Resource-Efficient Performance Monitoring on Sage Continuum Edge Devices

Cameron Durbin

December 6, 2024

1 Introduction

Geohazards such as earthquakes, wildfires, and floods pose significant risks to critical infrastructure, the economy, and human safety. Monitoring these hazards allows for timely prediction and preparation, reducing potential disaster impact. Sage Continuum, a cyberinfrastructure project funded by the National Science Foundation, focuses on geohazard monitoring through geographically distributed edge devices. This paper zeroes in on the ARM64-based NVIDIA Xavier NX devices within this infrastructure, which play a crucial role in collecting, processing, and transmitting sensor data like images and LiDAR scans.

To facilitate application scheduling and deployment, Sage Continuum employs the Waggle Edge Stack (WES) through a lightweight containerization platform based on Kubernetes. A key challenge is efficiently monitoring system performance on these resource-constrained edge devices without compromising real-time responsiveness. Traditional methods are often too heavy for constrained environments; thus, this study seeks a lightweight solution to monitor system health while operating under the limited resources of the ARM64-based edge devices.

Linux kernel's virtual filesystems, procfs and sysfs, provide an effective method for capturing system performance. These filesystems expose real-time metrics such as CPU usage, memory availability, and power consumption, making them well-suited for lightweight monitoring. This paper explores the tradeoff between performance and accuracy on the edge devices.

2 Background

2.1 Edge Computing

Edge computing emerged in response to the rapid growth of mobile applications and cloud services. Edge devices such as smartphones, wearables, and industrial IoT sensors, process data closer to its source, reducing the computational strain on centralized cloud servers. Real-time applications benefit from this architecture, as do use cases that require low latency and localized data processing [1].

Sage Continuum uses edge computing in sensor-equipped nodes. Such sensors include air quality detectors, weather systems, LoRaWAN gateways, and thermal cameras. For Sage Continuum, edge computing minimizes bandwidth usage and enables timely hazard predictions, transmitting only critical insights to its central system. [2].

2.2 System Monitoring Using Virtual Filesystem

The Linux kernel's virtual filesystems, procfs and sysfs, provide an interface to monitor system behavior without requiring additional modules or tools.

procfs

Procfs, mounted at `/proc`, offers insights into low level processes and the operating system. Each subdirectory corresponds to a running process, with numerical IDs representing individual processes. Real-time information about CPU time and interrupts, context switches, and other metrics are exposed through this filesystem [3].

This study focuses on two key files in procfs:

- `/proc/stat`: This virtual file provides aggregate CPU statistics across all cores as well as individual core data. It reports runtime in jiffies (1/100th of a second) making it well-suited for monitoring CPU usage.
- `/proc/meminfo`: This virtual file provides detailed information about the system's memory usage, including metrics such as total and available memory, free memory, cached data, and swap usage. It is a key resource for monitoring and analyzing memory performance and availability.

sysfs

Sysfs exposes hardware devices and their attributes[4], making it ideal for monitoring power consumption. It is mounted at `/sys` and offers real-time insight into the power usage of various components through the Inter-Integrated Circuit (I2C) bus[5]. This study focuses on power management data exposed by `/sys/bus/i2c/`, particularly the VDD_IN voltage and current attributes for detailed energy consumption analysis.

2.3 Tradeoff Between Performance and Accuracy

System health monitoring inherently consumes device resources, which can be a significant challenge in resource-constrained environments. Lightweight monitoring addresses this issue by using periodic sampling to collect performance data.

The choice of sampling interval is critical, as it directly impacts the tradeoff between resource overhead and monitoring accuracy. Smaller sampling intervals improve accuracy but increase system overhead, while larger sampling intervals conserve resources at the expense of accuracy.

2.4 Sage Continuum Plugins

Sage Continuum plugins are user-developed applications that are deployed for data collection and analysis on edge devices. These plugins run within the WES runtime, allowing for easy deployment and operation across distributed nodes. Each plugin is packaged as a lightweight Docker container, ensuring portability and consistent performance on different hardware platforms. This architecture enables efficient local processing of data from various sensors, cutting down on bandwidth use and reducing latency.

The plugins are designed to handle a range of tasks, from real-time data collection to running machine learning models. For instance, applications built with frameworks like TensorFlow or PyTorch can be seamlessly incorporated into the workflow. After processing data locally, the plugins send only key insights to a central system, where tools perform aggregation and advanced analysis. This approach improves scalability while reducing the resource demands of centralized processing.

3 Performance Monitoring Plugin

To capture performance data, a dedicated plugin is introduced to run on Sage Continuum's edge devices. The plugin captures CPU, memory, and power metrics and transmits them to its central server for further analysis.

3.1 Plugin Design and Functionality

The plugin is implemented as a Docker container based on a PyWaggle base image. Three monitoring units are defined within the plugin:

- /proc/stat Monitoring Unit
- /proc/meminfo Monitoring Unit
- /sys/bus/i2c Monitoring Unit

Each monitoring unit reads data from its respective virtual file at regular sampling intervals. The plugin's flexibility allows for the easy addition of new monitoring units to capture other relevant system health indicators.

`/proc/stat` Monitoring Unit

The `/proc/stat` monitoring unit processes data from the virtual file `/proc/stat`, which provides detailed CPU runtime metrics for all cores collectively and each core individually. These metrics include time spent in various states, such as user, nice, system, idle, I/O wait, IRQ, soft IRQ, and steal.

To compute CPU workload changes, the unit captures differences between consecutive sampling intervals. For each interval, it calculates the time spent in each state by subtracting the previous sample's values from the current one. These differences are then used to compute the percentage of total CPU time spent in each state. For example:

- Given initial values $t_1 = \{10, 5, 5\}$ (user, system, idle) and subsequent values $t_2 = \{24, 7, 9\}$, the differences $t_2 - t_1 = \{14, 2, 4\}$ represent time spent in each state during the interval.
- The total time is $14 + 2 + 4 = 20$ jiffies, yielding percentages of 70% (user), 10% (system), and 20% (idle).

The `/proc/stat` monitoring unit supports filtering metrics based on custom configurations, allowing a tailored view of CPU usage. For this experiment, only total CPU user time is considered.

/proc/meminfo Monitoring Unit

The `/proc/meminfo` monitoring unit processes data from the virtual file `/proc/meminfo`, which provides detailed information about the system's memory usage. Key metrics include total memory, free memory, buffers, cached memory, and swap usage.

This unit calculates memory utilization by subtracting free memory, M_F, from the total memory, M_T and dividing the result by the total memory, transforming it into a percentage:

$$\text{Memory Usage (\%)} = \left(\frac{M_T - M_F}{M_T}\right) \times 100$$

Memory data is processed at each sampling interval, enabling users to track changes in usage over time. The `/proc/meminfo` monitoring unit can be configured to focus on specific metrics such as memory used or swap usage. This experiment focuses exclusively on the memory usage.

/sys/bus/i2c Monitoring Unit

The `/sys/bus/i2c` monitoring unit tracks power usage by reading data from the I2C bus, yieldig three voltage rails: `VDD_IN`, `VDD_CPU_GPU_CV`, and `VDD_SOC`. These rails supply power to critical components of the system:

- `VDD_IN`: Maximum voltage intake.
- `VDD_CPU_GPU_CV`: Voltage between the CPU and GPU.
- `VDD_SOC`: Voltage for the system-on-chip (SoC).

The monitoring unit collects current and voltage readings from the virtual files in the `/sys/bus/i2c/` directory. The `/sys/bus/i2c` monitoring unit calculates the power consumption (in watts) for each rail using the electric power equation:

$$P\,(\text{Watts}) = V\,(\text{Volts}) \cdot I\,(\text{Amperes})$$

This data provides insights into the system's energy usage and the number of joules consumed over time. This experiment focuses exclusively on `VDD_IN`.

3.2 Integration with Waggle Edge Stack

The performance monitoring plugin runs as a container within the WES runtime environment. It operates alongside other WES-managed plugins, using the WES container orchestration system for efficient scheduling and execution.

Packaged as a lightweight Docker container, the plugin adheres to WES's modular deployment framework, leveraging containerized environments to maintain portability and consistency across various edge nodes. This approach eliminates the need for hardware-specific adjustments.

The collected metrics are published through WES-integrated messaging systems, ensuring resource usage data is accessible to other plugins. WES utilizes this data to enhance plugin scheduling and optimize workload distribution across the edge device, improving overall system efficiency.

