Experimenten & Evaluatie

De experimenten worden uitgevoerd op een Kubernetes cluster en met als doel het testen van de implementatie en surge hypothese rekening houdend met een SLA garantie voor de tenant. Om de experimenten uit te kunnen voeren zijn er enkele voorgaande stappen nodig die in de eerste twee onderdelen uitgelegd worden. Hierna worden de twee experimenten uitgediept, het doel wordt hier duidelijk en hoe het experiment uitgevoerd gaat worden.

# Pre-experiment: Tenant requests & Limits

## Inleiding

In dit inleidend experiment worden er verschillende resource configuraties uitgevoerd in combinatie met verschillende workloads. Hieruit wordt een resource matrix opgesteld die duidelijk per resource configuratie en user/s de latency weergeeft. Op basis van deze matrix wordt er verder een constante waarde bepaald waarvoor geldt dat deze waarde vermenigvuldigd met een aantal user/s workload voldoet aan de latency SLA voor die user/s. Deze waarde is de maximale workload per user en komt overeen met de limiet voor een Pod in Kubernetes. De request waarde is gelijk aan de average workload en wordt bepaald door een lagere waarde dan de limiet te nemen, in dit geval is dit ≈ 2/3 van de limiet.

## Doel

Het doel omvat het bekomen van een constante waarde per user voor de maximale en average workload die voldoet aan een SLA waarvoor geldt dat per user < 200ms latency is. Hieruit kan dan een omgeving opgesteld worden volgens het upgrade model uit de thesis.

## Tools

* K8-scalar
  + Wordt gebruikt om een workload te genereren.
* Bash script
  + Deze zal gebruikt worden om het hele proces te automatiseren, het zal om de 300s de k8-scalar opstarten voor verschillende resource configuraties en user loads
* Dat-parser
  + Homebrew tool die ervoor zorgt dat run-XX.dat files in verschillende folders omgezet worden naar CSV-formaat, zodat de data gemakkelijkere geanalyseerd kan worden.

## Experiment

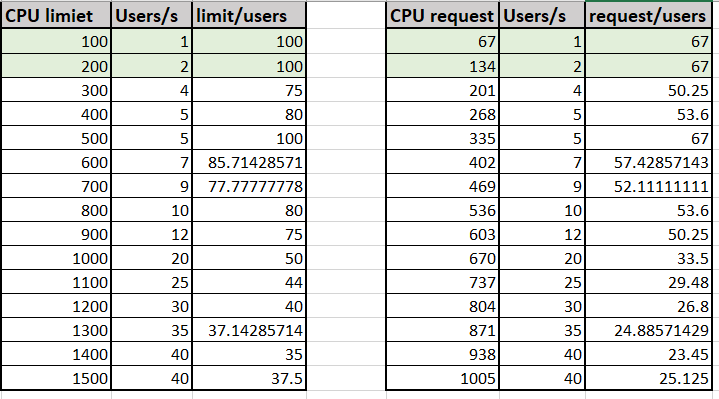
Voor het experiment zal de k8-scalar variërende workloads voor verschillende resource configuraties genereren. De workload wordt uitgevoerd op de testapplicatie waarvan 1 Pod aanwezig is, deze Pod zal dynamisch geüpdatet worden met verschillende resource configuraties. Elke workload zal 300s uitgevoerd worden op de Pod en re resultaten hiervan worden dan geaggregeerd in een resource matrix.

## Resultaten

In figuur 1 is de resource matrix zichtbaar, de groene vakken komen overheen met de user/s (horizontaal) voor de specifieke limiet waarde (verticaal) die voldaan aan de latency SLA van < 200ms. Op basis hiervan wordt er een tabel opgesteld in figuur 2 waar er gekeken wordt hoeveel users/s per limiet mogelijk zijn. De hoogste limiet per user is de constante waarde die gebruikt zal worden om verdere berekeningen uit te voeren. Deze zorgt voor een lineair verloop die geldig is voor de SLA. Op basis van deze eerste berekening kan dan ook de AVG workload bepaald worden door 2/3 van de limiet te nemen.



