experiment

October 16, 2023

1 Test Inspektor Gadget scaling

The goal of this experiment is to test until which number of nodes Inspektor Gadget scales and to also collect its CPU usage.

1.1 Experimental environment

1.1.1 Hardware environment

Inspektor Gadget will be tested on an AKS cluster using `Standard_DS2_v2` as node Azure image (*i.e* 2 CPU cores and 7 GB of DRAM). The number of nodes will vary over the experience with the following values: 2, 12, 25, 37 and 50.

1.1.2 Software environment

AKS runs kubernetes 1.25 and relies on kernel 5.15.

To test scaling, the idea is to deploy a pod on each node which will generate a lot of `exec()` and to count the number of `exec()` events reported by Inspektor Gadget. So, on each node, a pod running `stress-ng` is deployed to generate as many `exec()` it can during 1 second on all the node CPU (in our case 2). Before that, Inspektor gadget is deployed on the cluster to monitor all these pods (which belong to the same namespace). The experiment is described in `script.sh`.

1.1.3 Performance statistics

For each number of node, the above experiment is run 30 times to compute some statistics.

1.1.4 Resource statistics

For each Inspektor Gadget pod, we will collect the following statistics:

1. The whole CPU usage in microseconds, before the experiment is run.
2. The whole CPU usage in microseconds, after the experiment is run.
3. The current memory footprint in bytes, after the experiment is run.

All these statistics are collected using `cgroupv2` `cpu.stat` and `memory.current` file. By substracting the CPU usage after to the CPU usage before, we would get the CPU usage for the experiment. Note that, this is not possible to get the peak memory footprint, as `memory.peak` is only available in kernel 5.19.

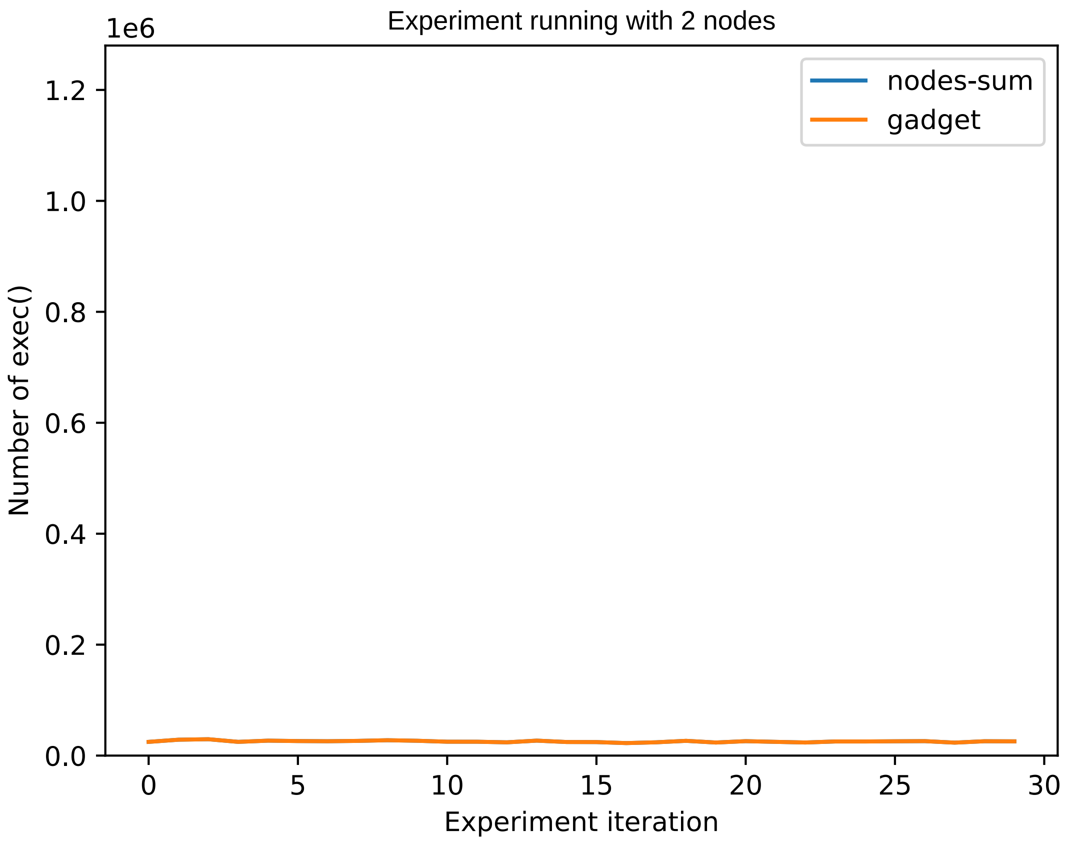
1.2 Results

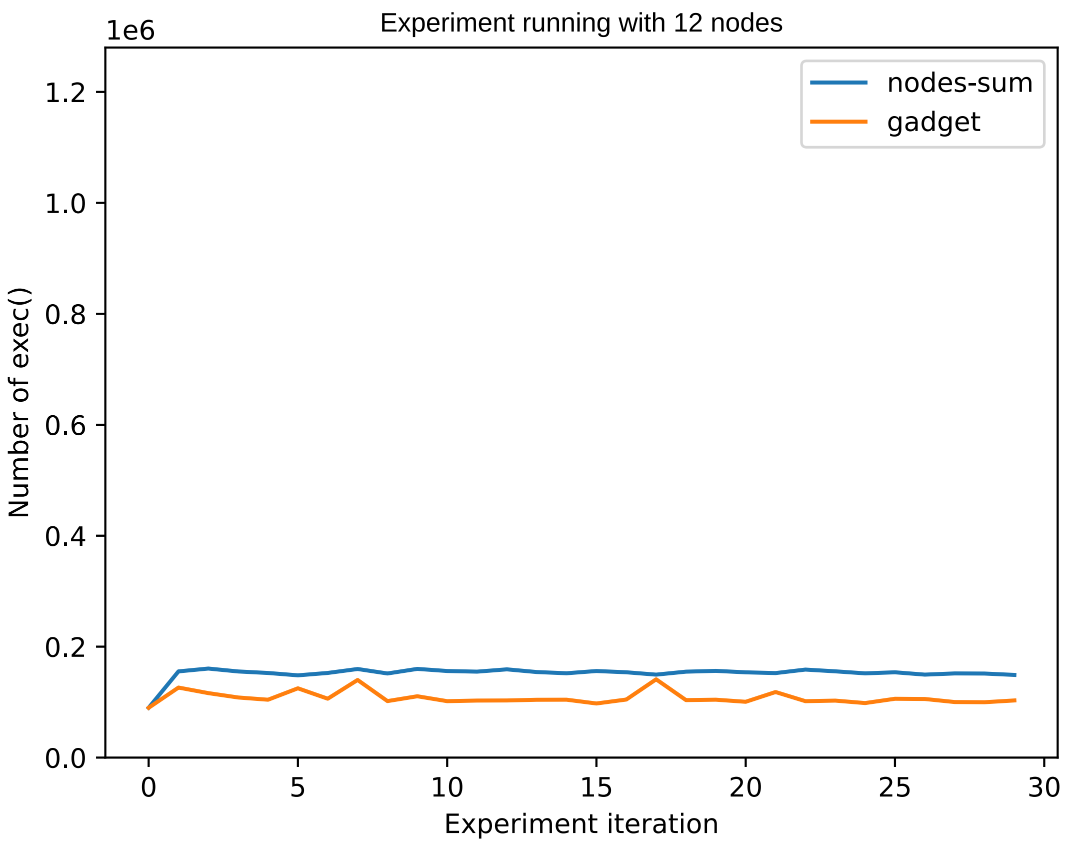
1.2.1 Scaling results

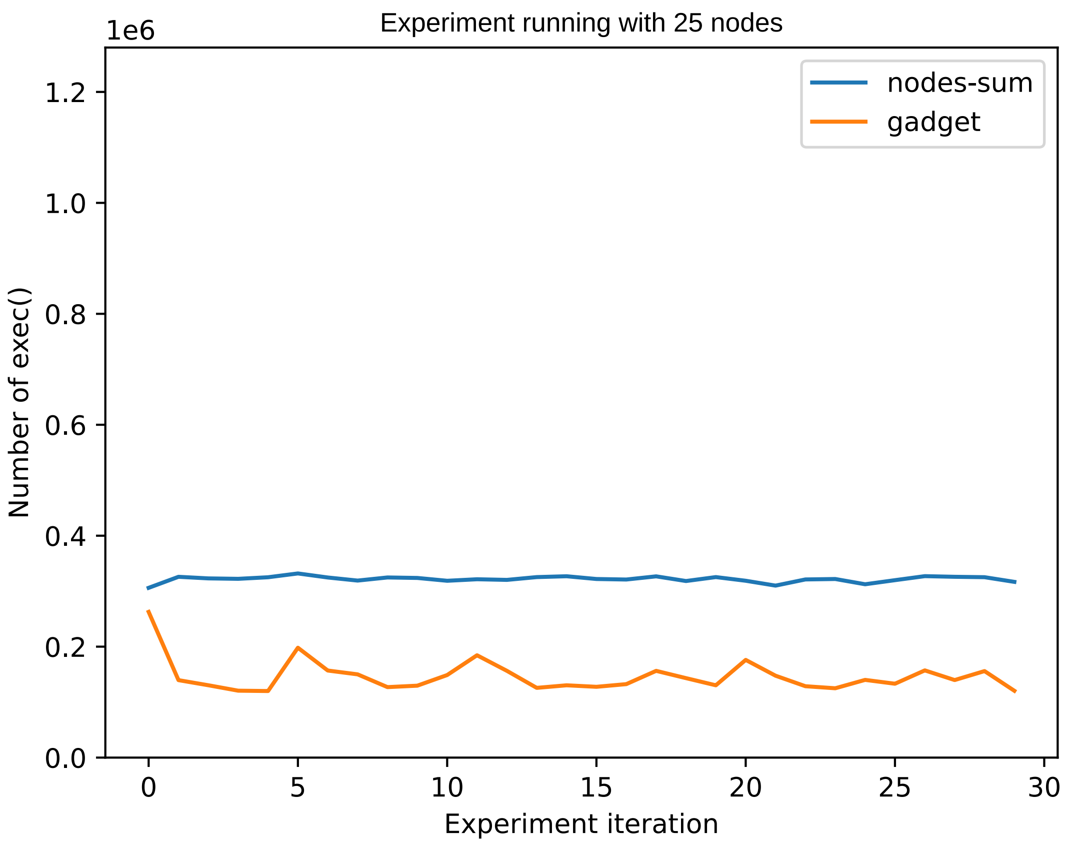
The results are depicted in the following graphs. The `x` axis indicates the experiment number from 1 to 30 while the `y` axis indicates the number of `exec()`. Each graph presents two curves:

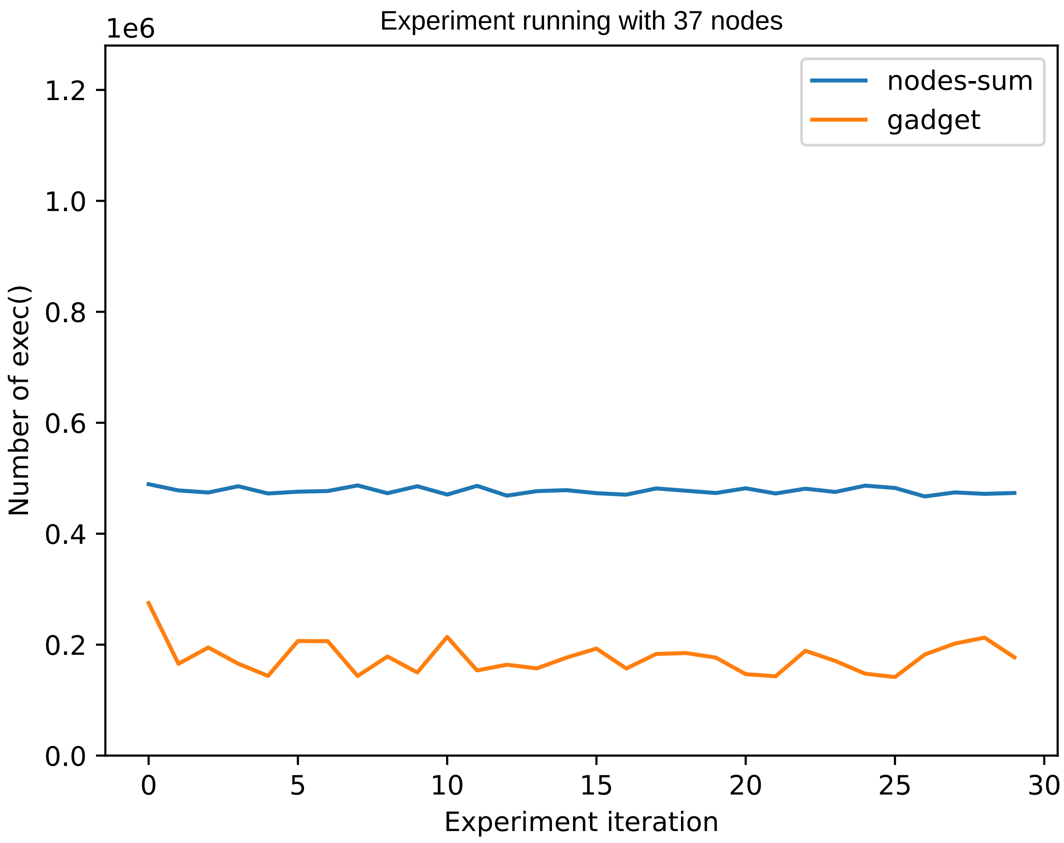
1. One which is the sum of all `exec()` generated for all nodes.
2. The other which is the number of `exec()` reported by Inspektor Gadget.

