|  |  |
| --- | --- |
| 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 |

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

|  |  |  |
| --- | --- | --- |
|  |  |  |

## 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 (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.

# 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: