuratie van een upgrade zal de tenant of tenant administrator extra parameters moeten meegeven om aan te geven wanneer en naar welke versie de upgrade moet gebeuren. De Endpoint zal deze configuratie doorgeven aan de Upgrade Scheduler.

Upgrades. Omdat de planner en de multi-tenant applicatie los van elkaar opereren is er het nadeel van inconsistente tenant data tussen de middleware en de applicatie. Daarom biedt de Endpoint een interface aan die ervoor zorgt dat, wanneer een upgrade uitgevoerd is, de multi-tenant SaaS-applicatie hier ook van op de hoogte is. De Endpoint zal dan de lijst van tenant *identifiers* en hun upgrade versie doorgeven via deze interface. Door dit aanspreekpunt aan de ontwikkelaar aan te bieden wordt de multi-tenancy aard van de applicatie ondersteund.

5.5 Upgrade Scheduler

De taak van de Upgrade Scheduler is het bepalen van optimale groepsgroottes en vervolgens het in groepen plaatsen van tenants. De bepaling van deze groottes gebeurt door het algoritme gedefinieerd in sectie 4.3. De parameters benodigd voor deze algoritmes worden geleverd door de Deployments, deze werden eerder door de Kubernetes API opgeroepen en al de benodigde data werd opgeslagen in de database. Het aantal tenants per versie wordt berekend op basis van de tenant data waarover de planner beschikt. De beschikbare hoeveelheid CPU om een upgrade uit te voeren is verder een belangrijke factor voor het bepalen van optimale groottes. De Kubernetes API heeft data over de CPU-capaciteit per node en de gereserveerde ruimte kan berekend worden door de som van de Pods per node uit te voeren. Deze laatste is echter wel zonder de ruimte gereserveerd door beide Deployments meegerekend omdat deze door het algoritme zelf in rekening wordt gebracht.

Een andere taak van de Upgrade Scheduler is het plaatsen van tenants in groepen, deze worden gevormd op basis van enkele tenant eigenschappen. Zo geldt er voor alle tenants in dezelfde tenant groep dat ze tot hetzelfde SLA-klasse, huidige versie, upgrade versie en applicatie behoren. Concreet komt dit overeen met de tenants die gebruik maken van dezelfde Deployment en migreren naar dezelfde Deployment. Nadat een upgrade voltooid wordt is het essentieel dat de grootte berekening opnieuw uitgevoerd wordt. Dit kan ervoor zorgen dat tenants uit de groep verwijderd worden op last-in first-out (LIFO) basis. Deze tenants worden door de Upgrade Scheduler uit de groep genomen en deze zullen het groep matching proces opnieuw doorlopen. Deze tenant groepen bepalen verder een deadline voor activatie op basis van de tenant met de meest dringende deadline. De applicatie controleert dit op regelmatige

basis en zal de tenant groep doorgeven aan de Activation Controller wanneer deze deadline bereikt wordt.

5.6 Activation Controller

De Activation Controller bestaat uit een bestaat uit een queue en activator gedeelte en zorgt voor de naleving van het upgradeproces, eerder al beschreven in sectie 4.1 De queue zal de groepen met tenants opvangen die klaar zijn voor de upgrade te starten. Het component laat maar één upgrade op hetzelfde moment toe. De berekening van het surge algoritme houden enkel rekening met de Deployments in de upgrade en meerdere upgrades op hetzelfde moment zou het model in gedrang kunnen brengen.

De Activator component maakt gebruik van data die bij de tenantgroep horen en verkrijgbaar is via de Deployments. Om ervoor te zorgen dat verschillende fases niet gelijktijdig verlopen wordt de Kubernetes API gebruikt om te controleren wanneer een Pod klaar is. Zodra de eerste rolling update volledig doorlopen is, kan de volgende gestart worden, dit verzekert de hoge beschikbaarheid van de applicatie. De request en limits per tenant zullen gebruikt worden om de nieuwe waardes van de resourceconfiguraties van de Deployments te bepalen. De huidige configuratie wordt als basis gebruikt en, afhankelijk van de hoeveelheid tenants die migreren worden de waardes erbij opgeteld of afgetrokken. Eén van de fases van het upgradeproces is het switchen van tenants naar de nieuwe versie. Tijdens deze stap wordt een update gestuurd via de Upgrade Endpoint, zo kan de applicatie reageren op deze verandering en tenants intern behandelen om een correcte service aan te bieden.

6 Implementatie

Er werd een prototype van de middleware, gepresenteerd in hoofdstuk 5, geïmplementeerd. Deze implementatie omvat de hoofdfunctionaliteiten van de middlewarelaag om de evaluatie van het surge model te voltooien. Het prototype wordt uitgerold bovenop een Kubernetes cluster en in sectie 6.1 wordt er uitleg gegeven over het overzicht van het prototype waaronder de Deployment op de cluster en implementatie keuzes. Vervolgens wordt de implementatie van de verschillende componenten uit de architectuur toegelicht, dit omvat de implementatie van de Upgrade Endpoint, Upgrade Scheduler en de Activation Controller.

6.1 Overzicht

Cluster. Het prototype is ontworpen om in een Docker container te draaien. Hiermee kan het draaien op elke soort Cloud infrastructuur die Kubernetes aanbiedt of op een lokale server infrastructuur met een Kubernetes cluster. De applicatie wordt als een Deployment uitgerold bovenop deze cluster. Hiervoor moet de applicatie in een Docker image gebundeld worden en wordt er een Service voorzien voor communicatie tussen de cluster en de buitenwereld. Om communicatie met de Kubernetes API te voorzien wordt er gebruik gemaakt van een serviceaccount, deze geeft toegang tot het gebruik van de API. [49]

Prototype. De applicatie maakt gebruik van het Java Spring framework [50] om een RESTful web service aan te bieden die communicatie met de cluster voorziet, de verschillende componenten zijn geïmplementeerd in één applicatie-instantie voor het prototype. Het Java Spring Framework is een volwassen en veel gebruikt framework dat aan de noden van het prototype voldoet. De middleware omvat verder ook een database laag en deze wordt vertaald in een PostgreSQL container [51], hier hoort ook een Kubernetes Service en ConfigMap bij. Deze laatste dient voor gevoelige informatie omtrent de connectie met de database zoals username en passwoord op te slaan. Beide worden in een verschillende Deployments op nodes uitgerold die voorbehouden worden voor ondersteunende services.

6.2 Upgrade Endpoint

De Upgrade Endpoint biedt een REST API [52] aan om geïntegreerd te worden in de self-service interface volgens de wensen van de SaaS-ontwikkelaar. Op deze manier worden er voor elk soort applicatie dezelfde parameters verwacht en kan de planner altijd correct zijn taak uitvoeren. Voor al de *endpoints* geldt dat ze zowel een POST- als een GET-methode aanbieden. De volgende aanspreekpunten worden aangeboden:

- **/tenants:** Dit pad zorgt voor het inladen van de initiële tenant data zoals tenant-ID, applicatienaam, SLA-klasse en de huidige applicatieversie. Het endpoint accepteert een lijst van JSON-objecten die verwerkt en opgeslagen zullen worden in de database. Het GET-request op dit aanspreekpunt geeft de volledige lijst van alle tenants terug in de database.

- **/deployment:** De deployment endpoint krijgt als parameter de naam van de Deployment meegegeven. Met deze naam wordt de Kubernetes API gebruikt om verdere informatie over de Deployments op te zoeken en op te slaan in de database. De GET-methode verwacht een applicatie-naam, versie en SLA-klasse als parameters en geeft de Deployment data terug die gevonden werd in de database.
- **/tenantconfig:** Deze wordt gebruikt door de tenants om hun configuratie voor de upgrade door te geven. Dit pad aanvaardt een lijst van JSON-objecten om meerdere upgrades op hetzelfde moment te kunnen plannen. In het prototype wordt er gebruik gemaakt van precieze deadlines, dit vergemakkelijkt de timing van de evaluatie. De GET-methode verwacht een tenant-ID en geeft de specifieke tenant data terug.

Wanneer tenants POST-verzoeken sturen naar de Upgrade Endpoint, wordt hier meteen een antwoord op gegeven in de vorm van hun verzoek om succes aan te duiden. Dit is door de asynchrone aard van de applicatie waarin elke configuratie door een planning proces moet gaan en de deadlines aanzienlijk later kunnen zijn dan het moment van verzoek.

6.3 Upgrade Scheduler

De Upgrade Scheduler wordt aangesproken wanneer een tenant een upgrade configuratie via het aanspreekpunt doorgeeft. Dit onderdeel zorgt niet enkel voor de bepaling van groepsgroottes maar zal verder ook het centraal punt zijn waar de lijst van tenant groepen bijgehouden wordt. De klasse is geïmplementeerd volgens het Singleton patroon om ervoor te zorgen dat er maar één instantie van de component is, hierdoor worden problemen zoals meerdere lijsten van tenants vermeden.

Zoals eerder vermeld zal de Upgrade Scheduler op basis van de SLA-klasse, applicatienaam, huidige versie en upgrade versie, de tenant configuratie in een tenant groep proberen te plaatsen. Indien er geen groep gevonden wordt zal een nieuwe groep aangemaakt worden waarbij de maximale grootte van de groep wordt bepaald op basis van de algoritmes gedefinieerd in sectie 4.3. Om te communiceren met de Kubernetes API wordt er gebruik gemaakt van de Kubernetes Java API-client. [53] Deze library wordt dan gebruikt om nodes en Pods met een specifiek label te selecteren en zo de beschikbare ruimte op de nodes, voor uitrol van de applicatie, te berekenen. De groepen van tenants worden als een TenantGroup Java Object gerepresenteerd, dit object bestaat uit een lijst van tenants, de huidige en de upgrade versie, een applicatienaam, de SLA-klasse, een deadline en de overeenkomstige Deployment Java objecten. Het is dus een aggregatie-klasse van Deployments en tenant configuraties. Het moet echter mogelijk zijn om deze deadlines te controleren op regelmatige basis. Hiervoor wordt gebruik gemaakt van de ScheduledExecutorService van Java, dit object biedt de mogelijkheid om periodiek een functie uit te voeren die controleert of de deadline bereikt is of niet. Als dit het geval is, wordt de groep doorgestuurd naar de Activation Controller a.d.h.v. het Observer patroon waar elke TenantGroup instantie als Publisher fungeert voor de ActivationController. Op dit moment kan de Scheduler stopgezet worden en de groep verwijderd worden uit de Upgrade Scheduler.

6.4 Activation Controller

De Activation Controller is ook volgens het Singleton patroon geïmplementeerd, dit was een noodzaak om ervoor te zorgen dat upgrades niet door verschillende controllers gestart kunnen worden. De controller krijgt updates van een TenantGroup wanneer deze de deadline bereikt, de groep wordt dan toegevoegd aan een queue. Zo kan ervoor gezorgd worden dat upgrades één per één uitgevoerd worden om de correctheid van het algoritme te garanderen.

De Kubernetes Java API-client speelt een grote rol in dit onderdeel want het voorziet de aanpassing van de Deployments in Kubernetes. Dit gebeurt door gebruik te maken van een Patch object [54], het biedt de mogelijkheid om JSON-objecten aan te passen door waardes te vervangen op basis van paden. In Figuur 14 wordt dit gebruikt om de requests en limits voor de CPU aan te passen door de oude waardes te vervangen met de nieuwe.

```
V1Patch patchLimitsRequests =
    new V1Patch("[{\"op\":\"replace\","
                +"\"path\":\"/spec/template/spec/containers/0/resources/requests/cpu\",\"value\":\"" + currRequests + "m\"},"
                + " {\"op\":\"replace\","
                +"\"path\":\"/spec/template/spec/containers/0/resources/limits/cpu\",\"value\":\"" + currLimits + "m\"}]");
```

Figuur 14: JSON Patch requests/limits

Vervolgens worden de fasen van het upgrade model gevolgd om zero downtime te garanderen. De Kubernetes API wordt gebruikt om elke seconde te controleren of de nieuwe Pod klaar is om verkeer te ontvangen en de verouderde Pod verwijderd is. Tijdens de eerste fase van het upgradeproces is er een periode waarin de tenants overschakelen van de oude versie naar de nieuwe, in het prototype wordt dit gedaan door een request naar de applicatie te sturen met hierin een lijst van alle tenants in de upgrade en de upgrade versie. Nadat deze fase is afgelopen kan er overgeschakeld worden naar de volgende fase waar de oude versie geoptimaliseerd wordt. Na het verloop van deze upgrade wordt de Deployment database geüpdatet door het gebruik van de Deployment namen, hier zal de Kubernetes API de overige metadata opvragen. De tenant database zal ook aangepast worden om de wijzigingen in applicatieversie te reflecteren.

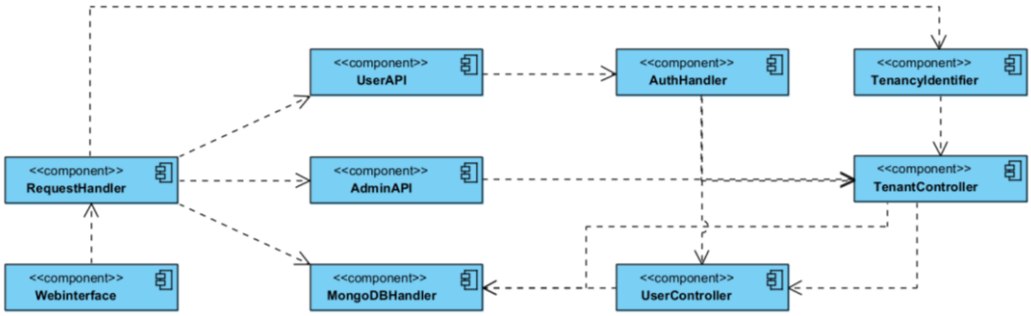
7 Evaluatie

In dit hoofdstuk worden experimenten uitgevoerd die zowel de juistheid van het prototype uit hoofdstuk 6 als het upgradeproces op basis van de groepsmethode uit hoofdstuk 4 evalueren. Het eerste experiment vergelijkt de verschillende upgrademethodes op basis van de CPU-gebruikskosten. Het tweede experiment vergelijkt de verschillende upgrademethodes in responsetijd door latency te meten.

Dit hoofdstuk is als volgt onderverdeeld, in de eerste sectie wordt er een korte uitleg gegeven over de applicatie die gebruikt wordt voor de evaluatie. Hierna volgt een sectie over de evaluatiemethode die in gaat op de omgeving en andere experiment onafhankelijke factoren. Verder worden de verschillende experimenten met elk hun onderzoeksvraag, opzet en resultaten beschreven. Als laatste volgt de discussie waar de resultaten kritisch bekeken zullen worden.

7.1 Multi-tenant applicatie

Voor de evaluatie van de middleware en het upgradeproces voor multi-tenant applicaties werd er een applicatie ontwikkeld die minimale eigenschappen van dit soort applicaties omvat. De applicatie maakt gebruikt van NodeJS [55] als javascript-runtime en het ExpressJS framework [56] om een javascript webserver omgeving aan te bieden. De tenant en gebruikersinformatie wordt opgeslagen in een MongoDB database implementatie. Een overzicht van de applicatie wordt getoond in Figuur 15. De webinterface is de server waardoor de gebruikers met de applicatie kunnen communiceren. Elk inkomend request wordt door de RequestHandler opgevangen, deze maakt gebruik van de TenancyIdentifier om te controleren of dat de tenant wel toegang heeft tot de applicatieversie van deze applicatie-instantie. Hierna worden de requests behandelt door de API-componenten.



Figuur 15: Overzicht architectuur applicatie

Om multi-tenancy te voorzien in de applicatie worden er een tenant en een gebruikers-database voorzien, elke tenant beschikt over een applicatieversie en meerdere gebruikers. Voor de tenants worden enkele basis-functionaliteiten voorzien zoals registratie, login en een homescherm. Verder worden er ook aanspreekpunten aangeboden om tenants van applicatieversie te wijzigen en tenants toe te voegen. De applicatieversie van de tenant wordt

gebruikt om te kijken of ze door de juiste applicatie-instantie bediend worden. Indien een tenant met een verschillende applicatieversie probeert te verbinden naar de applicatie-instantie, zal deze een 301 *Moved Permanently* antwoord krijgen.

In de evaluatie wordt er gebruik gemaakt van twee applicatieversies, versie 1 en versie 2. Deze applicatieversies zijn ongeveer kopieën van elkaar en verschillen niet in functionaliteit, de enigste verandering die zich voordoet is de wijziging van versie voor de instantie, deze kan opgevraagd worden via een API endpoint.

7.2 Evaluatiemethode

De evaluatie van de experimenten houdt rekening met verschillende aspecten. Beide experimenten bevatten grotendeels gemeenschappelijke onderdelen maar elk experiment heeft ook (zijn) eigen specifieke configuraties. Zo stemmen ze overeen in hetzelfde gebruikersscenario, cluster setup en meettools om werklading te genereren maar maken ze elk gebruik van verschillende upgrademethodes.

