|  |  |
| --- | --- |
| Metrics |  |
| Input rate | #Records / second |
| Taskmanagers | #Taskmanagers |
| Latency | Seconds |
| Throughput | #Records / second |
| Backpressure | Ms |
| CPU\_load | CPU utilization |
| Lag | #Records |
| Busy time | Ms |
| Idle time | ms |

Overall future work:

* How will the system handle 10 times as many autoscalers? Autoscalers that adapt when the optimal amount of autoscalers changes will most likely adapt much more often as there are a lot more possibilities to chose from.

Query 1

# Dhalion

|  |  |  |  |
| --- | --- | --- | --- |
| **Autoscaler** | **Latency** | **Taskmanagers** | **Rescale actions** |
| Dhalion-1 | 0.999 | 2.5 | 13 |
| Dhalion-5 | 1.6 | 2.4 | 13 |
| Dhalion-10 | 2.3 | 2.5 | 14 |

## Parameters

|  |  |
| --- | --- |
| Cooldown | 120s |
| Overprovisioning\_factor | 20% |
| **Backpressure\_threshold** | 100ms |
| **CPU\_threshold** | 60% |
| Scaling\_factor\_percentage | 20% |
| **Latency\_threshold** | 1s, 5s, 10s |

## Scaling decisions

**Underprovisioning, displayed as red in the graphs:**

* **Kafka lag is increasing**
* **Average event time lag >** Latency\_threshold **= 1s, 5s, 10s**
  + **Don’t trigger on noise**

**Overprovisioning, displayed as green in the graphs:**

* Maximum backpressure < backpressure\_threshold = 100ms
* CPU\_utilization < CPU\_threshold = 60%
* Event\_time\_lag < = THRESHOLD
  + Event\_time\_lag <= 0.2 \* Input rate
  + Average lag <= 0.2 \* input rate

Scale-down factor: scale down by Scaling\_factor\_percentage = 0.2

Scale-up factor: amount of time date consumption suspended / time normal processing

TODO

It is difficult to visualise the threshold for the event\_time\_lag, as the final results only saved the total\_event\_time\_lag. Because of that, we rewrite the threshold in the following way and display it in the graph:

## Results



Chart

Description automatically generated Figure 1 Q1 Dhalion 1

Chart

Description automatically generatedFigure 2 Q1 Dhalion 5

Chart

Description automatically generated

Figure 3 Q1 Dhalion 10

Chart, line chart

Description automatically generated

Figure 4 Q1 Dhalion 1,5,10

## Configuration comparison

Comparing the three autoscaler configurations, the biggest difference is the speed in which they choose to change the parallelism of the operators. Dhalion-1 here has the lowest threshold with the latency being 1.

Comparing configuration 1s, 5s and 10s.

The threshold chosen to deviate between is the latency threshold, which was set on 1, 5 and 10 seconds to detect under provisioning. The higher the threshold, the more latency should be build up before under provisioning is detected. Looking at the overall latency, the average amount of task managers and the number of scaling is similar between the configurations. The average latency, however, is significantly lower when the threshold is set at 1s than with 5 and 10 seconds. One of the reasons for this is that when not under provisioning or scaling, the latency is always close to zero. Only when under provisioning is occurring, the latency goes up. The lower the threshold it must pass, the faster under provisioning is detected. This is first detected with the threshold set at 1s, allowing for fast action to counter the decrease in latency. When setting this threshold higher, it will take longer before action is taken, resulting in more latency. While setting the threshold higher would make the autoscalers more resilient to temporary changes, the results point out that the solution is already resilient to this when set at 1 second. Investigating whether the threshold could potentially be set even lower would potentially make the solution even more efficient.

## Results of the best Dhalion configuration (1s)

For some reason, after the first scaling operation, it takes around five minutes to scale down again. This is strange, as the optimal amount of taskmanagers is around 3,4 around this time, while 6 taskmanagers are employed. One explanation for this is that the current CPU load is still relatively high, most likely just above 60% (blue line). When dropping below it, the autoscaler immediately scales down. Most likely, the CPU metric is most likely used to distinguish between over-provisioning and perfect provisioning. Though, CPU\_usage does not really seem to drop below 50%, this threshold might not been set that properly.

### Under provisioning detection

**Underpovisioning is detected using to checks:**

* **The average event time lag >** Latency\_threshold
* **The lag is increasing (over 1 minute timeframe)**

#### Average event time lag > Latency\_threshold

We already discuss the average event time lag in the comparison with the three different Dhalion setups, using 1s, 5s and 10s as latency threshold. Here we noticed that the latency was generally close to zero, only surpassing the threshold when under-provisioning. The higher the threshold, the more resilient the system is to noise, but the less time it takes for the system to respond to average event time lag.