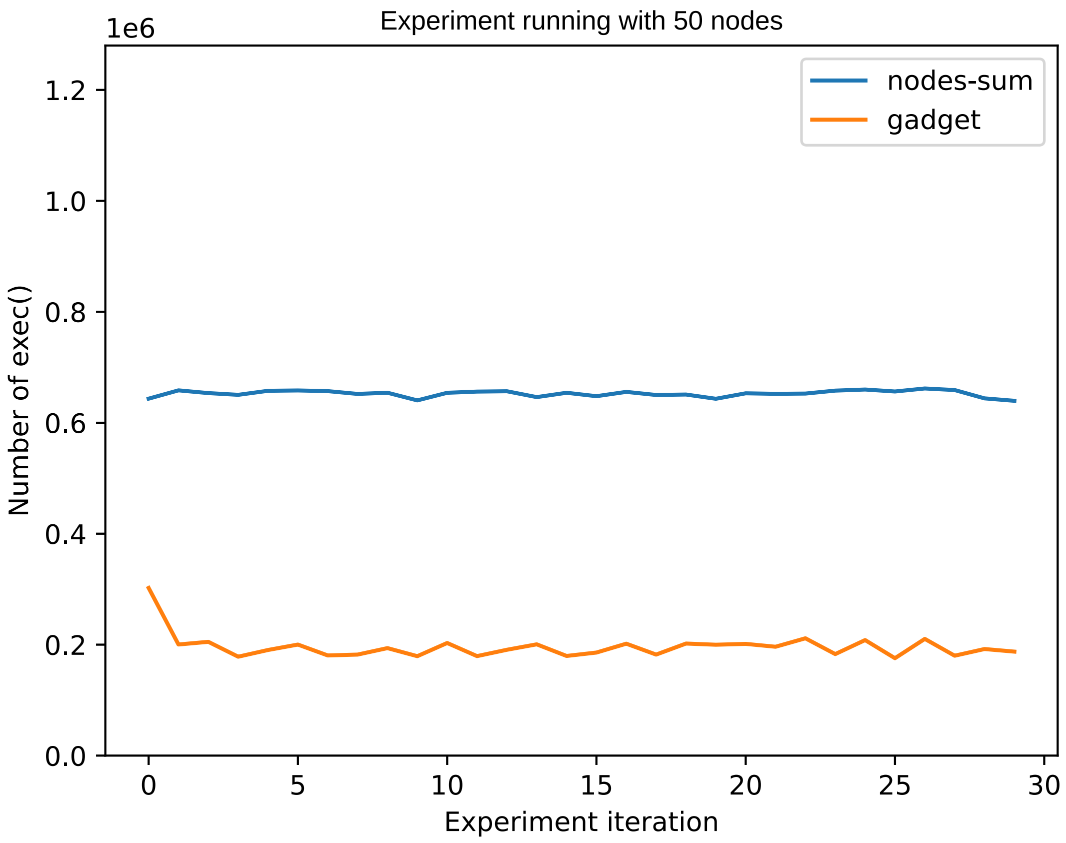
[10]:
```
!pip install pandas > /dev/null
!pip install matplotlib --upgrade > /dev/null
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib_inline.backend_inline


matplotlib_inline.backend_inline.set_matplotlib_formats('svg')

NODE_NUMBERS = [2, 12, 25, 37, 50]
dataframes = []

for node_nr in NODE_NUMBERS:
    df = pd.read_csv(f'horizontal-13-10-23-12-54-04/exec-{node_nr}-nodes.csv')
    df['nodes-sum'] = df.filter(regex = 'node').sum(axis = 1)

    dataframes.append(df)

for i in range(len(NODE_NUMBERS)):
    dataframes[i][['nodes-sum', 'gadget']].plot()
    plt.title(f'Experiment running with {NODE_NUMBERS[i]} nodes',␣
 ↪font='Liberation Sans')
    plt.xlabel('Experiment iteration')
    plt.ylabel('Number of exec()')
    plt.ylim(0, 1280000)
    plt.show()
```


Experiment running with 2 nodes
1e6
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Number of exec()
0
5
10
15
20
25
30
Experiment iteration
nodes-sum
gadget


Experiment running with 12 nodes
1e6
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Number of exec()
0
5
10
15
20
25
30
Experiment iteration
nodes-sum
gadget


Experiment running with 25 nodes
1e6
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Number of exec()
0
5
10
15
20
25
30
Experiment iteration
nodes-sum
gadget


Experiment running with 37 nodes
1e6
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Number of exec()
0
5
10
15
20
25
30
Experiment iteration
nodes-sum
gadget



1.2.2 Interpretation

First of all, the curves are quite flat, so we can conclude there are not so much variation across the iterations.