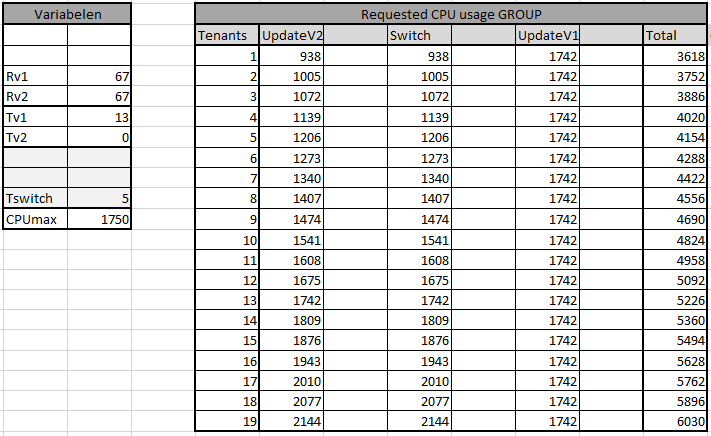
Figuur 1: Resource Matrix



Figuur 2: Limits en requests per user

Door het bekomen van deze resultaten kan er verder een omgeving opgesteld worden die optimaal is om het upgrade proces beschreven in deze thesis uit te testen. Het maximaal aantal tenants kan berekend worden door de formules beschreven in sectie X.

Hieruit blijkt dus dat een omgeving van 13 V1-tenants met de bovenstaande resource configuratie maximaal is onze cluster opzet. Concreet komt deze configuratie overeen met de onderstaande figuur, waar links de tabel met de variabelen is en rechts een overzicht van de totale request kost tijdens elke fase.



Figuur 3: Overzicht omgeving

# Pre-experiment: Verificatie van de implementatie

## Inleiding

Dit experiment dient om de validiteit van de upgradeplanner op de proef te stellen. Het is belangrijk dat deze de upgrades uitvoert zoals eerder beschreven maar ik zal het hier nog eens kort aanhalen. De eerste fase is een upgrade van de nieuwe applicatieversie waar deze meer resources toegekend krijgt. De tweede fase komt overeen met een downgrade van de oude applicatieversie waar deze resources verliest om optimaal te zijn voor het aantal tenants dat er bedient worden. Tussen deze fases is er een korte periode waar er tijdelijk geen verandering gaande is.

## Doel

Het valideren van de functionaliteit van de upgradeplanner zodat experimenten rondom Surge en het upgrade proces uitgevoerd kunnen worden.

## Tools

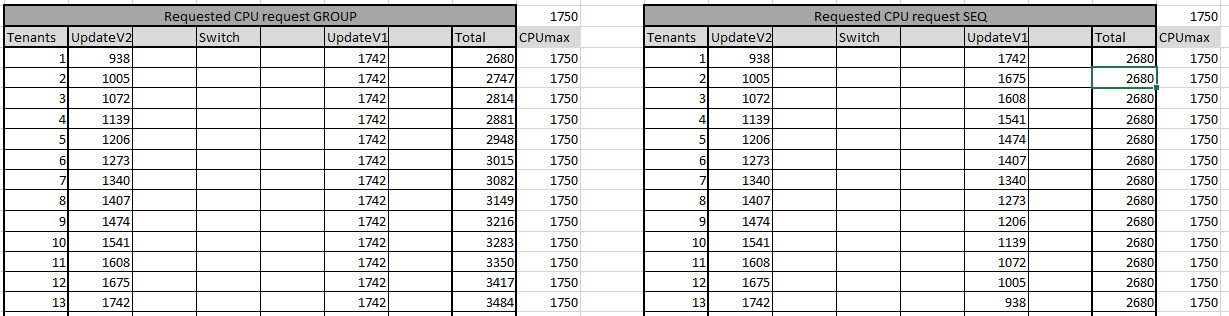
* Upgrade-script
  + Een script dat per tenant of in een groep upgrades uitvoert om zowel de baseline als de groepsmethode te testen
* Surge-validator
  + Homebrew applicatie die informatie van Kubernetes API gebruikt om een CSV bestand te genereren dat data per Pod bevat over huidige requests tijdens elke fase van het upgrade proces.

## Experiment

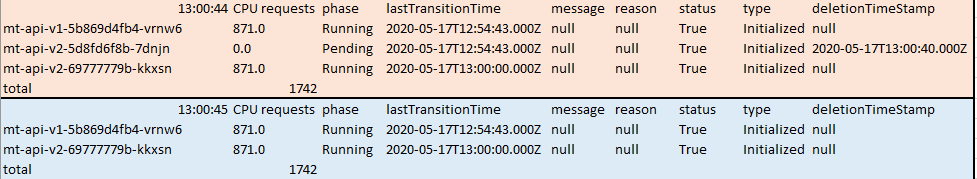
Het experiment wordt uitgevoerd in de omgeving gedefinieerd uit het vorige experiment. Er zal zowel volgens de baseline als groepsmethode upgrades uitgevoerd worden om te zien of ze overeenkomen met de geprojecteerde uitkomsten.

