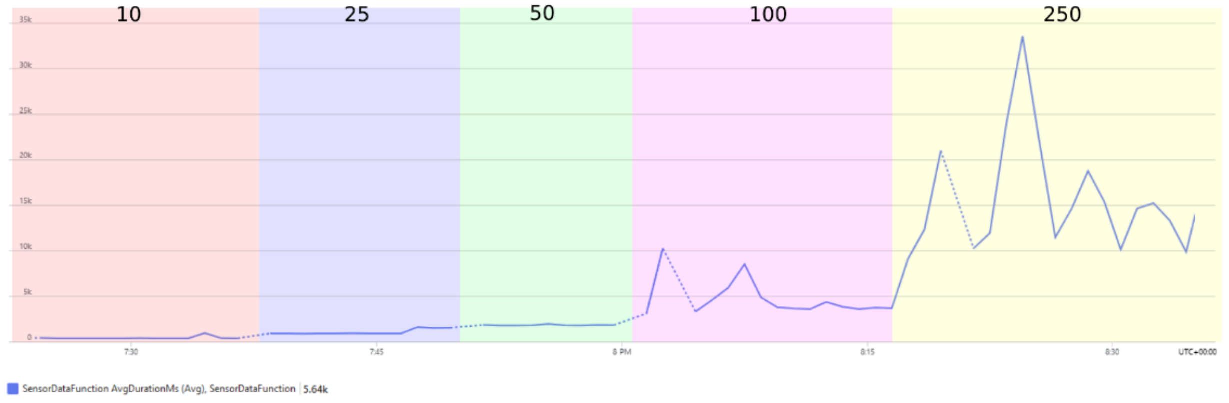
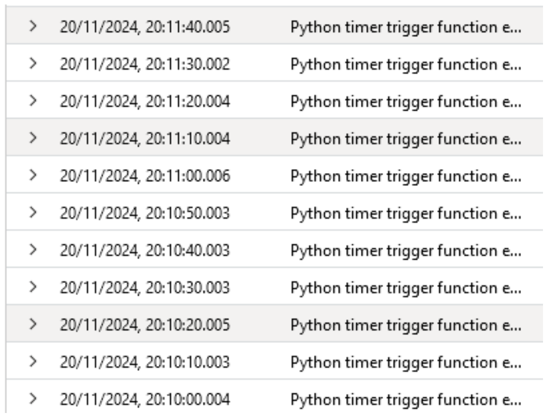
For the first function to test its scalability its implementation allows for the number of sensors simulated to be increased thus increasing the number of function invocations. So I made the function run on a timer interval of 10 seconds, slowly increasing the sensor number every 10 minutes or so. Most metrics remained stable between 10 to 50 sensors but when reaching 100+ this is where instability with the scaling occurred. The results for key metrics will be presented in graphs in which I've annotated the point around where the sensor number changed, some erroneous data occurs when the sensor number changes due to the Azure function needing to restart but it doesn't affect the overall trend.



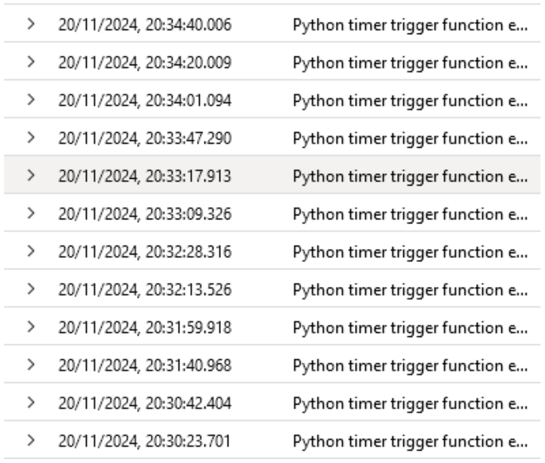
The graph above shows the average runtime of the function for different test cases. Below 50 sensors, the average runtime remains stable, demonstrating that the function handles this workload efficiently. When the sensor number doubles (e.g., from 25 to 50), the average runtime also doubles, as expected due to the increased number of database writes.

At 100 sensors, there is a more significant and less consistent increase in runtime. This suggests the system is beginning to struggle with the additional load, likely due to database write delays. Logs confirm that the timer trigger still fires consistently every 10 seconds at this stage, indicating that database write delays are the bottleneck rather than the function runtime.