Upgrademethodes. In deze evaluatie zullen er twee upgrademethodes gebruikt worden die met elkaar vergeleken worden. De eerste methode is de sequentiële upgrade waarbij tenants één per één een upgrade zullen uitvoeren tot een maximum aantal tenants bereikt is. De tweede methode is de groepsupgrade waarbij het prototype een maximale groep van tenants zal vormen en deze gelijktijdig zal upgraden. De methode maakt gebruik van de surge algoritmes zoals gedefinieerd in de thesis om deze berekeningen te doen. De correcte werking van de upgrade planner werd geverifieerd en wordt getoond in Appendix A.

Gebruiksscenario. Om de applicatie en middleware te evalueren is er een gebruiksscenario opgesteld voor de applicatie. Dit scenario komt overeen met een realistische workflow die gebruikers van de applicatie zouden kunnen uitvoeren. Gebruikers zullen zich éénmaal registeren, hierna loggen ze zich in en worden ze naar de homepagina doorverwezen. Als laatste wordt de applicatieversie info opgevraagd. Deze stappen vormen telkens een uitvoering van requests voor één gebruiker. Voor elk experiment zal er op voorhand een aantal gebruikers per seconde gedefinieerd worden die de laatste drie acties: login, home en info, elke seconde uitvoeren.

Cluster opzet. De middleware en applicatie draaien in een Kubernetes cluster omgeving met één master en twee worker nodes. De Kubernetes cluster heeft als client versie 1.14.1 en server versie 1.14.10. De master node beschikt over 4 vCPUs en 4GB RAM, deze configuratie geldt ook voor één van de twee worker nodes waarop de upgrade planner en andere hulp- en meetapplicaties zullen draaien. De andere worker node wordt gebruikt om de applicatie op uit te rollen, deze node beschikt over 2 vCPUs en 2GB RAM. Al deze nodes worden in een private Openstack Cloud-omgeving van het departement Computerwetenschappen opgezet en maken gebruik van Ubuntu 16.04.6 LTS als besturingssysteem. De server bestaat 56 cores (Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz), twee 10Gbit netwerk interfaces en 270GB RAM. De cluster-installatie is verlopen volgens de Kubernetes documentatie [57] en maakt gebruik van de Calico Pod-netwerk uitbreiding.

Workload. Rolling upgrades worden uitgevoerd in omgevingen met actieve gebruikers. Om dit te simuleren wordt er gebruik gemaakt van de k8-scalar *workbench*. [58] Deze meetapplicatie zal een werklading genereren op basis van een configuratie. De workbench is uitgebreid om

voor elke gebruiker het bovenstaande gebruiksscenario uit te voeren. Tijdens de metingen zullen upgrades uitgevoerd worden op de applicatie en horen gebruikers van versie te wisselen. De k8-scalar zal gebruikers van applicatieversie wisselen wanneer deze een *301 Permanently Moved* status terugkrijgt, dit geeft aan dat de gebruiker zijn versie niet overeenkomt met de applicatie-instantie versie. Op deze manier kan via deze kunstmatige werklading een upgrade met veranderend verkeer gesimuleerd worden.

Experiment opzet. Elk experiment bestaat uit enkele gemeenschappelijke onderdelen. De requests en limits per tenant zijn bepaald voor een SLA waarvoor de response latency lager dan 200ms moet zijn, op basis hiervan is er een response matrix opgesteld volgens de methode beschreven in sectie 4.2. De resourcematrix en berekening van de optimale omgeving wordt in sectie 7.3 uitgevoerd en produceert voor requests en limits de waardes 67millicores (m) en 100millicores (m) per tenant. Door deze waardes kan een optimale omgeving opgezet worden waarin 13 gebruikers aanwezig zijn, iedere gebruiker representeert één tenant. Voor applicatieversie 1 betekent dit een start waarde van 871m en 1300m. Voor de uitvoer van de experimenten, werden ook enkele tests opgezet om de mediaan duur van het upgradeproces te bepalen. Dit kwam uit op 110s en deze waarde komt overeen met één iteratie of de upgrade van één tenant/tenantgroep. Elk experiment wordt 10 keer voor elke methode uitgevoerd om een uitgebreide data set te genereren en trends te identificeren. Dit elimineert ook mogelijke uitschieters in data.

Validiteit experiment. De validiteit van de experimenten hangt af van verschillende factoren. De experimenten werden in een private Cloud omgeving uitgevoerd, elk experiment werd op verschillende tijdstippen over een periode van een week uitgevoerd. Dit betekent dat elk experiment in een andere conditie is uitgevoerd en dit kan invloed hebben op de resultaten. Verder is het niet mogelijk om een perfecte timing tussen de workbench en de upgrade in te stellen. De workbench vraagt op voorhand een vaste tijdsperiode om te meten terwijl een upgrade van variabele duur is. Voor het experiment waar latency tijdens de upgrade gemeten moet worden kan dit een impact hebben op de resultaten. Te veel neutrale ruimte in de meting kan ervoor zorgen dat de data niet representatief is. De applicatie die gebruikt wordt om gebruiksdata van elke Pod te meten (Grafana) krijgt deze informatie om de minuut. Omdat upgrades ongeveer 110s duren zijn wijzigingen in CPU-gebruik moeilijk te meten.

7.3 Pre-experiment: Resourcematrix

Het pre-experiment toont de meting van de data en berekening van de constante request en limiet waarden die verder gebruikt worden om upgrades te plannen. Deze waardes worden zo bepaald dat er geen SLA-breuk voorkomt en op basis hiervan wordt er een optimale omgeving bepaald voor de volgende experimenten.

Opzet. Het experiment maakt gebruik van de k8-scalar *workbench* om een resourcematrix te genereren. Dit gebeurt door verschillende werkladingen tegen verscheidene resourceconfiguraties uit te voeren en de latency te meten. Het uitvoeren van deze metingen gebeurt op een geautomatiseerde manier door de k8-scalar een bereik van gebruikers/s voor specifieke limiet configuraties te laten uitvoeren. Wanneer de metingen voor één configuratie uitgevoerd zijn, wordt de configuratie aangepast tot een lagere limiet en wordt het proces herhaald. Elke meting produceert de response latency voor 50, 90, 95, 99, 99.90, 99.99% van de aanvragen. Voor de resourcematrix wordt er enkel rekening gehouden met de 95%

metingen. Op basis van deze matrix wordt er dan verder een constante waarde bepaald die overeenkomt met de limiet waarde in de Kubernetes configuratie.

Resultaten. De resultaten van de verschillende metingen worden verzameld in een resourcematrix zoals zichtbaar in Figuur 16. De latency waardes in de groene vakken voldoen aan de SLA voor de resourceconfiguratie. Voor grote limietwaardes (>1000m) voldoet de SLA voor meer dan 20 gebruikers/s, bij de lagere resourceconfiguraties is dit echter moeilijker te garanderen. In Tabel 5 wordt de berekening van een constante waarde voor de limits getoond die ervoor zorgt dat de SLA altijd voldoet voor alle gebruikers. Dit gebeurt door de maximale limiet in te vullen voor de verschillende aantallen gebruikers/s en zo vervolgens een limiet/user configuratie te bekomen. Deze limiet is 100m en de request waarde wordt berekend op een constante waarde (2/3) van de limiet waarde wat als uitkomst 67m geeft.