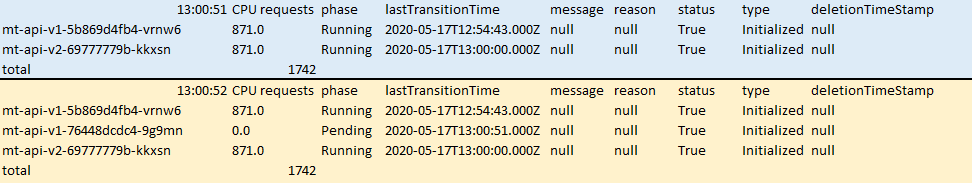
## Resultaten

De verwachte uitkomsten worden berekent door gebruik te maken van de formules die eerder gedefinieerd werden. In de onderstaande figuur ziet u hoe rechts dit voor een groepsupgrade en links voor een sequentiële upgrade eruit ziet.

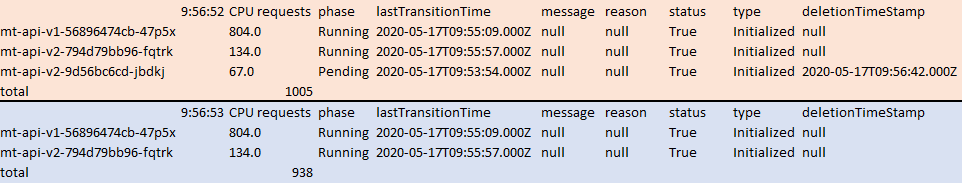


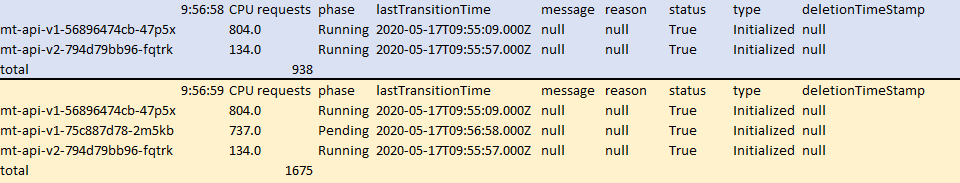
De eerste runs zal via de groepsmethode gaan en er wordt hier telkens een maximale upgrade uitgevoerd, in dit geval zijn dat 13 tenants. In de onderstaande afbeeldingen ziet u snippets van de data die de surge-validator produceert. Het oranje kader is de eerste fase waar een nieuwe applicatieversie meer resources toegekend krijgt. Dit is ook duidelijk zichtbaar in de afbeelding waar de 2de Pod momenteel een deletionTimeStamp heeft en de 3de Pod een grote hoeveelheid resources toegekend gekregen heeft. De blauwe regio is de korte tussenperiode waar er niets gebeurd. Hierna volgt de gele kader die overeenkomt met de tweede fase waar in dit geval de 1ste Pod vernietigd zal worden om de tweede Pod te laten draaien. De totale waardes komen verder ook overeen met de bovenstaande tabel.





Bij de incrementele/sequentiële methode zullen de waardes verschillen afhankelijk van de hoeveelste upgrade getoond wordt. Tijdens deze validatie wordt de tweede upgrade getoond en dit komt overeen met 1005 in de 1ste upgrade fase en 1675 in de 2de downgrade fase. Voor de onderstaande figuren gelden dezelfde regels als hierboven en ook hier zien we dat de planner dezelfde waarden produceert als de tabel.





# Experiment 1: Resource gebruik

## Onderzoeksvraag

* Worden resources gespaard? Wanneer werkt deze techniek het beste en wanneer totaal niet?
  + Accumulatieve CPU usage resource kost over tijd

## Inleiding

Het doel van dit experiment is het analyseren van de totale kost van beide upgrade methodes. De baseline methode zal 13 tenants upgraden volgens de incrementele methode (één per één). De groep methode zal 13 tenants op basis van maximale groepsgrootte tenants van versie 1 naar versie 2 migreren. Tijdens deze migratie worden tenants onder een AVG workload gezet.

## Doel

Het doel is om een best-case scenario voor de groepsmethode te tonen. Het totale resource gebruik bij de incrementele methode zal hoger liggen dan tijdens de groepsmethode vooral door het tijdsaspect van de upgrade.

## Tools

* Upgrade planner
  + Zal zowel de incrementele als groep upgrades verzorgen, dit wordt gedaan door tijdelijk groepen te vormen op basis van identieke deadlines. Voor de incrementele methode wordt er per tenant een verschillende deadline opgesteld, terwijl voor de groepsmethode alle deadlines identiek zijn.
  + Tijdens de Switch phase van het upgrade proces zal de test-applicatie verwittigd worden van de tenants die geüpgraded zijn.
* K8-scalar
  + De k8-scalar is aangepast geweest om genoeg tenants User classes te hebben
  + Verder zal de k8-scalar van service veranderen wanneer een Tenant een request stuurt en een 301 status code terugkrijgt. Zo wordt een upgrade gesimuleerd met de wisseling van user-flow.
    - Het gebruik van verschillende runs is niet meer nodig hierdoor omdat de k8-scalar zelf dynamisch users van versie zal wisselen.
  + De requests die uitgevoerd worden door k8-scalar Tenants is gelijk aan 3 requests per keer, dit representeert de natuurlijke flow van login, home en info. Register wordt ook initieel eenmalig uitgevoerd.
* Grafana
  + Grafana wordt gebruikt om de CPU-usage per Pod te meten.

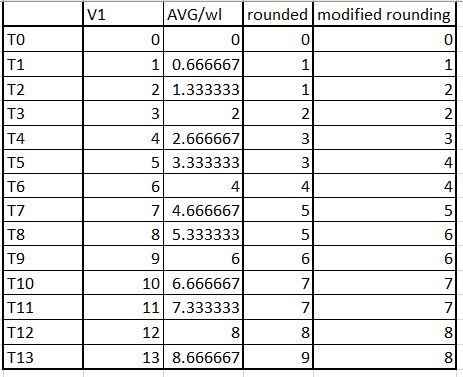
## Opzet

Het experiment wordt uitgevoerd op een single node met één Pod per versie. Deze Pods bezitten over labels om ze te matchen met de correcte service en een Readiness Probe die ervoor zorgt dat de container zeker klaar is voor ontvangst. De Pods hun resource configuratie volgt deze van de pre-experimenten.

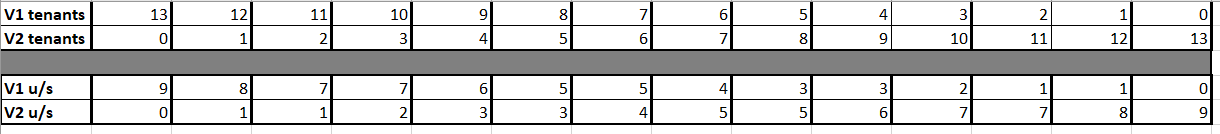
Om een average workload te genereren wordt er inventief gebruik gemaakt van de k8-scalar zijn methodes. Voorgaand hebben we al bepaald dat de maximale workload voor de resource-configuratie 13 users/s is. De average workload bedraagt dan ≈ 8 users/s dit is de peak load die ingesteld zal worden in het experiment.properties bestand die door de experiment-controller gebruikt wordt.

Zoals in het Tools onderdeel vermeld, is de k8-scalar aangepast geweest om automatisch te veranderen van service wanneer een implementatie van een Tenant van versie wisselt. Een restrictie die de k8-scalar heeft is dat er minimaal 1 user/s load moet zijn voor elke user-implementation die gedefinieerd wordt, in dit geval is de average workload echter niet gelijk aan het aantal tenants. Concreet is het probleem dat i.p.v. 13 user/s workload er nu maar telkens 8 user/s op hetzelfde moment requests versturen. Om ervoor te zorgen dat zowel V1 als V2 hun average workload krijgen tijdens het upgrade proces moeten upgrades in een bepaalde volgorde gepland worden.

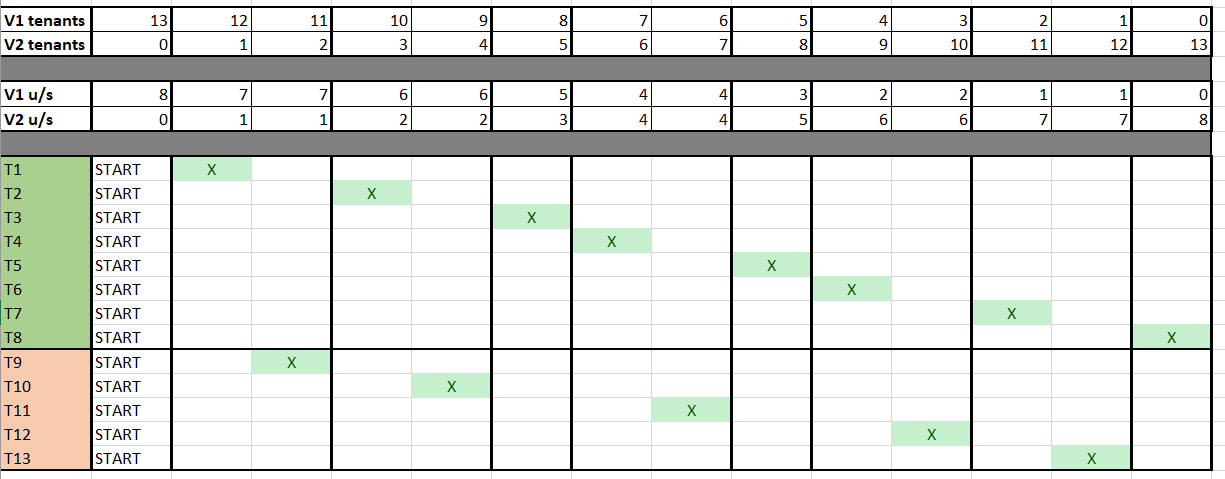
In figuur 4 ziet u de berekening voor de average workload per aantal tenants in een versie. Dit komt natuurlijk uit op niet-gehele getallen, met standaard afronding verkrijgt u de reeks die zichtbaar is in de kolom **“rounded”**. Echter een probleem met deze reeks is dat het met de huidige k8-scalar methode het niet mogelijk is om beide versies op hetzelfde moment de correcte AVG workload aan te bieden zoals te zien in Figuur 5. Daarom stel ik een licht gemodificeerde rounding voor zichtbaar in de kolom “**modified rounding”** die beide versies op elk moment van de upgrade kan voorzien van ≈ hun AVG workload. In Figuur 6 is het zichtbaar welke tenant voor elke stap (kolom) van het upgrade proces een upgrade zal uitvoeren zodat de AVG workload gegarandeerd kan worden.



Figuur : Berekening AVG workload



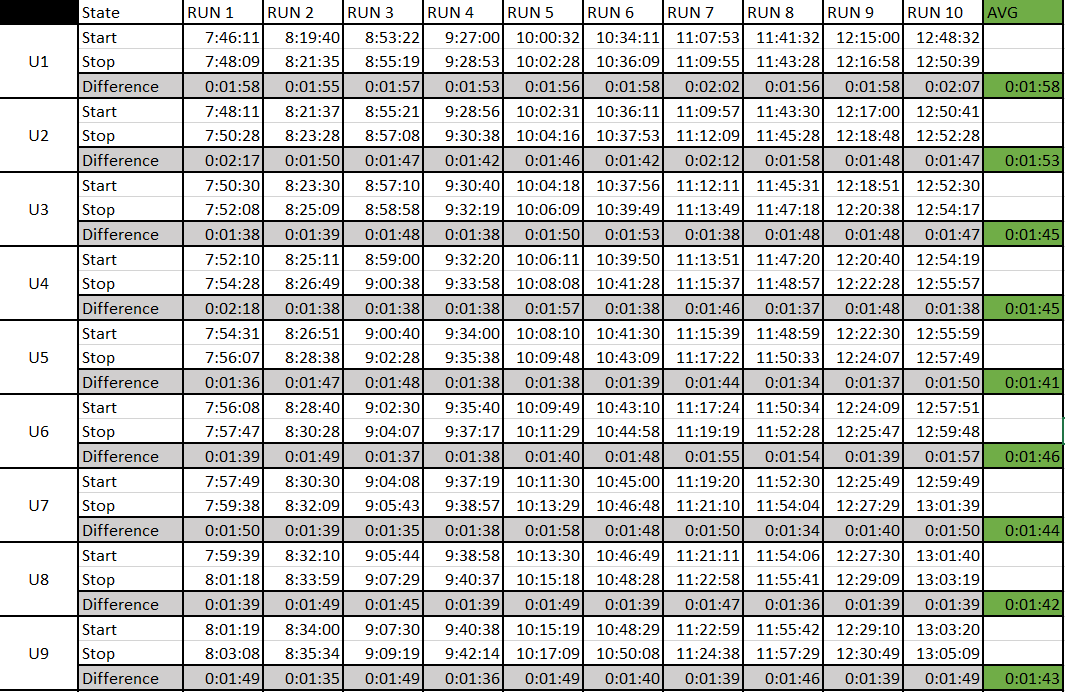
Figuur : Breakdown tenant upgrades per versie versus workload per upgrade



Figuur : Aangepaste breakdown upgrade per tenant om AVG workload te garanderen

Figuur 6 toont de volgorde aan van de tenant upgrades om ervoor te zorgen dat er een AVG workload per versie gegarandeerd kan worden. Deze zal ook in het experiment gevolgd worden.