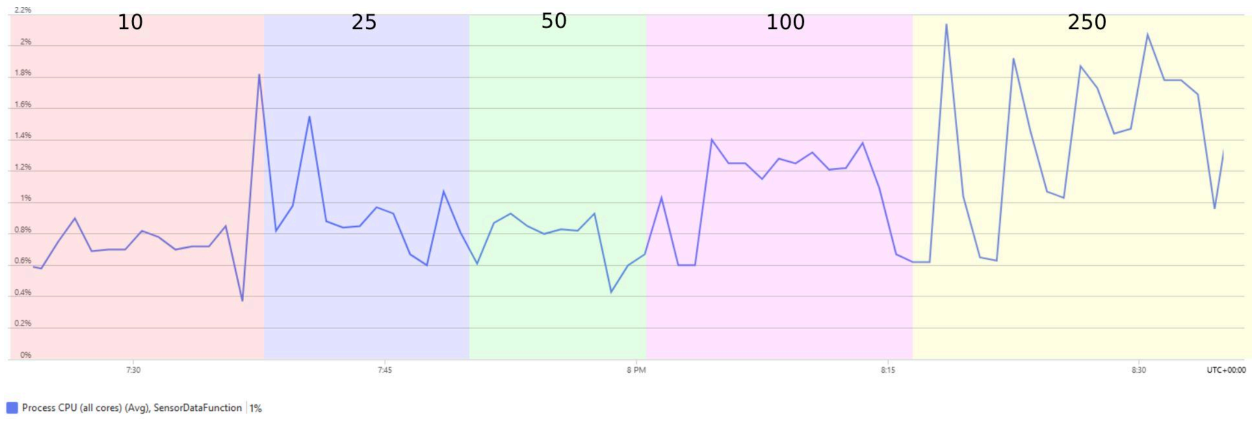


```
> 20/11/2024, 20:11:40.005    Python timer trigger function e...
> 20/11/2024, 20:11:30.002    Python timer trigger function e...
> 20/11/2024, 20:11:20.004    Python timer trigger function e...
> 20/11/2024, 20:11:10.004    Python timer trigger function e...
> 20/11/2024, 20:11:00.006    Python timer trigger function e...
> 20/11/2024, 20:10:50.003    Python timer trigger function e...
> 20/11/2024, 20:10:40.003    Python timer trigger function e...
> 20/11/2024, 20:10:30.003    Python timer trigger function e...
> 20/11/2024, 20:10:20.005    Python timer trigger function e...
> 20/11/2024, 20:10:10.003    Python timer trigger function e...
> 20/11/2024, 20:10:00.004    Python timer trigger function e...
```

At 250 sensors, the average runtime becomes both greater and more inconsistent. Additionally, the timer trigger itself becomes irregular, which suggests the function execution time has exceeded the timer interval. This behaviour supports that both the database's write delays and the function runtime contribute to the instability at higher sensor counts.