4 Applications for Performance Testing

In this experiment, two plugins, a motion detector and an image captioner, were used for performance testing due to their high resource demands and real-time processing needs. Each trial lasts two hours, with the performance monitoring plugin continuously active. During this time, the motion detector runs every 10 minutes, and the image captioner runs every 15 minutes, resulting in deliberate overlaps every 30 minutes.

4.1 Motion Detector

The motion detector plugin [6] is a versatile tool designed for detecting motion patterns using various object detection and tracking methods. The primary detection method, Farnebäck's Dense Optical Flow Method [7], leverages polynomial expansion to estimate motion between frames, making it ideal for periodic processing of camera feeds. Additional methods include naive and advanced background subtraction techniques [8], offering flexibility for diverse use cases.

Relying on the OpenCV and NumPy libraries, the plugin ensures a lightweight and adaptable implementation. Running every 10 minutes, it generates a CPU- and memory-intensive workload, making it well-suited for evaluating periodic performance in real-time scenarios.

4.2 Image Captioner

The image captioner plugin [9] utilizes Microsoft's Florence-2-base model [10] to generate comprehensive descriptions of images captured by the device's camera. It performs inference locally using the CPU and GPU, avoiding reliance on cloud-based processing.

For each image, the plugin executes the model three times with distinct prompts:

- **Detailed captioning**: Produces a descriptive summary of the entire image.
- **Phrase grounding**: Identifies key elements in the image and pinpoints their locations.
- **Region-specific captioning**: Captures fine details from specific regions, emphasizing subtle or overlooked features.

The outputs from these runs are merged and refined through post-processing to remove duplicate and unnecessary metadata. This process delivers a detailed summary and labeled components, creating a holistic description for each image.

Running every 15 minutes, the plugin generates a substantial workload by combining computer vision and large language model processing, making it an excellent test case for assessing system performance under resource-intensive conditions.

5 Performance Analysis

Millions of data points were gathered at various sampling intervals to measure CPU usage, memory utilization, and energy consumption. The motion detector and image captioner applications are run concurrently at fixed intervals to simulate system load and assess behavior under stress.

Trials last for two hours and are configured with sampling intervals of 0.015625, 0.03125, 0.0625, 0.125, 0.25, 0.33, 0.66, 0.75, 1.0, 1.5, 2.0, and 3.0 seconds

5.1 Impact of Sampling Interval on Performance

5.1.1 CPU Usage

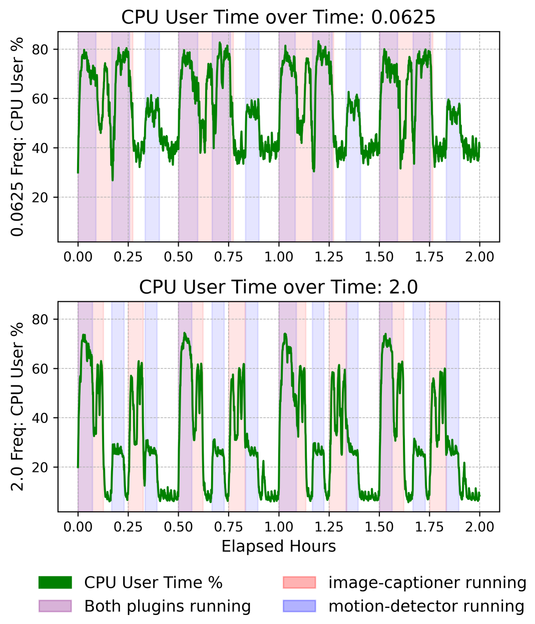

Figure 1: *Comparison of CPU user % time between sampling intervals of 0.0625 and 2.0 seconds*

Smaller sampling intervals increase the frequency of monitoring tasks, leading to overlapping execution with application workloads and other system processes, which raises CPU utilization as resources are shared among competing tasks. This overlap results in more context switches, causing delays for time-sensitive applications, reduced throughput, and overall system inefficiency.

Figure 1 highlights this impact by comparing a 0.0625-second interval to a 2.0-second interval, showing how faster sampling exacerbates task overlapping, reduces CPU availability, and degrades performance, underscoring the tradeoff between sampling frequency and system efficiency.

Across all trials, the CPU operates in distinct phases. These phases can be captured and analyzed by applying a Gaussian Mixture Model (GMM) on the CPU usage data, revealing unique clusters. The clusters are insightful for understanding how sampling intervals impact CPU usage patterns across varying workload intensities. By capturing distinct low, moderate, high, and peak phases, the analysis highlights the tradeoff between sampling granularity and phase clarity. The clustering process reveals four key phases of the trials:

Cluster 1: Idle state when neither plugin is running, reflecting baseline CPU usage.

Cluster 2: Motion detector runtime only, indicating moderate CPU demand.

Cluster 3: Image captioner runtime only, reflecting higher computational load.

Cluster 4: Concurrent runtime of both plugins, representing peak CPU usage.

The histogram comparison in Figure 2 illustrates how CPU usage phases differ between the 0.0625-second interval and 2.0-second interval. At the 0.0625-second interval, the clusters are distinct but exhibit higher variability due to the granularity of frequent sampling. Notably, only three clusters are observed because cluster 3 (image captioner runtime) and cluster 4

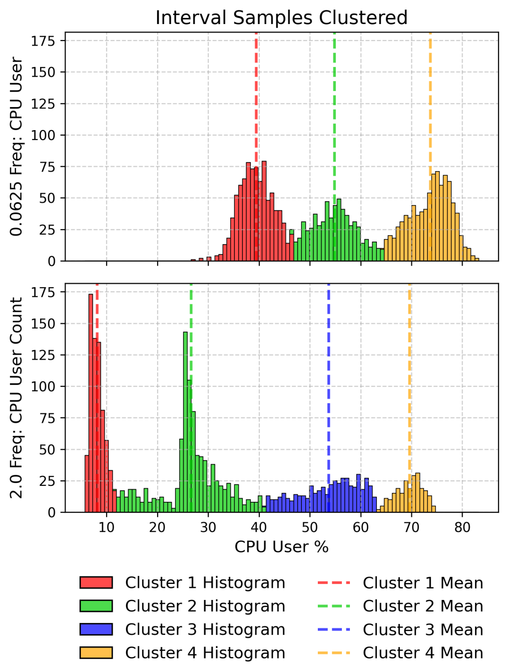

Figure 2: *Histogram comparison of CPU user % time between sampling intervals of 0.0625 and 2.0 seconds*

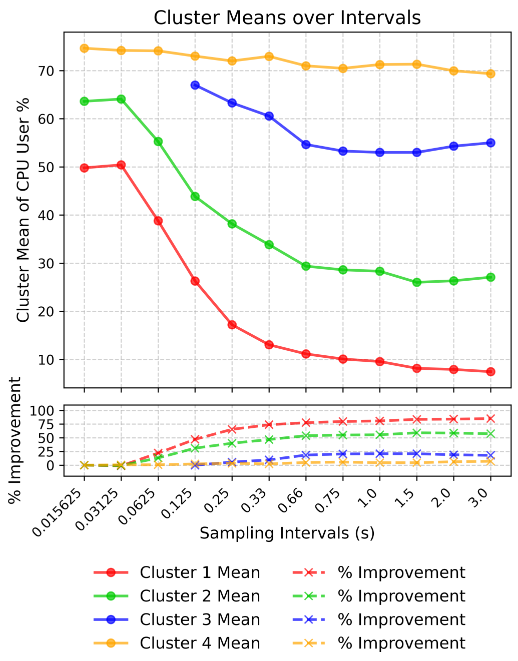

Figure 3: *Comparison of CPU user % mean time across all intervals.*

(concurrent runtime of both plugins) overlap, reflecting increased task interference.

In contrast, at the 2.0-second sampling interval, all four clusters are clearly separated. The reduced monitoring overhead allows for better phase distinction, as workload tasks have more uninterrupted CPU time. This demonstrates how smaller intervals increase overlap and cluster complexity, while larger intervals provide clearer separation, highlighting the impact of sampling frequency on task distribution and CPU utilization patterns.

Shorter sampling intervals are anticipated to yield higher accuracy in monitoring; however, the overhead introduced from monitoring complicates precise CPU usage measurements. As the sampling interval decreases, the performance monitoring plugin's overhead grows, causing interference that distorts the true CPU usage patterns of the test application workload. This interference reduces the reliability of performance data and highlights the tradeoff between monitoring granularity and the effectiveness of system performance analysis.

Figure 3 compares the mean CPU user times across all clusters for each trial. Cluster 1 shows the most significant improvement, with CPU usage increasing by approximately 80% as the sampling interval grows from 0.015625 seconds to 3.0 seconds. Cluster 2 experiences a 55% increase, while clusters 3 and 4 display more modest gains of 20% and 8%, respectively. Notably, at the 0.125-second sampling interval, the means of clusters 3 (image captioner runtime) and 4 (concurrent runtime) begin to differentiate clearly, highlighting the impact of reduced monitoring interference on phase separation.

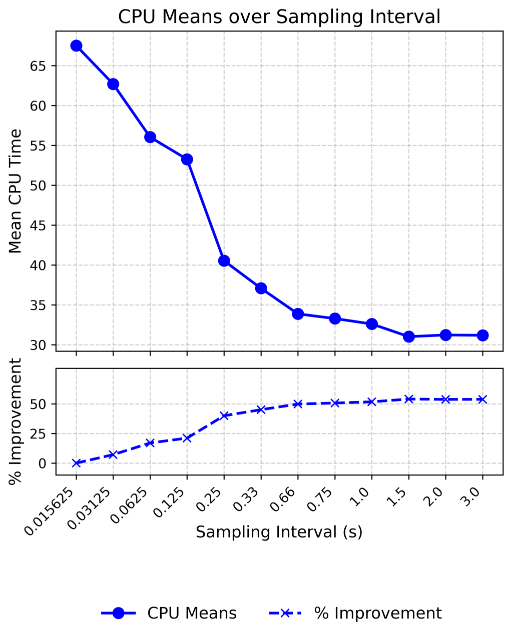

Figure 4: *Comparison of CPU user % mean time across all intervals.*

Figure 4 provides a holistic view of the trials by plotting the mean CPU user time for each sampling interval. The graph highlights the significant impact of sampling interval on CPU performance, offering valuable insights into the tradeoff between performance and accuracy. Shorter intervals produce more data points, enabling finer-grained monitoring but at the cost of higher overhead. In contrast, greater sampling intervals produce fewer data points, reducing overhead and interference, but potentially sacrificing detailed insights into CPU usage patterns.

5.1.2 Memory Usage

Memory usage analysis follows a similar methodology to CPU usage, examining the effects of different sampling intervals on observed utilization patterns. Figure 5 compares memory usage percentages at the 0.0625-second interval and 2.0-second interval, highlighting how utilization varies across application states.

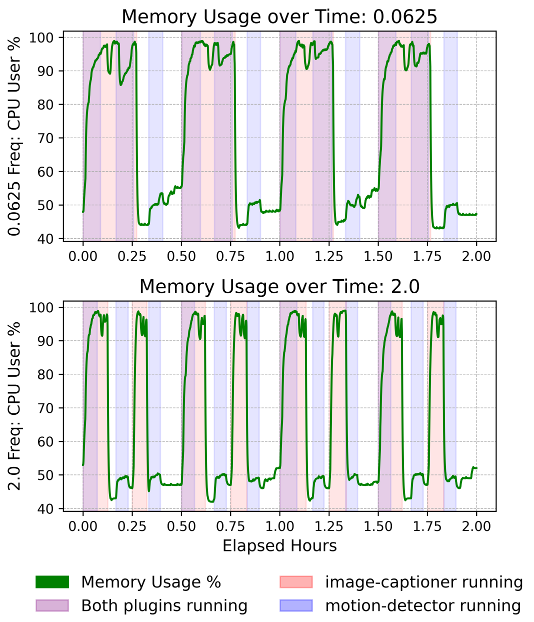

Figure 5: *Comparison of memory usage % between scheduled applications at sampling intervals of 0.0625 and 2.0 seconds.*

The analysis shows that memory utilization remains relatively stable across sampling intervals. As illustrated in Figure 5, memory usage fluctuates between 40% and 50% when neither plugin is active. During the concurrent execution of both plugins, memory usage spikes to 90%–100%, reflecting near-total utilization of the device's memory capacity. This behavior brings out the significant memory demands of running resource-intensive plugins simultaneously.