#### The lag is increasing (over 1 minute timeframe)

The second check is that the lag is increasing. This increase is measured over a 1 minute timeframe. While this is a logical check, as the lag should be increasing when under provisioning, it might be vulnerable to noise. Investigating the lag, one can see that the lag is continuously switching between increasing and decreasing due to random changes in the distribution. This makes the check somewhat unreliable. Investigating whether the 1 minute timeframe can be increased to ensure a more reliable check can be interesting.

#### Conclusion

In total, under-provisioning is detected four times. All three when the input\_rate was increasing. This successfully ensured the overall latency remained low while maintaining minimal taskmanagers. From the two checks, the latency\_threshold check seems to have most influence on the behavior of the detector, as the lag increase check is vulnerable against noise.

### Overprovisioning detection

Overpovisioning is detected using three thresholds:

* Average CPU-load <= 60%
* Average lag should be smaller than 20% of the input\_rate
* Maximum Backpressure < 100ms

#### Average CPU-load <= 60%

CPU load is not a very good indication of overprovisioning. It generally takes a long time time before detecting overprovisioning. After the first scale-down operation, the autoscaler does not do anything for 5-6 whole minutes (period between first two red vertical lines). This is due to the CPU load being above the threshold of 60%. Still, at this point the system is very much overprovisioning with 6 CPUs. In addition, after increasing at 60 minutes, it again takes another 15 minutes before overprovisioning is detected and a scale-down operation is triggered.

#### Average lag should be smaller than 20% of the input\_rate

The average lag was an interesting metric, as it was the only one providing some indication of a recent scaling operation. As scaling requires the newly arriving events being buffered in the Kafka queue, the average lag goes up significantly after a scaling operation. Only when the system managed to catch up with the queued events, the average lag goes below the threshold again. This makes it a good indication whether the system is ready for another scaling operation, potentially making an explicit cooldown period unnecessary. Beside the period after a scaling operation, the lag was generally close to the threshold, occasionally surpassing it and then going below it again. This effect was somewhat caused by the fact that the threshold was set to be 20% of the input\_rate and both the input\_rate and lag have a positive correlation. Futher investigating the scaling operations, one would notice that the final trigger for a scaling operation was generally caused by the CPU\_load diving under 60%. Though, when the input\_load was increasing, the lag was generally above the threshold, ensuring that no overprovisioning was detected. In addition, there was one case in which a scaling operation was triggered while the lag was still above threshold (the 6th scaling operation).

#### Maximum Backpressure < 100ms

The average backpressure is also used as an indication of overprovisioning. With When the input\_rate increases, the average backpressure indeed goes up, eventually surpassing the threshold. Though, when looking at the backpressure rates, most of the backpressure seems to be caused by the overhead of the scaling operations. While the backpressure was in the beginning close to zero when more task managers were online because of the initial start, when the system starts to deploy only 3 or 4 autoscalers, the backpressure remains high and is never fully worked away. Requiring the backpressure to first go down more, might result in less backpressure and therefore less latency. This, will however also, most likely, also result in the deployment of more autoscalers adding additional costs to the system.

### Conclusion

Overall, the system does a good job at distinguishing system over provisioning. While at start-up it takes a somewhat long time before the system is at its optimal 3-4 taskmanagers. This is due to the CPU-load being higher than the threshold for some time. Maybe removing or changing this check might result in better performance at the beginning at the run. An alternative solution for this would be to set a higher scale-down factor, allowing for faster scaling operations at the beginning. Setting this value too high, though, might result in scaling down too much, requiring an additional scale-up operation, adding additional overhead.

The remaining scale-down operations were on logical moments and the system was able to effectively distinguish overprovisioning from effective provisioning.

## Future work

* **Investigate effect of latency threshold < 1s**
* **Difference between using avg(CPU) and avg(latency) and total or maximum CPU or latency.**
* **Investigate the effect of increasing the period over which the lag should be increasing in when detecting underprovisioning**

# Kubernetes HPA

|  |  |  |  |
| --- | --- | --- | --- |
| **Autoscaler** | **Latency** | **Taskmanagers** | **Rescale actions** |
| HPA-50 | 0.443 | 6.2 | 31 |
| HPA-70 | 0.611 | 3.9 | 19 |
| HPA-90 | 0.921 | 2.8 | 13 |

## Parameters

|  |  |
| --- | --- |
| CPU Utilization | 50%, 70%, 90% |

## Scaling Decisions

HPA determines the ideal number of taskmanagers using the following equation:

]

Every 15 seconds (default configuration) the desired\_taskmanagers is calculated and recorded. Every 5 minutes (downscale-stabilization\_period, 5 minutes is the default configuration), all recommendations of the previous 5 minutes are checked, and the highest recommendation is chosen. This means scaledowns occur gradually, smoothing out the impact of rapidly fluctuating metric values.