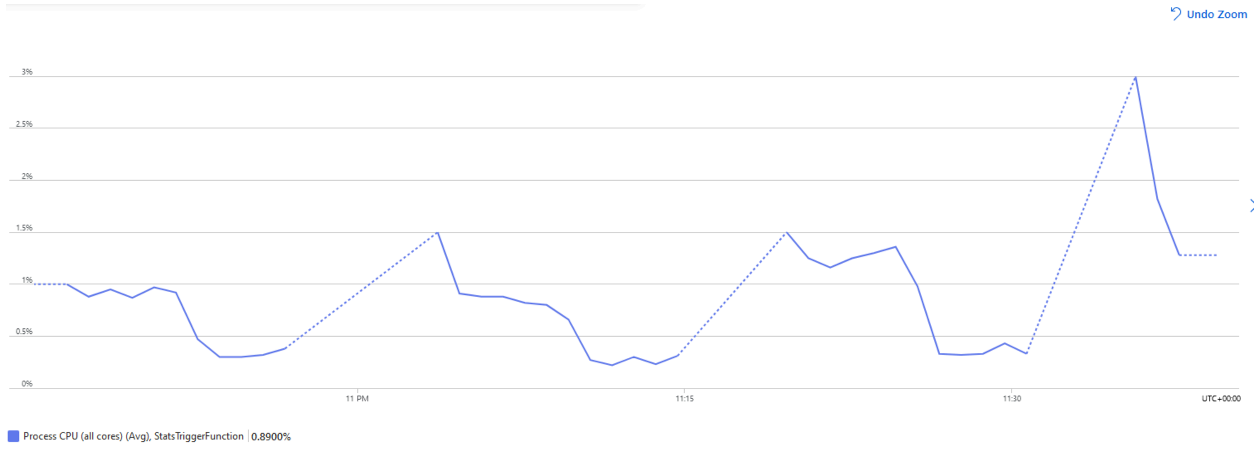


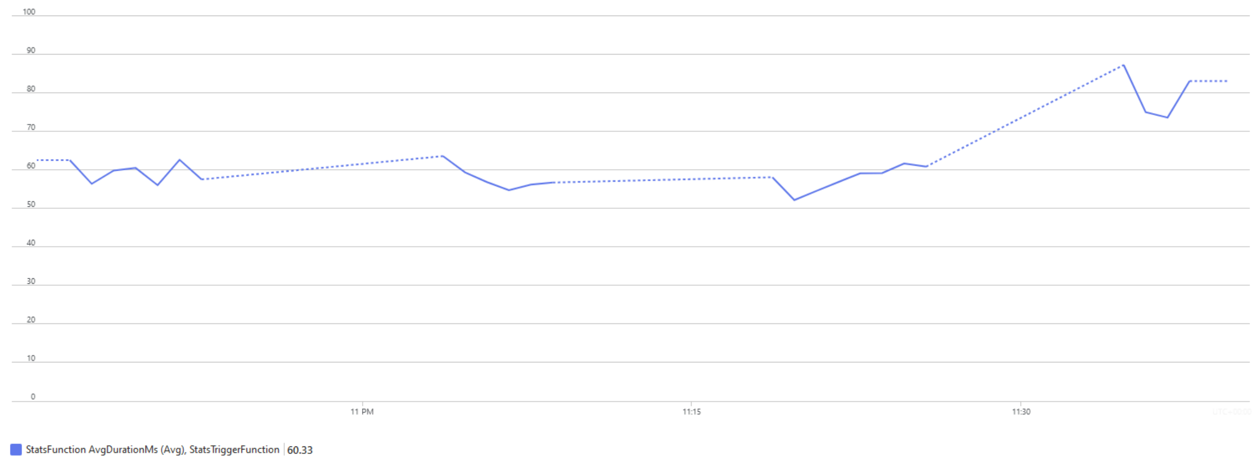
20/11/2024, 20:34:40.006	Python timer trigger function e...
20/11/2024, 20:34:20.009	Python timer trigger function e...
20/11/2024, 20:34:01.094	Python timer trigger function e...
20/11/2024, 20:33:47.290	Python timer trigger function e...
20/11/2024, 20:33:17.913	Python timer trigger function e...
20/11/2024, 20:33:09.326	Python timer trigger function e...
20/11/2024, 20:32:28.316	Python timer trigger function e...
20/11/2024, 20:32:13.526	Python timer trigger function e...
20/11/2024, 20:31:59.918	Python timer trigger function e...
20/11/2024, 20:31:40.968	Python timer trigger function e...
20/11/2024, 20:30:42.404	Python timer trigger function e...
20/11/2024, 20:30:23.701	Python timer trigger function e...



The above graph shows the CPU utilisation throughout the different test cases, as the sensor number increases there is a slight increase in CPU utilisation. As the CPU usage for the function was consistently low throughout all test cases, remaining at approximately 0.5–2%, it indicates that the computational workload of the function itself is minimal. The low CPU usage demonstrates that the function's runtime is not limited by processing capacity but rather by external factors such as database operations. Additionally, the stability of CPU usage across different sensor counts suggests that the function can handle increased invocation rates without significant strain on computational resources.

For the second function, the function invocations can be increased as the number of sensors in the first function is increased, if more changes occur then the second function will run more. Shown below are the graphs for the average runtime and the CPU utilisation done for 10, 25, 100 and 250 sensors. Across all the sensors. numbers the values remain consistent with no significant trend of increase, only a jump being seen at the 250 sensor mark where the first function has bottlenecks. This makes sense because the function is most of the time updating data to a database, and these updates don’t happen every change but in batches of 100 due to the SQL triggers attributes. So this function can scale by tweaking this batch number if the first function can handle a greater load.


Undo Zoom
11 PM
11:15
11:30
UTC+00:00
Process CPU (all cores) (Avg), StatsTriggerFunction 0.8900%


11 PM
11:15
11:30
StatsFunction AvgDurationMs (Avg), StatsTriggerFunction 60.33