	1 user/s	2 user/s	3 user/s	4 user/s	5 user/s	6 user/s	7 user/s	8 user/s	9 user/s	10 user/s	11 user/s	12 user/s	13 user/s	14 user/s	15 user/s
100m	196.094ms.	310.512ms.	403.211ms.	495.973ms.	1031.393ms.										
200m	92.19ms.	100.79ms.	188.04ms.	200.893ms.	298.252ms.										
300m	78.207ms.	93.789ms.	95.382ms.	104.319ms.	197.62ms.										
400m	69.078ms.	86.598ms.	87.837ms.	95.382ms.	106.574ms.	197.446ms.	302.043ms.	294.757ms.	312.042ms.	384.762ms.					
500m	54.773ms.	80.035ms.	92.466ms.	162.917ms.	186.056ms.	198.143ms.	215.584ms.	281.07ms.	277.631ms.	296.941ms.	309.823ms.	401.928ms.	582.16ms.	483.354ms.	580.688ms.
600m	45.631ms.	75.376ms.	74.485ms.	73.162ms.	78.98ms.	100.751ms.	174.596ms.	188.535ms.	201.116ms.	191.808ms.	270.538ms.	198.578ms.	275.8ms.	291.415ms.	307.116ms.
700m	28.407ms.	67.91ms.	60.347ms.	68.175ms.	70.431ms.	82.622ms.	106.722ms.	160.762ms.	183.293ms.	171.623ms.	200.97ms.	187.103ms.	274.487ms	210.075ms.	286.386ms.
800m	20.422ms.	49.834ms.	45.597ms.	53.955ms.	60.377ms.	72.387ms.	96.605ms.	108.395ms.	125.855ms.	112.993ms.	200.646ms.	170.124ms.	208.305ms	207.005ms.	208.968ms.
900m	15.822ms.	48.385ms.	34.024ms.	56.275ms.	59.204ms.	69.077ms.	90.241ms.	113.811ms.	129.863ms.	116.942ms.	185.933ms.	181.351ms.	197.524ms	188.509ms.	218.615ms.
1000m	9.907ms.	9.211ms.	12.339ms.	13.102ms.	13.032ms.	62.508ms.	44.926ms.	86.624ms.	91.492ms.	90.895ms.	101.466ms.	101.226ms.	107.395ms	103.197ms.	159.69ms.
1100m	11.774ms.	13.945ms.	17.927ms.	13.135ms.	35.911ms.	65.798ms.	70.41ms.	75.112ms.	79.393ms.	80.394ms.	91.06ms.	95.363ms.	100.669ms	96.71ms.	114.239ms.
1200m	12.604ms.	26.507ms.	18.523ms.	17.858ms.	41.718ms.	50.254ms.	59.221ms.	83.483ms.	90.41ms.	99.218ms.	100.733ms.	106.529ms.	127.589ms	154.5ms.	152.627ms.
1300m	15.726ms.	16.535ms.	16.291ms.	16.472ms.	22.488ms.	30.896ms.	64.771ms.	71.008ms.	96.512ms.	91.382ms.	101.355ms.	101.054ms.	113.47ms.	107.134ms.	158.656ms.
1400m	12.593ms.	13.948ms.	21.253ms.	12.065ms.	16.728ms.	34.014ms.	52.363ms.	75.987ms.	77.269ms.	76.515ms.	94.066ms.	85.809ms.	99.434ms.	108.761ms.	118.542ms.
1500m	15.033ms.	12.562ms.	16.495ms.	15.351ms.	16.937ms.	33.853ms.	41.146ms.	59.172ms.	70.838ms.	61.866ms.	76.09ms.	75.726ms.	94.151ms.	88.488ms.	97.851ms.

Figuur 16: Resourcematrix

Tabel 5: Berekening constante waarde voor limits op basis van de resourcematrix

CPU-limiet	Users/s	Limit/users
100	1	100.00
200	2	100.00
300	4	75.00
400	5	80.00
500	5	100.00
600	7	85.71
700	9	77.78
800	10	80.00
900	12	75.00
1000	20	50.00

Deze waardes worden verder gebruikt om een optimale omgeving te bepalen volgens de formules uit sectie 4.3.1. Hieruit blijkt dat 13 versie *p* tenants met de bovenstaande resourceconfiguratie per tenant, een optimale omgeving produceren waarin het aantal tenants even groot is als de maximale upgrade groep.

$$RC_{Vp+i}(x) = (67 * x) + 2 * (67 * 0) + (67 * x) \leq 1750$$

$$RC_{Vp}(x) = 2 * (67 * x) + (67 * 0) \leq 1750$$

$$Fase\ ScaleUp = x\ \leq \frac{1750 - 2 * 67 * 0}{134} = 13{,}05$$

$$Fase\ ScaleDown = x\ \leq \frac{1750 - 67 * 0}{2 * 67} = 13{,}05$$

7.4 Experiment 1: Gebruikskost

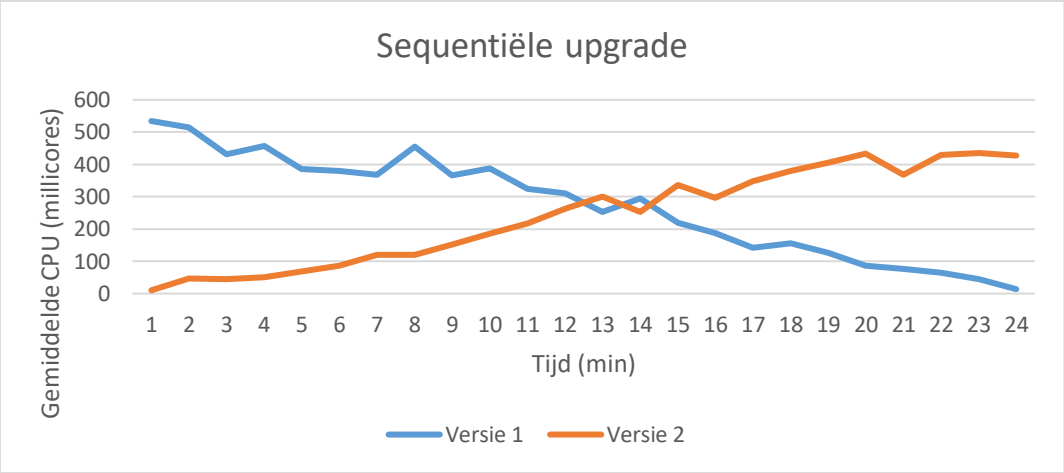
Het eerste experiment wordt uitgevoerd om de totale gebruikskost van beide upgrademethodes te evalueren. Het doel van dit experiment is om te zien of het CPU-gebruik lager ligt bij de sequentiële of groepsmethode.

Onderzoeksvraag. Worden CPU-resources gespaard door gebruik te maken van de groepsmethode?

Hypothese. De verwachting van de resultaten is dat de sequentiële methode meer CPU-resources zal gebruiken dan de groepsmethode. De CPU-resources per upgrade zullen niet verschillen maar omdat de sequentiële methode langer duurt zal deze CPU-resources langer in beslag nemen.

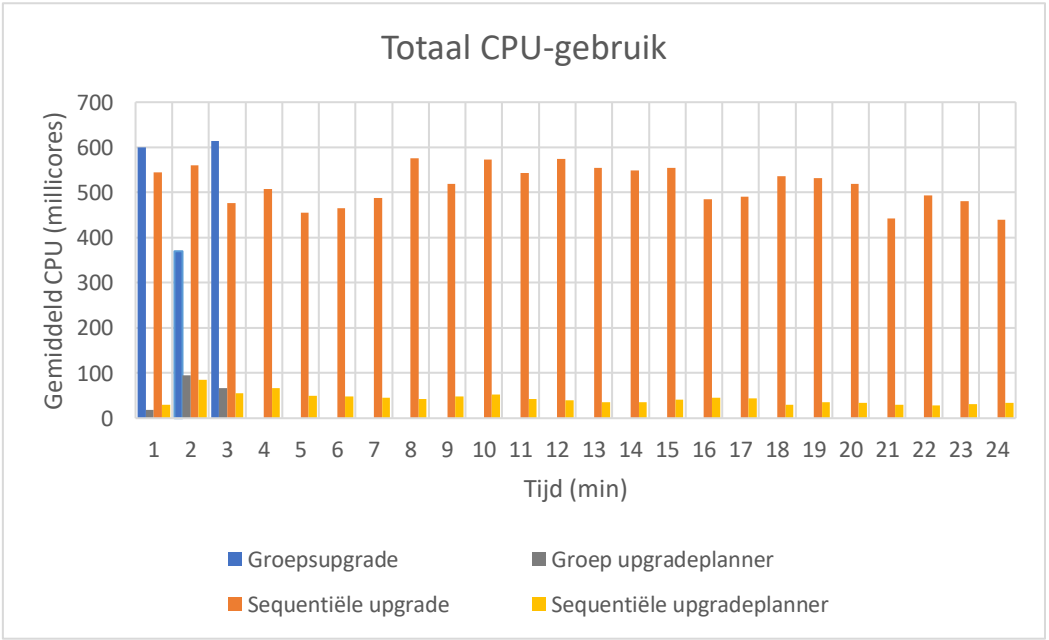
Opzet. Het experiment zal upgrades simuleren volgens de sequentiële en groepsmethode tijdens een gemiddelde werklading. Dit scenario komt overeen met een normaal gebruik van de applicatie door de tenants. De k8-scalar zal 8 user/s aan werklading genereren in een omgeving die begroot is op 13 gebruikers. Het meetproces bestaat uit drie fases: opwarming, upgrades en buffer. De opwarm fase is een periode van 120s voordat de upgrades van start gaan, in deze periode wordt de eerste applicatieversie al onder werklading gezet. De upgrade stap zal de upgrades starten, voor de sequentiële methode betekent dit een periode van 13 keer 110s en voor de groepsmethode 1 keer 110s. Na deze stap volgt er een buffer fase van 15s per uitgevoerde upgrade om upgrades die langer dan de gemiddelde tijd duren op te vangen. De k8-scalar configuratie zal enkel en *peak load* en *peak duration* meten.

Resultaten. De resultaten van dit experiment worden in verschillenden grafieken weergegeven. In Figuur 17 wordt het verloop van de sequentiële runs getoond, deze grafiek is opgebouwd op basis van de gemiddelde data van de sequentiële upgrades. De grafiek toont het verloop van werklading van één versie naar de andere en de tijdsduur van een sequentiële upgrade.

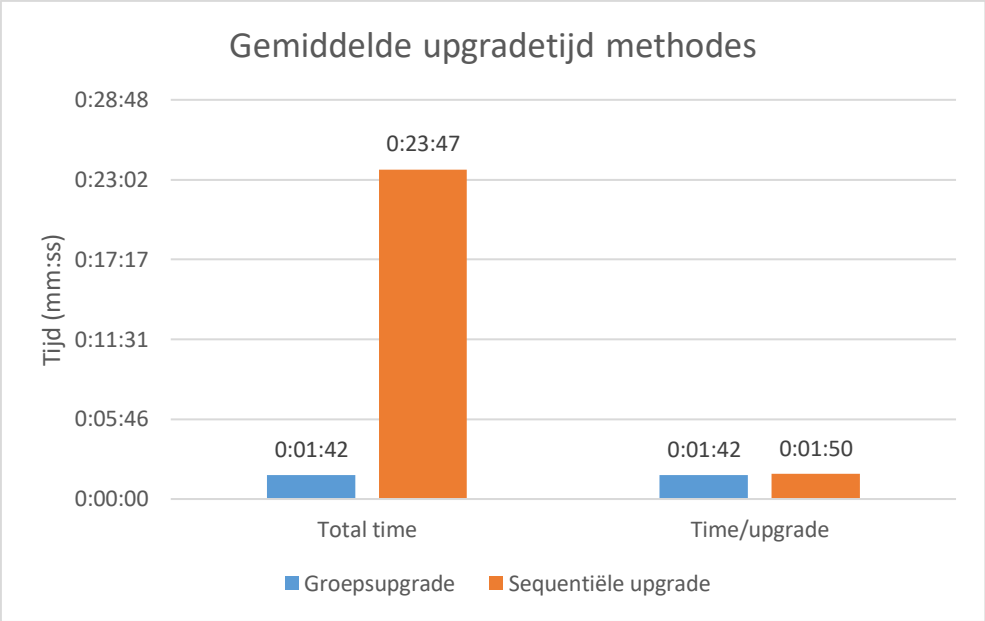


Figuur 17: Grafiek van het upgradeproces via een sequentiële upgrade

Figuur 18 toont het totaal gemiddeld CPU-gebruik per minuut en het CPU-gebruik van de upgradeplanner voor beide upgrademethodes. Deze grafiek geeft het tijdsverschil tussen beide methodes weer en het totale CPU-gebruik voor elk datapunt. Het CPU-gebruik van de sequentiële en groepsupgrade verschilt niet veel per stap zoals verwacht maar de sequentiële methode wordt over een langere periode uitgevoerd. Voor elke methode worden ook de CPU-resources van de upgradeplanner getoond en deze vertonen gelijke waardes. De laatste grafiek, Figuur 19, toont het verschil in gemiddelde upgradetijd tussen beide methodes.



Figuur 18: Totaal gemiddeld CPU-gebruik per datapunt en upgradeplanner CPU-gebruik



Figuur 19: Totale en gemiddelde tijd per upgrademethode

7.5 Experiment 2: High workload upgrades

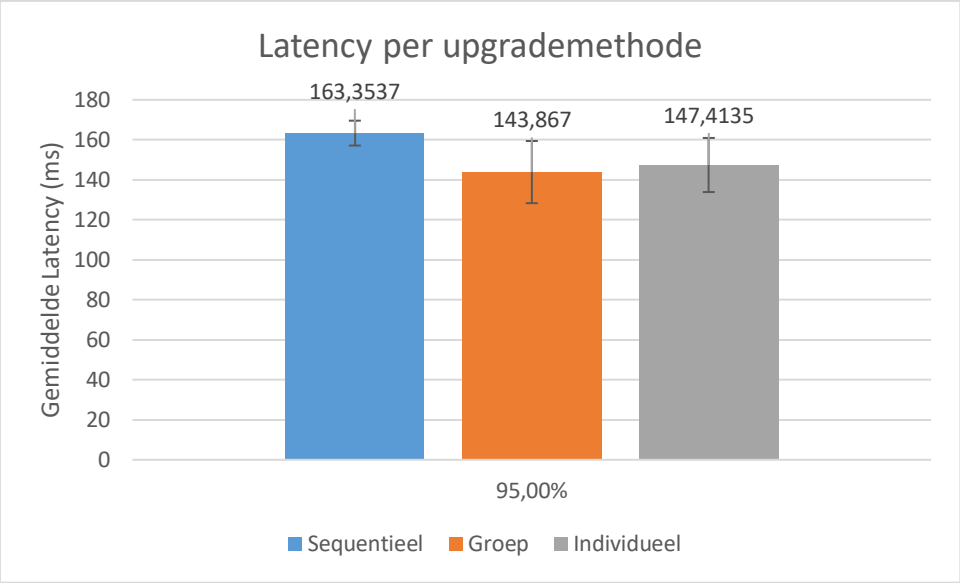
Het tweede experiment wordt uitgevoerd om de impact van beide methodes op vlak van latency te meten tijdens een hoge werklading. Het doel van dit experiment is om te bewijzen of de surge Pod van de groepsmethode een negatieve impact heeft op de CPU Scheduler en zo de latency van de request responses verhoogt.

Onderzoeksvraag. Is er een impact op de latency tussen de twee methodes wanneer upgrades tijdens een hoge werklading uitgevoerd worden?

Hypothese. De groepsmethode heeft een impact op de latency van requests. Deze assumptie wordt gemaakt door de manier waarop CPU verdeeld wordt in timeshares. Door een hoge werklading te genereren zal de applicatie onder stress komen te staan. Een upgrade voegt tijdelijk een extra Pod met een hoge request en limiet waarde toe, hierdoor kan mogelijk *contention* voor CPU-resources ontstaan.

Opzet. Tijdens dit experiment wordt er een hoge werklading gebruikt om een upgrade op een piekmoment te simuleren. De werklading is 13 gebruikers/s en komt overeen met de maximale werklading die een response latency garandeert en voldoet aan de afgesproken SLA. In dit experiment wordt er ook een derde methode toegevoegd, de individuele methode komt overeen met een upgrade van één tenant in de omgeving met maximaal 13 tenants. Deze wordt toegevoegd om een upgrademethode te hebben die een incrementele upgrade nabootst maar met dezelfde duur als de groepsmethode.

Resultaten. In Figuur 20 wordt de gemiddelde response latency voor de drie verschillende methodes getoond. De sequentiële methode lijkt een hogere latency te produceren dan de twee andere methodes, die meer gelijkende waardes lijken te vertonen.



Figuur 20: Vergelijking van de response latency tussen verschillende methodes

7.6 Discussie

In dit onderdeel wordt er kritisch gekeken naar de resultaten en wordt er een mogelijke verklaring gegeven voor de resultaten van elk experiment gegeven.

Experiment 1. Het experiment werd uitgevoerd om de gebruikskost van de verschillende upgrademethodes te meten, voor beide methodes toont de data een upgrade van 13 tenants. De resultaten dienen niet om resources waardes voor een upgrade te bepalen onder een gemiddelde werklading maar tonen eerder een trend van resourcegebruik. Zo zijn er geen grote verschillen tussen het CPU-gebruik van de groepsmethode en dat van de sequentiële methode per datapunt. Dit komt overeen met de verwachtingen van het experiment. De totale gebruikskost schetst echter een ander verhaal, het is duidelijk dat de sequentiële methode veel meer tijd nodig heeft om een upgrade uit te voeren. Zo kan het tot 13 keer langer duren in dit experiment maar voor grotere omgevingen met meer tenants zal dit dus een nog grotere impact hebben. Dit zorgt voor een veranderende omgeving en kan impactvol zijn op resourcegebruik, latency en andere servicekwaliteiten. De tijd zorgt ook voor een langer gebruik van de upgradeplanner en een kleine maar constante overhead die deze introduceert tijdens het upgradeproces.

Experiment 2. Het doel van dit experiment was het observeren van latency verschillen tijdens upgrademethodes. Dit ondersteunt of de upgradeplanner voor een zero downtime upgrade zorgt en of de groepsmethode meer latency introduceert door CPU-resource *contention*. Voor dit experiment is er een derde methode toegevoegd geweest om een upgrade voor te stellen van één tenant maar met dezelfde tijdsduur als een groepsupgrade.

Uit de resultaten blijkt er geen extra productie van latency voor te komen. De sequentiële methode resultaat echter in een hogere latency. Dit kan verklaard worden door de tijdsduur van de upgrade en deze periode geeft mogelijk een correcter beeld van de resultaten, terwijl de groeps- en individuele methode zeer kort zijn. Een mogelijke reden waarom er geen verhoogde

of zelfs hoge latency zichtbaar is, kan zijn omdat de werklading niet hoog genoeg was. De omgeving is bepaald op basis van een lineair model en deze kan naarmate de waardes stijgen verschillen van de reële limiet voor een specifieke resourceconfiguratie.

8 Conclusie

8.1 Overzicht

Het doel van deze thesis was het ontwerpen, implementeren en evalueren van een upgrade planner voor verticale upgrades van multi-tenant SaaS-applicaties in Kubernetes. Het gebruik van verticale upgrades moest zonder extra servicedegradatie voorzien worden, dit gebeurde op basis van de surge hypothese. Deze stelt dat het gelijktijdig upgraden van tenants in groepen een lagere surgekost genereert in vergelijking met het uitvoeren van een upgrade voor elke tenant individueel. De planner biedt de tenants de mogelijkheid tot inspraak over de upgrade van hun applicatieversie en realiseert kostenefficiëntie voor de beheerder. Dit laatste gebeurt door de upgrades uit te voeren met een maximaal aantal tenants zonder dat servicedegradatie optreedt als gevolg van een te hoge surge. Verder wordt de colocatie van containers zoveel mogelijk toegepast om geheugenefficiëntie uit te buiten.

Uit de evaluatie van de upgrade planner blijkt dat de hypothese inderdaad correct is en dat het upgraden van tenants in maximale groepen geen extra servicedegradatie introduceert. Zo ligt de gebruikskost van de groepsmethode tot n keer lager dan bij de sequentiële methode waarbij n gelijk is aan het aantal tenants. Dit enorm verschil komt doordat de duur van de sequentiële upgrades ook n keer langer is. Verder zijn er geen grote verschillen in response latency tussen de verschillende methodes wat aanwijst dat er geen servicedegradatie voorkomt. Hierdoor kan een upgrade met surge best gebruik maken van de groepsmethode omdat dit meer tenants in een kortere periode kan upgraden. Om tot deze conclusie te komen werden er verschillende stappen ondernomen. Als eerste werd er een motiverend experiment uitgevoerd dat het delen van shared libraries in geheugen tussen containers en Pods bewijst. Dit bevestigt de opportuniteit voor colocatie van containers met shared libraries om multi-tenant SaaS-applicaties kostenefficiënt aan te bieden. Vervolgens werd er een upgradeproces en surge-analyse opgesteld. Het upgradeproces is gebaseerd op een dubbele rolling upgrade om het verticaal schalen van Pods op een zo optimaal mogelijk manier uit te voeren. De rolling upgrades produceren surge om de upgrades zonder downtime uit te voeren voor de tenants, hierdoor wordt het vertrouwen van de tenants niet geschaad. Het surge-concept werd via een kostenmodel in formules omgevormd om de totale surge voor de groepsmethode en sequentiële methode te voorspellen volgens een lineair model. Dit initieel onderzoek werd dan vervolgens gebruikt om algoritmes op te stellen die door de middleware gebruikt kunnen worden om maximale groepen te bepalen en optimale omgevingen te genereren. Het onderzoek naar surge samen met het motiverend experiment rond shared libraries duidt aan dat verticale upgrades op een zo kostenefficiënte mogelijke manier uitgevoerd kunnen worden via de groepsmethode. Het zorgt voor een basis waarop de architectuur van de middleware gebouwd kan worden.

De surge hypothese en het experiment rond shared libraries vormen dus de basis voor de architectuur van de upgrade planner. Deze werd ontwikkeld voor en door het Kubernetes platform. De architectuur maakt gebruik van verschillende functionaliteiten van het Kubernetes platform om multi-tenant SaaS-applicaties te kunnen upgraden op een tenant-specifieke basis. Zo vangt Kubernetes de colocatie van Pods op om shared libraries in geheugen te delen, voorziet het faciliteiten om resources te limiteren voor een container en verzorgt het de rolling upgrade van de Pods. De architectuur voorziet een API voor tenants om upgrades te plannen

en voor de beheerder om nieuwe applicatieversies toe te voegen. De middleware probeert dan maximale groepen van tenants te produceren om upgrades volgens het upgradeproces uit te voeren.

Op basis van de architectuur werd er een prototype ontwikkeld om bepaalde aspecten van de architectuur te evalueren. Het prototype voorziet minimale functionaliteiten om een evaluatie uit te voeren op basis van het upgradeproces. Een applicatie werd ontwikkeld om deze evaluatie te faciliteren en bevat een minimale multi-tenant functionaliteit.

8.2 Toekomstig werk

In deze thesis wordt er in het voorgestelde kostenmodel enkel rekening gehouden met CPU om de Pod-plaatsing te garanderen. Als toekomstig werk kan het kostenmodel uitgebreid worden om de impact van secundair gestreste resources zoals geheugen uit te beelden. Het algoritme zou in dit geval aangepast kunnen worden om rekening te houden met zowel de primair gestreste resources als de secundaire gestreste resources. Het huidige kostenmodel houdt verder geen rekening met omgevingen waarin verschillende SLA-klassen zich bevinden, dit vormt een optimalisatieprobleem om Pods te verdelen over verschillende nodes op basis van de gereserveerde CPU-resources.

Daarnaast is Kubernetes bezig met de ontwikkeling van de Kubernetes Vertical Pod Autoscaler die automatisch resources verticaal zal schalen en dit rekening houdend met request- en limiet-verhoudingen. [59] Dit is een geautomatiseerde oplossing voor verticaal schalen maar kan vergeleken worden met het onderzoek uit deze thesis op vlak van de gebruikte resources en Pod-allocatie.

Bibliografie

[1] F. Gey, „Middleware for Customizable Evolution of SaaS applications," KU Leuven, 2017.

[2] S. Neely en S. Stolt, „Continuous Delivery? Easy! Just Change Everything (Well, Maybe It Is Not That Easy)," in *2013 Agile Conference (AGILE)*, Nashville, TN, USA, 2013.

[3] B. Fitzgerald en K.-J. Stol, „Continuous software engineering: A roadmap and Agenda," *Journal of Systems and Software,* nr. Journal of Systems and Software, pp. 176-189, 2017.

[4] F. Gey, D. Van Landuyt, W. Joosen en V. Jonckers, „Continuous Evolution of Multi-tenant SaaS Applications: A Customizable Dynamic Adaptation Approach," in *2015 IEEE/ACM 7th International Workshop on Principles of Engineering Service-Oriented and Cloud Systems (PESOS)*, Florence, Italy, 2015.

[5] „Service Level Agreement," [Online]. Available: https://searchitchannel.techtarget.com/definition/service-level-agreement.

[6] A. Tosatto, P. Ruiu en A. Attanasio, „Container-Based Orchestration in Cloud: State of the Art and Challenges," in *Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*, Santa Catarina, Brazil, 2015.

[7] „Empowering App Development for Developers | Docker," [Online]. Available: https://www.docker.com/.

[8] „Lightweight Container Runtime for Kubernetes," CRI-O, [Online]. Available: https://cri-o.io/.

[9] „Linux Containers," [Online]. Available: https://linuxcontainers.org/.

[10] „containerd," [Online]. Available: https://containerd.io/. [Geopend 18 05 2020].

[11] „Production-Grade Container Orchestration - Kubernetes," [Online]. Available: https://kubernetes.io/.

[12] „Pod Overview," [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/. [Geopend 15 10 2019].

[13] M. Allen, „N+1 redundancy," 21 03 2014. [Online]. Available: https://www.datacenters.com/news/redundancy-n-1-n-2-vs-2n-vs-2n-1.

[14] J. B. Ferreira, M. Cello en J. O. Iglesias, „More Sharing, More benefits? A study of library sharing in container-based infrastructures," in *Euro-Par 2017: parallel processing*, New York, NY, 2017.

[15] M. Rouse, „AUFS," [Online]. Available: https://searchaws.techtarget.com/definition/AUFS-Advanced-Multi-Layered-Unification-Filesystem.