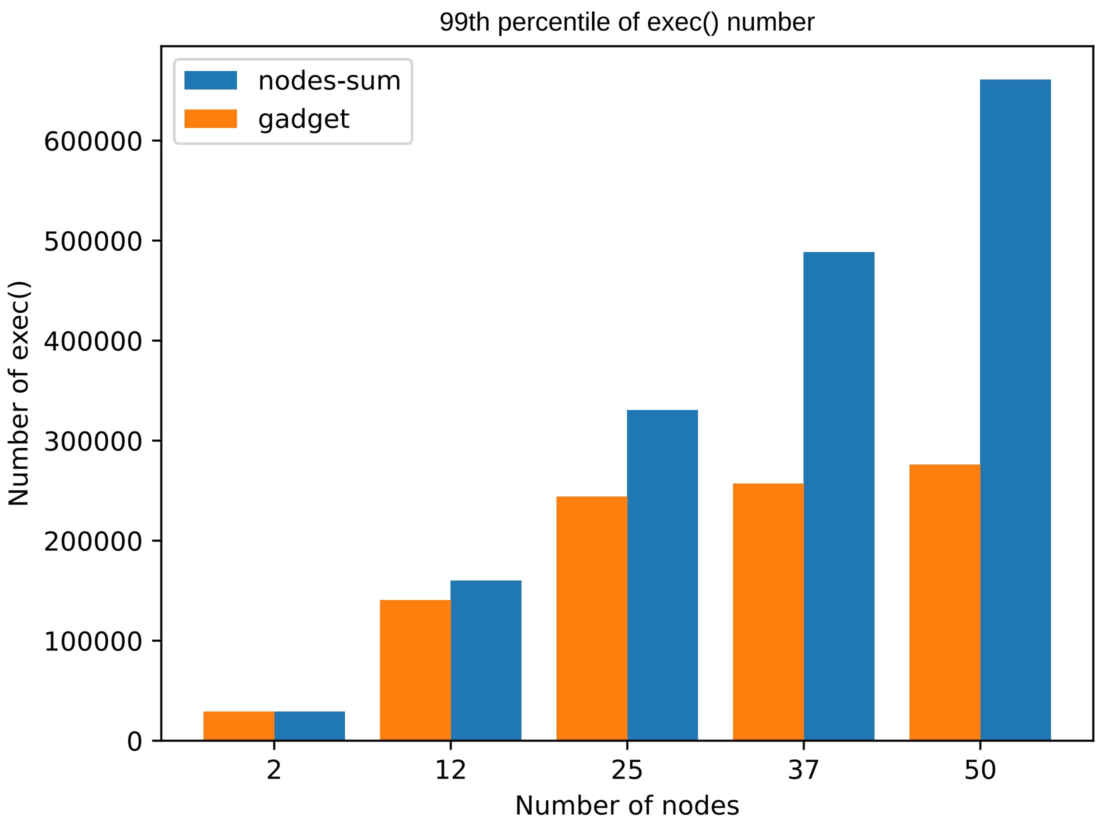
There are nonetheless two iterations with 37 nodes where Inspektor Gadget failed but the number `stress-ng` pods also failed. During the experiment, some `context canceled` and `timeout` errors were generated, but this a minority of case.

We can also see there are some iterations where Inspektor Gadget reports less events than generated when run on 50 nodes but the difference is small.

The number of events reported by Inspektor Gadget follows the number of `exec()` generated.

Let's compute the 99th percentile for all number of nodes:

[6]:
```python
import numpy as np

percentiles = {'nodes-sum': [], 'gadget': []}

for i in range(len(NODE_NUMBERS)):
    # 99th percentile
    quantile = dataframes[i][['nodes-sum', 'gadget']].quantile(0.99)
    percentiles['nodes-sum'].append(quantile['nodes-sum'])
    percentiles['gadget'].append(quantile['gadget'])

df = pd.DataFrame(percentiles)
nodes_sum = df[['nodes-sum']].to_numpy().ravel()
gadget = df[['gadget']].to_numpy().ravel()

nodes_sum_fig = plt.bar(range(len(nodes_sum)), nodes_sum, align='edge', width=0.
 ↪4, label='nodes-sum')
gadget_fig = plt.bar(range(len(gadget)), gadget, align='edge', width=-0.4,␣
 ↪label='gadget')

plt.legend()
plt.xticks(np.arange(len(NODE_NUMBERS)), NODE_NUMBERS, rotation=0)
plt.xlabel('Number of nodes')
plt.ylabel('Number of exec()')
plt.title('99th percentile of exec() number', font='Liberation Sans')

plt.show()
```



With the 99th percentile, we can clearly see that Inspektor Gadget scales.

Let's now take a look to the resource consumption of Inspektor Gadget.