## Results

A sheet of music

Description automatically generated with medium confidence

Figure 5 Q1 HPA - 50

A sheet of music

Description automatically generated with medium confidence



Figure 6 Q1 HPA 70

A sheet of music

Description automatically generated with medium confidence

Figure 7 Q1 HPA 90

Chart

Description automatically generated

Figure 8 Q1 HPA - 50 70 90

## Configuration comparison

Comparing the three configurations (50%, 70% and 90%), one can notice the general trend of the more task managers online, the lower the latency. From the configurations, the 50% had the lowest latency, but the most taskmanagers, while the 90% had the highest latency, but the least taskmanagers.  
Overall, the range of amount of taskmanagers in which the configurations operate is for the 50% between 1 and 8 taskmanagers, for the 70% between 1 and 5 and or the 90% between 1 and 3.

The higher the target CPU load is, the higher the overall CPU load of the node. By setting a high CPU target, the system ensures nodes are assigned more work before deciding to scale up. This ensures more cost-effective use of the nodes. As a downside to this is that the system is less resilient against noise and will decide to scale up later, resulting in additional lag and latency. Another downside is that the autoscaler will decide faster to scale down, potentially resulting in the system taking more time to get rid of existing backpressure. This negative effect is, however, somewhat countered by the autoscaler issuing less scaling operations. This will introduce less lag caused by pausing the operators.

The lower the target CPU is, the faster scaling operations are triggered. When setting the target CPU at 50%, the system switches between 1 and 8 taskmanagers, issuing a total of 31 scaling operations. While this results in less latency, the autoscaler is strongly overprovisioning and also introducing a lot of additional overhead due to the abundant amount of scaling operations. Extending the gradual-scaledown period would counter this effect and would bundle more scaling operations together, reducing the scaling overhead and making the system more efficient. Though, this would not change anything to the fact that the autoscaler is still strongly overprovisioning.

Overall, the CPU-target of 70% is a good balance between setting the CPU target too high or too low. While it would be interesting to compare its effect against 60% and 80% (maybe also 75% and 80%), instead of the extreme cases of 50% and 90%, the autoscore has significantly less scaling operations and still a very low latency.



## Results of the best HPA configuration (70)

Further investing the effectiveness of HPA with a target CPU-utilization of 70%

Overall, the autoscalers works quite efficiently, switching between 1 and 5 taskmanagers. While the optimal amount of taskmanagers appears be between 2 and 3, it is still rather efficient with on average using 3.9 taskmanagers and having an average latency of 0.611.

Looking at the graph, one would notice a small scaling operation (marked with a red circle), that went from 2 taskmanagers to 1, after which it immediately scaled up again. While the autoscaler assumed scaling down one more node would still result in sufficient processing power, the scaling overhead resulted in an immediate scale-up. This made the scaling operation online increase tot latency, making the scaling operation inefficient

Another thing that can be noted is that it takes a significant amount of time before the first scaling operation is triggered. While it might have to do something with the downscale-stabilization\_period of 5 minutes, it is still a bit strange as the average CPU load was well below 70% for that period.

# Varga1 and Varga2

|  |  |  |  |
| --- | --- | --- | --- |
| **Autoscaler** | **Latency** | **Taskmanagers** | **Rescale actions** |
| Varga1-0.3 | 0.651 | 14.9 | 44 |
| Varga1-0.5 | 2.1 | 14.8 | 59 |
| Varga1-0.7 | 1.5 | 14.1 | 62 |
| Varga2-0.3 | 0.375 | 3.5 | 13 |
| Varga2-0.5 | 1.0 | 2.6 | 11 |
| Varga2-0.7 | 3.1 | 2.3 | 9 |

## Parameters

|  |  |
| --- | --- |
| Stabilization window | 120s |
| Threshold | 50000 records |
| Utilization | 0.3, 0.5, 0.7 |
| Cooldown | 2 minutes |

## Scaling Decisions

Vargav leverages the HPA framework in combination with two custom metrics.

RelativelagChangeRate is calculated in the following way:

The target rate of the realtiveLagChangeRate is 1, as this indicates that the system can keep up with the load.

Utilization represents the time in milliseconds that an operator is idle. A task can be idle for two reasons:

1. There is no data to process
2. The data is bottlenecked by a down-stream process

Utilization is calculated in the following way:

We visualise the utilization threshold in the plots by using the idleTimeMsPerSecond metric. The threshold is visualised by:

So with utilization being 0.3, 0.5, 0.7, we plot the threshold at idle\_time = 0.7, 0.5, 0.3 respectively

The relativeLagChangeRate both requires the derivative of the lag and the KafkaConsumer\_record\_consumed\_rate. As they are both unavailable in the data of the experiments, the relativeLagChangeRate is not plotted.

To get an idea of what the lag\_rate would be, the average value is set at 1, which would mean the lag is unchanging or decreasing.

## Results

A picture containing chart

Description automatically generated

Figure 9 Q1 Varga1 0.3

A picture containing graphical user interface

Description automatically generated

Figure 10 Q1 Varga1 0.5

Graphical user interface

Description automatically generated

Figure 11 Q1 Varga1 0.7

Chart

Description automatically generated

Figure 12 Q1 Varga2 0.3

Chart

Description automatically generatedFigure 13 Q1 Varga2 0.5



Chart

Description automatically generated

Figure 14 Q1 Varga2 0.7

Chart

Description automatically generated

Figure 15 Q1 Varga1 Combined

Chart

Description automatically generated

Figure 16 Q1 Varga2 Combined

## Varga 1

All three runs of Varga 1 are constantly overprovisioning, sitting on the maximum amount of 16 taskmanagers. When the system decides to scale down to a more reasonable amount of taskmanagers, the lag resulting from the scaling operation, strongly decreasing the idle time, and thereby triggering a scale-up operation. This results in a cascade of scale-up events that is eventually stopped when reaching the maximum amount of taskmanagers.

Every 8 minutes, a downscale operation is performed. This is due to the set stabilization window, in which every 15 seconds, the desired Replicas are recorded. After the stabilization window, the highest recommendation is chosen and a down-scale is triggered. Introducing a cooldown for the scale-up operations would result in a more efficient system (hence Varga2).

## Varga 2 – configuration comparison

Varga2 performs much better than varga1. The lower the utilization threshold, the more taskmanagers are used, resulting in a lower latency. The difference in parameter is the target utilization. The lower the target utilization is set, the less work is assigned to the different taskmanagers. While the system succeeds at setting the average utilization of both the 0.3 and 0.5 configuration around their threshold, the system has trouble maintaining the 0.7 utilization. After a scaling operation, the utilization goes up to 0.7 (idle\_time goes down to 0.3) due to the lag caused by the scaling operation. This lag is, however worked way relatively fast, making the utilization go down again. The 0.3 configuration has a low latency, but a higher taskmanager use. On the other hand, the 0.5 configuration has a higher average latency but uses less taskmanagers. Depending on what you find more important in your system, one could argue that both configurations are good and choosing the best one is more dependent on user preference. Still the average amount of taskmanagers of the 0.3 configuration is quite high and the latency of the 0.5 is quite high. An average utilization goal between these two values would most likely result in an even better autoscaler. For now we will choose the 0.5 configuration for additional investigation, as the average amount of taskmanagers is closer to the desired amount of taskmanagers.

## Results of the 0.5 Varga2 configuration

The autoscaler waits a long time before scaling down for the first time. This is caused by the stabilization window set at 480 seconds. While the default value of the stabilization window is set at 300 seconds. The larger the window, the longer the system will wait to invoke a scale-down operation, limiting the scaling overhead. As a downside, the autoscaler has less flexibility when adapting to changing datastream characteristics. Investigating the effect of setting a different stabilization window can be interesting, as it is unclear why the stabilization window was initially set at 480 seconds.

Another interesting aspect is the scale-up occurring after the first scale down period. This period suggests that the scale-up cooldown period is too small, as it immediately decides to scale-up because of the previous scale-down operation which removed 5 of the 8 taskmanagers. This scale-down operation was unnecessary as the build-up lag was already being worked away quite fast. Scaling up again only increased the lag and extended the overall latency of the application. As it is not clear why the scale-up cooldown period is set at 120 seconds, and why it is different than the scale-down cooldown period is set at 480, further investigating this scale-up cooldown period would potentially result in a more efficient autoscaler.