[16] „Use the OverlayFS storage driver," Docker, [Online]. Available: https://docs.docker.com/storage/storagedriver/overlayfs-driver/.

[17] P. Mell en T. Grance, „The NIST Definition of Cloud Computing," p. 7, 2011.

[18] S. Walraven, E. Truyen en W. Joosen, „A Middleware Layer for Flexible and Cost-Efficient Multi-tenant Applications," in *ACM/IFIP/USENIX 12th Internation Middleware Conference*,

Lisbon, Portugal, 2011.

[19] D. Van Landuyt, G. Fatih, E. Truyen en W. Joosen, „Middleware for Dynamic Upgrade Activation and Compensations in Multi-tenant SaaS," in *Service-Oriented Computing: 15th International Conference*, Malaga, Spain, 2017.

[20] G. G. Claps, R. B. Svensson en A. Aurum, „On the journey to continuous deployment: Technical and social challenges along the way," *Information and Software Technology,* nr. 57, pp. 21-31, 2015.

[21] R. Yasrab, „Mitigating Docker Security Issues," *CoRR,* nr. abs/1804.05039, p. 11, 2018.

[22] D. Bernstein, „Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing,* pp. 81-84, 2014.

[23] J. Evans, „What even is a contaienr: namespaces and cgroups," [Online]. Available: https://jvns.ca/blog/2016/10/10/what-even-is-a-container/.

[24] S. Julian, M. Shuey en S. Cook, „Containers in Research: Initial Experiences with Lightweight Infrastructure," in *Proceedings of the XSEDE16 on Diversity, Big Data, and Science at Scale - XSEDE16*, Miami, USA, 2016.

[25] „Docker: The underlying technology," Docker, [Online]. Available: https://docs.docker.com/get-started/overview/#the-underlying-technology.

[26] Y. Li, B. An, J. Ma en D. Cao, „Comparison between Chunk-Based and Layer-Based Container Image Storage Approaches: an Empirical Study," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, San Francisco East Bay, CA, USA, 2019.

[27] „About images, containers, and storage drivers," Docker, [Online]. Available: https://docs.docker.com/storage/storagedriver/#container-and-layers. [Geopend 23 02 2020].

[28] „The Kubernetes API," Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/overview/kubernetes-api/. [Geopend 03 11 2019].

[29] E. Truyen, D. Van Landuyt, V. Reniers, A. Rafique, B. Lagaisse en W. Joosen, „Towards a container-based architecture for multi-tenant SaaS applications," in *Proceedings of the 15th International Workshop on Adaptive and Reflective Middleware - ARM 2016*, Trento, Italy, 2016.

[30] „ReplicationController," Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/. [Geopend 16 10 2019].

[31] „ReplicaSet," Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/.

[32] „Kubernetes Deployment," [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. [Geopend 26 10 2019].

[33] „Canary Deployments in Kubernetes," [Online]. Available: https://kubernetes.io/docs/concepts/cluster-administration/manage-deployment/#canary-deployments. [Geopend 11 12 2019].

[34] „Labels and Selectors," Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/. [Geopend 15 10 2019].

[35] „Managing Compute Resources for Containers,” Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/.

[36] „CPU Kubernetes,” [Online]. Available: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#meaning-of-cpu. [Geopend 27 05 2020].

[37] „Kubernetes Future Resource Types,” [Online]. Available: https://github.com/eBay/Kubernetes/blob/master/docs/design/resources.md#future-resource-types.

[38] „Affinity,” Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#affinity-and-anti-affinity. [Geopend 15 10 2019].

[39] „Configure Liveness, Readiness and Startup Probes,” Kubernetes, [Online]. Available: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/#define-readiness-probes. [Geopend 15 03 2020].

[40] M. Fowler, „Blue Green Deployment,” [Online]. Available: https://martinfowler.com/bliki/BlueGreenDeployment.html.

[41] T. Dumitraş en P. Narasimhan, *Why do Upgrades Fail and What can we do about it,* Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[42] Brewer, „Lessons from giant-scale services,” *IEEE Internet Computing,* 2001.

[43] T. Dumitraş en I. Neamtiu, „Cloud software upgrades: Challenges and Opportunities,” in *2011 IEEE International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA)*, Williamsburg, VA, USA, 2011.

[44] A. Fox en E. Brewer, „Harvest, yield, and scalable tolerant systems,” in *HotOS-VII: 7th Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, USA, 1999.

[45] F. Gey, D. Van Landuyt en W. Joosen, „Middleware for Customizable Multi-staged Dynamic Upgrades of Multi-tenant SaaS Applications,” 2017.

[46] „Managing Resources for Containers,” [Online]. Available: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#how-pods-with-resource-requests-are-scheduled.

[47] D. V. L. W. J. V. J. Fatih Gey, „Continuous Evolution of Multi-tenant SaaS Applications: A Customizable Dynamic Adaptation Approach,” 2015.

[48] X. Ma, L. Baresi, C. Ghezzi, V. Panzica La Manna en J. Lu, „Version-consistent dynamic reconfiguration of component-based distributed systems,” in *the 19th ACM SIGSOFT symposium and the 13th European conference*, Szeged, Hungary, 2011.

[49] „Accessing the API from a Pod,” [Online]. Available: https://kubernetes.io/docs/tasks/access-application-cluster/access-cluster/#accessing-the-api-from-a-pod. [Geopend 17 03 2020].

[50] „Spring Framework,” Spring Team, [Online]. Available: https://spring.io/projects/spring-framework. [Geopend 28 03 2020].

[51] „PostgreSQL: The world's most advanced open source database,” [Online]. Available: https://www.postgresql.org/. [Geopend 18 04 2020].

[52] M. Rouse, „RESTful API,” [Online]. Available: https://searchapparchitecture.techtarget.com/definition/RESTful-API.

[53] „Kubernetes Java API client,” Kubernetes, [Online]. Available: https://github.com/kubernetes-client/java.

[54] „JSON Patch,” [Online]. Available: http://jsonpatch.com/.

[55] „Node.js,” Node.js, [Online]. Available: https://nodejs.org/en/.

[56] „Express - Node.js web application framework,” [Online]. Available: https://expressjs.com/.

[57] „Installing kubeadm,” Kubernetes, [Online]. Available: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/.

[58] W. Delnat, E. Truyen, A. Rafique, D. Van Landuyt en W. Joosen, „K8-scalar: a workbench to compare autoscalers for container-orchestrated database clusters,” in *the 13th International Conference*, Gothenburg, Sweden, 2018.

[59] „Vertical Pod Autoscaler,” Kubernetes, [Online]. Available: https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler.

Appendix

A. Illustratie van de Werking van de Upgrade Planner

De validiteit van de upgrade planner is van cruciaal belang om de experimenten correct te kunnen uitvoeren en te evalueren. De planner hoort het upgradeproces beschreven in sectie 4.1 te volgen en maakt gebruik van de algoritmes uit sectie 4.3 om maximale groepen te bepalen.

Opzet. De planner voert een upgrade uit volgens de experiment setup beschreven in sectie 7.2. Er wordt gebruik gemaakt van een Java applicatie die tijdens het upgradeproces de Kubernetes API elke seconde aanspreekt om requests en limits van de Pods op te vragen. Op het einde van het programma zal deze data in een CSV-formaat opgeslagen worden voor analyse.

Resultaten. De resultaten worden weergeven a.d.h.v. figuren uit de CSV bestanden. De zichtbare data komt overeen met de overgang van *ScaleUp* fase naar neutraal en verder naar *ScaleDown* fase zoals gedefinieerd in het upgradeproces. In Figuur 21 wordt de 3de iteratie van de sequentiële upgrade getoond met in het bovenste deel de upgrade waarin de kleinere versie 2 Pod verwijderd wordt en in het onderste deel de downgrade van de versie 1 Pod. Voor de groepsmethode wordt dezelfde data getoond in Figuur 22 met het grote verschil dat deze maar één iteratie nodig heeft om een volledige omschakeling te volbrengen.