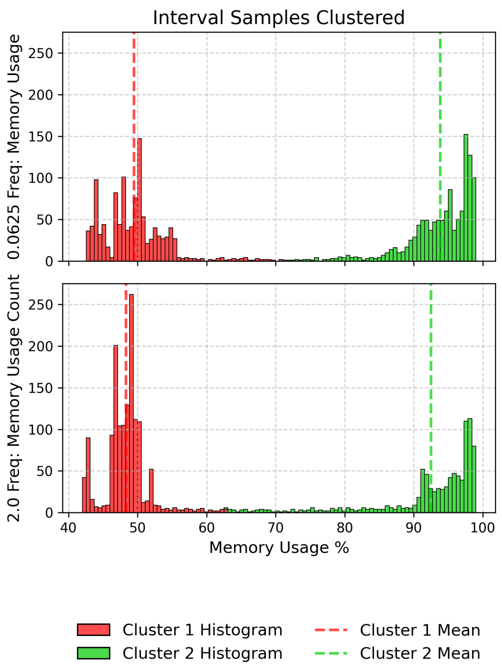

Figure 6: *Clustered histogram comparison of memory usage % at sampling intervals of 0.0625 and 2.0 seconds.*

Figure 6 shows clustered histograms that indicate similar memory utilization means between the two sampling intervals. However, at the 2.0-second interval, there is less time spent at larger memory capacities compared to the 0.0625-second interval. Despite this variation, the overall distributions remain largely consistent, showing minimal impact across sampling intervals.

The memory usage remains consistent across all sampling intervals, with significant increases observed only during the simultaneous execution of both plugins. These findings indicate that memory utilization is less sensitive to changes in sampling interval size compared to CPU user time, highlighting its stability under varying monitoring conditions.

5.1.3 Energy Consumption

Energy consumption analysis examines the impact of different sampling intervals on power usage, adopting a methodology similar to CPU and memory analysis. Figure 7 compares energy usage (in Watts) at the 0.0625-second interval and 2.0-second interval, revealing distinct patterns in power consumption.

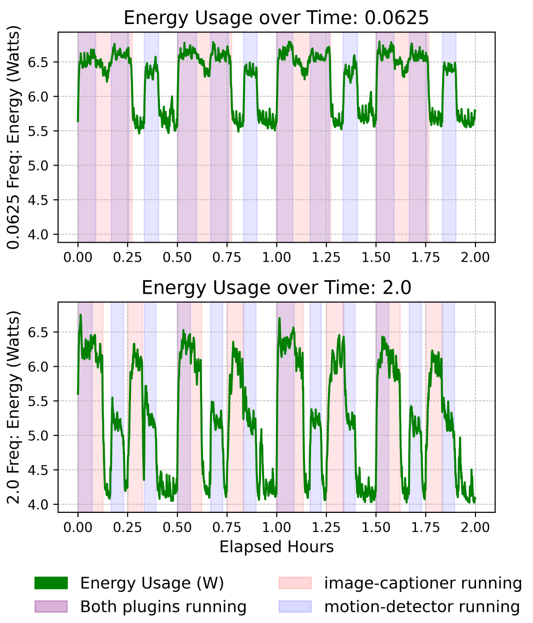

Figure 7: *Comparison of energy usage (Watts) between scheduled applications at sampling intervals of 0.0625 and 2.0 seconds.*

The data shows considerable variation in energy consumption between intervals. At the 0.0625-second interval, energy usage ranges from approximately 5.5W to 6.7W. In contrast, the 2.0-second interval results in a range of 4.1W to 6.5W. Less frequent monitoring lowers baseline energy requirements, as reflected in the reduced minimum power usage.