**TODO: it might not really be a scale-down cooldown of 8 minutes, as some scale-down operations follow each other faster than that.**

As described in the Varga implementation, problems may arrive with the HPA implementation when the auto-scaler can keep up with the number of records and there is no lag. While in this case it might be appropriate to scale down, the relativeLagChangeRate is set at its desired value, suggesting the same amount of desired replicas as are currently online. When using multiple metrics, HPA always picks the largest amount of desired replica’s, not allowing the system to scale down. To prevent this, the system defines the relativeLagChangeRate metric to only be positive when above 50000 records. If it is below 50000 records, only the utilization metric is considered, allowing for scale-down operations. Investigating the data, downscale operations seem generally to be immediately triggered after the lag drops below 50000 records. This suggests that when only taking the utilization threshold into consideration, the system would want to scale down much earlier. As the lag is generally fluctuating around 100000 records, it can take a long time before the threshold of 50000 records is reached. Setting this threshold at around 100000 records, would take the utilization metric more into consideration and can potentially lead to more efficient behavior. This behavior would explain why the taskmanager takes a long time before scaling down when the input rate is dropping.

One weird behavior, though, is the scale-down operation of the 0.7 configuration occurring around the 60 minutes mark. Here a scale-down operation is triggered way before the lag is below its threshold of 50000. This would mean that when the lag is dropping, the relativeLagChangeRate is below one, suggesting that a scale-down operation can be invoked. Also, as the utilization also requests a scale-down operation, a scale-down is performed relatively fast.

## Future work

* For Varga1 and Varga2 a stabilization window of 480 seconds is chosen. This is different from the default of 300 seconds. Why is this chosen and what would be the effect of setting a different one.
* The scale-up cooldown period is different from the scale-down cooldown period and might result in inefficient behavior. Further investigating this would potentially result in a more efficient autoscaler.
* Setting the lag threshold higher would put more weight in the utilization metric. Investigating the effect of this could result in faster scale-down operations and therefore a smaller amount of taskmanagers. Investigating whether this would also lead to a higher latency would point out whether this is desirable.

# DS2-Original and DS2-Updated

|  |  |  |  |
| --- | --- | --- | --- |
| **Autoscaler** | **Latency** | **Taskmanagers** | **Rescale actions** |
| DS2-original-0 | 203.6 | 1.7 | 8 |
| DS2-original-33 | 3.3 | 2.6 | 8 |
| DS2-original-66 | 2.4 | 3.0 | 8 |
| DS2-updated-0 | 71.3 | 1.8 | 7 |
| DS2-updated-33 | 2.4 | 2.5 | 8 |
| DS2-updated-66 | 2.4 | 2.8 | 8 |

## Parameters

|  |  |
| --- | --- |
| Cooldown | 240 seconds |
| Over\_provisioning factor | 0%, 33%, 66% |
| *Scale\_up\_lag\_threshold* | 5s |
| *Scale\_down\_lag\_threshold* | 1s |

## Scaling Decisions

DS2 calculates the optimal amount of taskmanagers using the following equation:

For the experiment the following metrics are used:

## Results

Chart

Description automatically generated

Figure 17 Q1 DS2-original 0%

Graphical user interface, chart

Description automatically generated

Figure 18 Q1 DS2-original 33%

A sheet of music

Description automatically generated with medium confidence

Figure 19 Q1 DS2-original 66%

Chart

Description automatically generated

Figure 20 Q1 DS2-original Combined

## DS2-original parameter comparison

The parameter that was changed was the overprovisioning\_factor. The overprovisioning\_factor is a multiplier for the optimal parallelism for each operator. When overprovisioning, more taskmanagers are employed which can deal with temporal lag spikes better, making scaling up unnecessary. Looking at the three configurations, one can notice the few scale-up events that occur: only 8, of which 4 occur in the beginning. When the input\_rate is increasing only a single scale-up action is performed and when the input\_rate is decreasing only a single scale-down action is performed.

Comparing the three configurations, one can notice that the configuration with 0% overprovisioning factor fluctuates between 1 and 2 nodes and is therefore constantly under provisioning. This results in a large amount of lag resulting in a immense average latency.

The 33% overprovisioning configuration fluctuates between 2 and 3 nodes, allowing the system to keep up with the input rate and effectively catching up after a scaling operation. The 66% over provisioning configuration fluctuates between 2 and 4 nodes, somewhat overprovisioning. While this results in a smaller latency, it also requires a bit more taskmanagers to be online.

After the first scale-down operation, the autoscalers decide to scale up again. This is due to the increase in lag due to the operations overhead. While the 33% overprovisioning configuration only scales up a single task manager, the 66% overprovisioning scales up two taskmanagers. This makes the operation even more expensive than it has to be, as it scales down again a few minutes later after the lag is gone. For this reason, one can argue that in this experiment, the 33% overprovisioning configuration performed the best.

## DS2-original 33% overprovisioning

As mentioned before, after the first scale-down activity, the autoscaler decides to scale up again. This is likely caused by the scaling overhead of the scale-down activity.