9:58:50	**CPU requests**	**phase**	**lastTransitionTime**	**message**	**reason**	**status**	**type**	**deletionTimeStamp**
mt-api-v1-75c887d78-2m5kb	737.0	Running	2020-05-17T09:56:58.000Z	null	null	True	Initialized	null
mt-api-v2-598c5f5d6d-kwfth	201.0	Running	2020-05-17T09:57:57.000Z	null	null	True	Initialized	null
mt-api-v2-794d79bb96-fqtrk	**134.0**	**Running**	**2020-05-17T09:55:57.000Z**	**null**	**null**	**True**	**Initialized**	**2020-05-17T09:58:39.000Z**
total	1072							
9:58:51	**CPU requests**	**phase**	**lastTransitionTime**	**message**	**reason**	**status**	**type**	**deletionTimeStamp**
mt-api-v1-75c887d78-2m5kb	737.0	Running	2020-05-17T09:56:58.000Z	null	null	True	Initialized	null
mt-api-v2-598c5f5d6d-kwfth	201.0	Running	2020-05-17T09:57:57.000Z	null	null	True	Initialized	null
mt-api-v2-794d79bb96-fqtrk	**134.0**	**Running**	**2020-05-17T09:55:57.000Z**	**null**	**null**	**True**	**Initialized**	**2020-05-17T09:58:39.000Z**
total	1072							
9:58:52	**CPU requests**	**phase**	**lastTransitionTime**	**message**	**reason**	**status**	**type**	**deletionTimeStamp**
mt-api-v1-75c887d78-2m5kb	737.0	Running	2020-05-17T09:56:58.000Z	null	null	True	Initialized	null
mt-api-v2-598c5f5d6d-kwfth	201.0	Running	2020-05-17T09:57:57.000Z	null	null	True	Initialized	null
total	938							
9:58:53	**CPU requests**	**phase**	**lastTransitionTime**	**message**	**reason**	**status**	**type**	**deletionTimeStamp**
mt-api-v1-75c887d78-2m5kb	737.0	Running	2020-05-17T09:56:58.000Z	null	null	True	Initialized	null
mt-api-v2-598c5f5d6d-kwfth	201.0	Running	2020-05-17T09:57:57.000Z	null	null	True	Initialized	null
total	938							

...

9:58:57	**CPU requests**	**phase**	**lastTransitionTime**	**message**	**reason**	**status**	**type**	**deletionTimeStamp**
mt-api-v1-75c887d78-2m5kb	737.0	Running	2020-05-17T09:56:58.000Z	null	null	True	Initialized	null
mt-api-v2-598c5f5d6d-kwfth	201.0	Running	2020-05-17T09:57:57.000Z	null	null	True	Initialized	null
total	938							
9:58:58	**CPU requests**	**phase**	**lastTransitionTime**	**message**	**reason**	**status**	**type**	**deletionTimeStamp**
mt-api-v1-75c887d78-2m5kb	737.0	Running	2020-05-17T09:56:58.000Z	null	null	True	Initialized	null
mt-api-v2-598c5f5d6d-kwfth	201.0	Running	2020-05-17T09:57:57.000Z	null	null	True	Initialized	null
total	938							
9:58:59	**CPU requests**	**phase**	**lastTransitionTime**	**message**	**reason**	**status**	**type**	**deletionTimeStamp**
mt-api-v1-6d866bcb7c-29448	**670.0**	**Pending**	**2020-05-17T09:58:58.000Z**	**null**	**null**	**True**	**Initialized**	**null**
mt-api-v1-75c887d78-2m5kb	737.0	Running	2020-05-17T09:56:58.000Z	null	null	True	Initialized	null
mt-api-v2-598c5f5d6d-kwfth	201.0	Running	2020-05-17T09:57:57.000Z	null	null	True	Initialized	null
total	1608							
9:59:00	**CPU requests**	**phase**	**lastTransitionTime**	**message**	**reason**	**status**	**type**	**deletionTimeStamp**
mt-api-v1-6d866bcb7c-29448	**670.0**	**Pending**	**2020-05-17T09:58:58.000Z**	**null**	**null**	**True**	**Initialized**	**null**
mt-api-v1-75c887d78-2m5kb	737.0	Running	2020-05-17T09:56:58.000Z	null	null	True	Initialized	null
mt-api-v2-598c5f5d6d-kwfth	201.0	Running	2020-05-17T09:57:57.000Z	null	null	True	Initialized	null
total	1608							

Figuur 21: Derde iteratie sequentiële upgrade, overgang van ScaleUp fase naar neutraal en van neutraal naar ScaleDown fase

13:00:43	CPU requests	phase	lastTransitionTime	message	reason	status	type	deletionTimeStamp
mt-api-v1-5b869d4fb4-vrnw6	871.0	Running	2020-05-17T12:54:43.000Z	null	null	True	Initialized	null
mt-api-v2-5d8fd6f8b-7dnjn	**0.0**	**Running**	**2020-05-17T12:54:43.000Z**	**null**	**null**	**True**	**Initialized**	**2020-05-17T13:00:40.000Z**
mt-api-v2-69777779b-kkxsn	871.0	Running	2020-05-17T13:00:00.000Z	null	null	True	Initialized	null
total	1742							
13:00:44	**CPU requests**	**phase**	**lastTransitionTime**	**message**	**reason**	**status**	**type**	**deletionTimeStamp**
mt-api-v1-5b869d4fb4-vrnw6	871.0	Running	2020-05-17T12:54:43.000Z	null	null	True	Initialized	null
mt-api-v2-5d8fd6f8b-7dnjn	**0.0**	**Pending**	**2020-05-17T12:54:43.000Z**	**null**	**null**	**True**	**Initialized**	**2020-05-17T13:00:40.000Z**
mt-api-v2-69777779b-kkxsn	871.0	Running	2020-05-17T13:00:00.000Z	null	null	True	Initialized	null
total	1742							
13:00:45	**CPU requests**	**phase**	**lastTransitionTime**	**message**	**reason**	**status**	**type**	**deletionTimeStamp**
mt-api-v1-5b869d4fb4-vrnw6	871.0	Running	2020-05-17T12:54:43.000Z	null	null	True	Initialized	null
mt-api-v2-69777779b-kkxsn	871.0	Running	2020-05-17T13:00:00.000Z	null	null	True	Initialized	null
total	1742							
13:00:46	**CPU requests**	**phase**	**lastTransitionTime**	**message**	**reason**	**status**	**type**	**deletionTimeStamp**
mt-api-v1-5b869d4fb4-vrnw6	871.0	Running	2020-05-17T12:54:43.000Z	null	null	True	Initialized	null
mt-api-v2-69777779b-kkxsn	871.0	Running	2020-05-17T13:00:00.000Z	null	null	True	Initialized	null
total	1742							

...

13:00:50	CPU requests	phase	lastTransitionTime	message	reason	status	type	deletionTimeStamp
mt-api-v1-5b869d4fb4-vrnw6	871.0	Running	2020-05-17T12:54:43.000Z	null	null	True	Initialized	null
mt-api-v2-69777779b-kkxsn	871.0	Running	2020-05-17T13:00:00.000Z	null	null	True	Initialized	null
total	1742							
13:00:51	**CPU requests**	**phase**	**lastTransitionTime**	**message**	**reason**	**status**	**type**	**deletionTimeStamp**
mt-api-v1-5b869d4fb4-vrnw6	871.0	Running	2020-05-17T12:54:43.000Z	null	null	True	Initialized	null
mt-api-v2-69777779b-kkxsn	871.0	Running	2020-05-17T13:00:00.000Z	null	null	True	Initialized	null
total	1742							
13:00:52	**CPU requests**	**phase**	**lastTransitionTime**	**message**	**reason**	**status**	**type**	**deletionTimeStamp**
mt-api-v1-5b869d4fb4-vrnw6	871.0	Running	2020-05-17T12:54:43.000Z	null	null	True	Initialized	null
mt-api-v1-76448dcdc4-9g9mn	**0.0**	**Pending**	**2020-05-17T13:00:51.000Z**	**null**	**null**	**True**	**Initialized**	**null**
mt-api-v2-69777779b-kkxsn	871.0	Running	2020-05-17T13:00:00.000Z	null	null	True	Initialized	null
total	1742							
13:00:53	**CPU requests**	**phase**	**lastTransitionTime**	**message**	**reason**	**status**	**type**	**deletionTimeStamp**
mt-api-v1-5b869d4fb4-vrnw6	871.0	Running	2020-05-17T12:54:43.000Z	null	null	True	Initialized	null
mt-api-v1-76448dcdc4-9g9mn	**0.0**	**Pending**	**2020-05-17T13:00:51.000Z**	**null**	**null**	**True**	**Initialized**	**null**
mt-api-v2-69777779b-kkxsn	871.0	Running	2020-05-17T13:00:00.000Z	null	null	True	Initialized	null
total	1742							

Figuur 22: Groepsupgrade, overgang van ScaleUp fase naar neutraal en van neutraal naar ScaleDown fase

FACULTEIT WETENSCHAPPEN
Kasteelpark Arenberg 11 bus 2100
3001 LEUVEN, BELGIË
tel. + 32 16 32 14 01
www.kuleuven.be