Verder was het belangrijk om een gemiddelde upgrade tijd te bepalen voor de applicatieversie. Deze is uitgevoerd en op basis van informatie uit de upgrade planner opgezet. De upgradeplanner geeft de start en eindtijd van elke upgrade terug en dit geeft een ruw beeld op de avg upgrade tijd. Op basis hiervan zal dan een tabel opgezet worden voor elke upgrade die uitgevoerd wordt zoals te zien in onderstaande afbeelding.



Door gebruik te maken van grafana data kan dan elk punt in de grafana data verbonden worden aan een specifiek update moment. Op basis hiervan kan dan vergeleken worden wat de totale CPU kost is voor een volledige upgrade.

## Experiment

Het experiment zelf verloopt via het ExperimentSurge.sh script en volgt deze stappen:

1. Database van Upgradeplanner met 13 tenants in te laden
2. Database van Upgradeplanner met 2 deployments in te laden (mt-api-v1 en mt-api-v2)
3. Database van test-applicatie in te laden met 13 tenants
   1. De tenant-ids in deze applicatie zijn identiek aan die van de planner
4. **[OPTIE]** Inladen van het correcte experiment.properties in de experiment-controller
   1. **Incrementeel**
      1. User peak is constant 8
      2. Lange duration
         1. Berekent op basis van het gemiddeld aantal tijd nodig om upgrades uit te voeren voorheen.
         2. 13 \* 100 seconden = 1300s
         3. Korte voor en na periode als baseline van 120 seconden
   2. **Groep**
      1. Peak is ook constant 8
      2. Korte duration
         1. Berekent op basis van het gemiddeld aantal tijd nodig om upgrades uit te voeren voorheen.
         2. 1 \* 100 seconden = 100s
         3. Korte voor en na periode als baseline van 120 seconden
5. **[OPTIE]** Het opstellen van de upgrades en deadlines voor de upgradeplanner met een vaste constante waarde die zorgt voor vertraging op de deadlines
   1. **Incrementeel**
      1. Op basis van een python script dat voor elke user in het JSON bestand de deadline +10 seconden doet t.o.v. van de vorige user. De eerste deadline wordt bepaald door huidige tijd + constante waarde
   2. **Groep**
      1. Zal door gebruik van sed (stream-editor) de deadlines voor al de gebruikers gelijk stellen door huidige tijd + constante waarde te doen
6. Uitvoeren van de experiment-controller in de achtergrond
7. Het posten van de upgrade aan de upgradeplanner om na verloop van tijd de upgrades te starten

Het experiment wordt een 5-tal keer herhaald om te zien of de data relatief gelijk loopt en overeen komt met elkaar. Er wordt per run redelijk wat data bijgehouden zoals de Pod requests door gebruik te maken van de surge-validator, de tijd van start en einde van elke upgrade door de upgradeplanner en het gebruik via grafana.

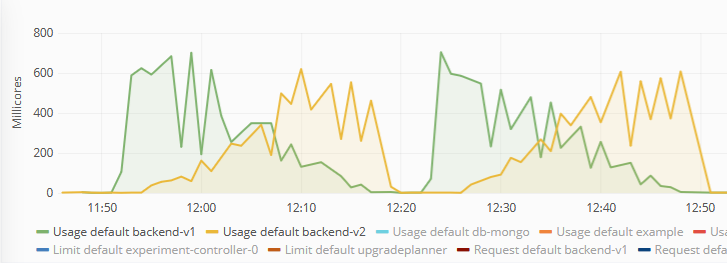
## Resultaten

// NOG MEE BEZIG

Ik verwacht hier te zien dat het totale CPU gebruik over de tijd heen lager zal liggen bij de groepsmethode dan bij de incrementele/sequentiële methode. De conclusie hieruit volgt dan dat doordat resource gebruik lager ligt en voor een korte periode is dit interessanter voor de beheerder. Zo zal de applicatie ook beter de SLA kunnen garanderen voor een langere periode.