1.2.3 Resources results

As the result across different nodes are quite stable, we will only compute statistics over resource consumption on the experiment with 50 nodes.

Let's compute the 99th percentile of CPU usage during the stress, the memory footprint after the stress for each node and also over all nodes:

[11]:
```
from IPython.display import display, Markdown

def to_sec(microsecond):
    return microsecond / 10 ** 6

def to_megabytes(bytes):
    return bytes / 1024 / 1024

nodes_nr = 50
dataframes = {}
quantiles = {}

# We need index_col = False because the CSV file contains a "," at the end of␣
↪each line.
# This is not correct, but read_csv can handle it with this option.
df_cpu_before = pd.read_csv(f'horizontal-13-10-23-12-54-04/
↪cpu-before-{node_nr}-nodes.csv', index_col=False).filter(regex = 'node')
df_cpu_after = pd.read_csv(f'horizontal-13-10-23-12-54-04/cpu-{node_nr}-nodes.
↪csv', index_col=False).filter(regex = 'node')

dataframes['cpu'] = df_cpu_after - df_cpu_before
dataframes['memory'] = pd.read_csv(f'horizontal-13-10-23-12-54-04/
↪memory-{node_nr}-nodes.csv', index_col=False).filter(regex = 'node')

for key in dataframes:
    quantiles[key] = dataframes[key].quantile(0.99)

markdown_array = ''
for node in quantiles['cpu'].index:
    memory = quantiles['memory'][node]
    cpu = quantiles['cpu'][node]

    markdown_array += f'|{node}|{to_sec(cpu):.
↪2f}|{int(to_megabytes(memory))}|\n'

all_cpu = to_sec(quantiles['cpu'].quantile(0.99))
all_memory = int(to_megabytes(quantiles['memory'].quantile(0.99)))
```

```
markdown_array += f"|all nodes|{all_cpu:.2f}|{all_memory}|\n"
markdown_array = f"""
|node| CPU usage (s) | Memory footprint (MB) |
|----|----------------|-------------------------|
{markdown_array}
"""

display(Markdown(markdown_array))
```

node	CPU usage (s)	Memory footprint (MB)
node-1	1.63	135
node-2	1.62	131
node-3	1.58	127
node-4	1.60	133
node-5	1.58	126
node-6	1.61	133
node-7	1.65	138
node-8	1.61	134
node-9	1.62	127
node-10	1.61	135
node-11	1.62	132
node-12	1.66	134
node-13	1.67	129
node-14	1.57	125
node-15	1.60	132
node-16	1.68	132
node-17	1.65	131
node-18	1.66	126
node-19	1.71	138
node-20	1.69	135
node-21	1.69	132
node-22	1.67	136
node-23	1.76	131
node-24	1.74	135
node-25	1.71	130
node-26	1.73	126
node-27	1.72	134
node-28	1.73	136
node-29	1.81	133
node-30	1.73	133
node-31	1.68	136
node-32	1.74	131
node-33	1.79	134
node-34	1.82	135
node-35	1.71	137
node-36	1.75	138

node	CPU usage (s)	Memory footprint (MB)
node-37	1.82	128
node-38	1.77	136
node-39	1.78	134
node-40	1.83	130
node-41	1.81	131
node-42	1.87	139
node-43	1.87	137
node-44	1.84	133
node-45	1.88	139
node-46	1.86	139
node-47	1.87	134
node-48	1.81	135
node-49	1.85	136
node-50	1.84	145
all nodes	1.88	142

1.2.4 Interpretation

As we can see, the 99th percentile CPU usage on all nodes is 1.61 seconds. This number is higher than the duration of the experiment, which was one second. This is totally normal and possible, as there the node has 2 CPU, so the whole time would be 2 seconds.

The memory fooprint is 108 MB, but it was collected at the end of the experiment when no gadget were working.

2 Conclusion

To conclude, Inspektor Gadget scales until 50 nodes. Its CPU usage may be high but this should be more investigated and its peak memory footprint will be collected once AKS support kernel 5.19.