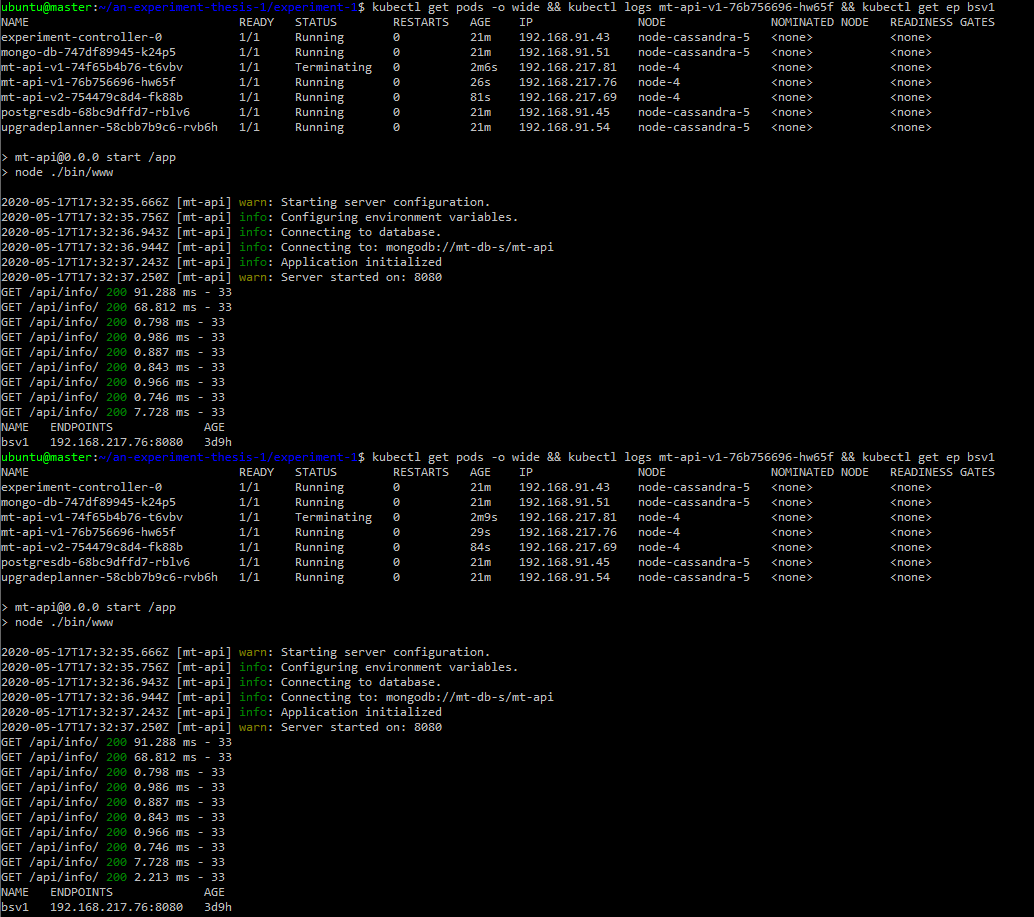
## Problemen

Tijdens de preliminaire experiment test runs stootte ik op een eigenaardigheid tijdens het meten van de CPU usage. Hier werden telkens kleine pieken geproduceerd terwijl een relatief lineaire lijn verwachtte. In de onderstaande figuur ziet u dit van een aantal upgrades die uitgevoerd werden via het script en allemaal met dezelfde configuratie.



Initieel dacht ik dat ik iets fout geconfigureerd had waardoor er downtime werd geproduceerd. Maar hierna probeerde ik het met een Deployment die geen Probes heeft en/of andere geavanceerde instellingen, en dit werd ook eens uitgevoerd via de kubectl om gewoon een rolling upgrade te starten. Dit bleek hetzelfde effect te hebben op de Grafana metingen en ging ik op verder onderzoek uit om dit toch proberen te valideren of er iets fout zat met mijn Deployment configuratie.

In de onderstaande figuur ziet u 3 commando’s die uitgevoerd worden, eerst worden al de huidige Pods getoond, hierna de logs van de nieuwe Pod die zojuist opgestart is en als volgt de endpoints van de service die deze Pod hoort te bedienen.



De /api/info call is voor de Readiness Probe om te controleren of deze Pod ready is. Maar zoals zichtbaar in de afbeelding roep ik tweemaal deze commando’s op. Beide keren is de eerste Pod aan het terminaten maar krijgt de tweede nog altijd geen traffic (buiten de Readiness Probe) tot de oude Pod 0/1 ready is. Het vreemde hieraan is dat het zichtbaar is dat de endpoint wel degelijk al richting de nieuwe Pod wijst.

Het lijkt dus dat de oftewel de service pas verstuurd wanneer de eerste Pod afgesloten (0/1) is oftewel er iets fout gaat met het verzenden van de requests vanuit de k8-scalar waardoor requests gebonden zijn aan het vorige endpoint i.p.v. het nieuwe. Ik heb Cédric een gelijkend experiment laten uitvoeren zonder mijn configuratie om te zien of het aan mijn configuratie lag maar het lijkt dus eerder aan Kubernetes te liggen.

Uit de SLA data lijkt het wel dat alles in orde dus de pieken worden vooral geproduceerd doordat de nieuwe Pod wat tijd nodig heeft om CPU usage te genereren. Deze stijgt naarmate meer requests ontvangen worden en tijdens deze initiële overgang is het aantal requests nog laag. Ik neem aan dat dit geen probleem vormt maar wou dit toch even langs u laten gaan om hier zeker van te zijn.

# Experiment 2: Latency & QoS violaties

## Onderzoeksvraag

Is er een impact op de latency tussen de twee methodes wanneer upgrades tijdens een high workload uitgevoerd worden?

Als resources vrijgehouden moeten worden om tenants gelijktijdig te laten upgraden dan worden bepaalde gereserveerde resources niet gebruikt. Is deze kost niet groter dan de resources die gespaard moeten worden?

## Inleiding

Upgrades gebeuren echter niet enkel tijdens AVG workload maar kunnen ook tijdens een high workload uitgevoerd worden. Tijdens een high workload staat de CPU onder zware stress en wanneer via de groepsmethode een upgrade gestart wordt zal er een grote hoeveelheid CPU toegekend worden aan de nieuwe Pod. Dit zal hoogstwaarschijnlijk een impact hebben op de CPU Scheduler hoe die CPU tijd verdeelt onder de Pods.

## Doel

Het doel van dit experiment is het meten van latency tijdens upgrades van high workloads en te meten of tijdens een groepsupgrade de impact op de latency hoger ligt dan tijdens de kleinere incrementele upgrades.

## Tools

* Upgrade planner
  + Zal zowel de incrementele als groep upgrades verzorgen, dit wordt gedaan door tijdelijk groepen te vormen op basis van identieke deadlines. Voor de incrementele methode wordt er per tenant een verschillende deadline opgesteld, terwijl voor de groepsmethode alle deadlines identiek zijn.
  + Tijdens de Switch phase van het upgrade proces zal de test-applicatie verwittigd worden van de tenants die geüpgraded zijn.
* K8-scalar
  + De k8-scalar is aangepast geweest om genoeg tenants User classes te hebben
  + Verder zal de k8-scalar van service veranderen wanneer een Tenant een request stuurt en een 301 status code terugkrijgt. Zo wordt een upgrade gesimuleerd met de wisseling van user-flow.
    - Het gebruik van verschillende runs is niet meer nodig hierdoor omdat de k8-scalar zelf dynamisch users van versie zal wisselen.
  + De requests die uitgevoerd worden door k8-scalar Tenants is gelijk aan 3 requests per keer, dit representeert de natuurlijke flow van login, home en info. Register wordt ook initieel eenmalig uitgevoerd.
* Grafana
  + Grafana wordt gebruikt om de CPU-usage per Pod te meten.

## Experiment

Het experiment wordt uitgevoerd in de volgende stappen via een script, deze biedt de optie om incrementeel of volgens de groepsmethode upgrades te starten. Dit doet het door:

1. Database van Upgradeplanner met 13 tenants in te laden
2. Database van Upgradeplanner met 2 deployments in te laden (mt-api-v1 en mt-api-v2)
3. Database van test-applicatie in te laden met 13 tenants
   1. De tenant-ids in deze applicatie zijn identiek aan die van de planner
4. **[OPTIE]** Inladen van het correcte experiment.properties bestand in de experiment-controller
   1. **Incrementeel**
      1. User peak is constant 13
      2. Lange duration
         1. Berekent op basis van het gemiddeld aantal tijd nodig om upgrades uit te voeren voorheen.
         2. 13 \* 100 seconden = 1300s
         3. Korte voor en na periode als baseline van 120 seconden
   2. **Groep**
      1. Peak is ook constant 13
      2. Korte duration
         1. Berekent op basis van het gemiddeld aantal tijd nodig om upgrades uit te voeren voorheen.
         2. 1 \* 100 seconden = 100s
         3. Korte voor en na periode als baseline van 120 seconden
5. **[OPTIE]** Het opstellen van de upgrades en deadlines voor de upgradeplanner met een vaste constante waarde die zorgt voor vertraging op de deadlines
   1. **Incrementeel**
      1. Op basis van een python script dat voor elke user in het JSON bestand de deadline +10 seconden doet t.o.v. van de vorige user. De eerste deadline wordt bepaald door huidige tijd + constante waarde
   2. **Groep**
      1. Zal door gebruik van sed de deadlines voor al de gebruikers gelijk stellen door huidige tijd + constante waarde te doen
6. Uitvoeren van de experiment-controller in de achtergrond
7. Het posten van de upgrade aan de upgradeplanner om na verloop van tijd de upgrades te starten

Hierna zal de data uit de controller gehaald worden voor verdere analyse.

## Resultaten

// Nog bezig hieraan

Hier ben ik niet zeker hoe ik de data moet representeren om conclusies hieruit te kunnen trekken. Moet ik gewoon de latencies letterlijk vergelijken?