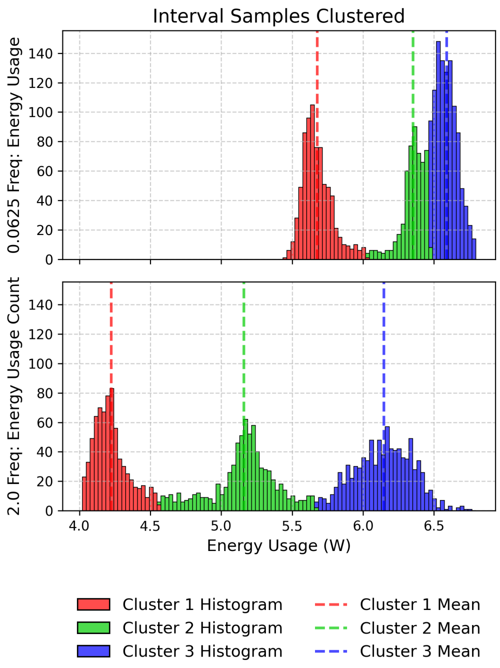

Figure 8: *Clustered histogram comparison of energy usage (Watts) at sampling intervals of 0.0625 and 2.0 seconds.*

Figure 8 presents the clustered histograms for energy usage at the 0.0625-second interval and 2.0-second interval, categorized into three clusters, unlike the four clusters observed in CPU user time analysis. The reduction to three clusters is attributed to a hardware-imposed limit on power draw during the simultaneous execution of both plugins, preventing overheating or exceeding the device's energy budget. This limitation causes overlapping energy usage patterns for the image captioner and the combined execution of both plugins, effectively merging them into a single cluster.

The three phases in energy clustering for this experiment are as follows:

Cluster 1: Idle state when neither plugin is running.

Cluster 2: Runtime of the motion detector only.

Cluster 3: Runtime of the image captioner only and the combined runtime of both plugins.

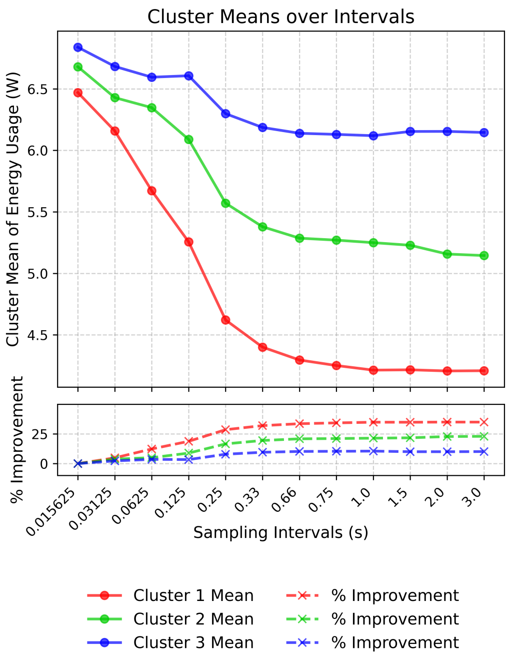

Figure 9: *Comparison of clustered means of energy usage (Watts) across all sampling intervals.*

Figure 9 compares the clustered means of energy usage across all sampling intervals. At the smallest interval (0.015625 seconds), the cluster means are closely grouped, reflecting the high granularity of frequent monitoring. This granularity increases the runtime of the performance monitoring plugin, resulting in overlaps between application runtimes and blending energy usage patterns.

As the sampling interval increases, the runtime of the monitoring plugin decreases, reducing overlaps and allowing for clearer separation of application phases. Larger intervals spread the cluster means further apart, emphasizing distinct energy usage patterns as transitions between application states become less frequent and smoother.

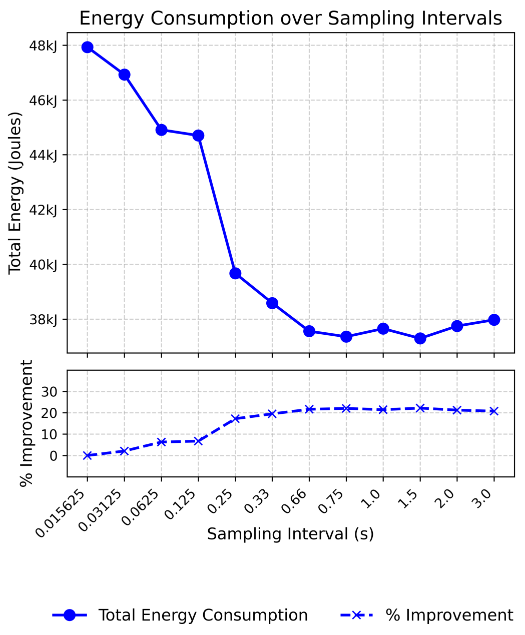

Figure 10: *Comparison of joules consumption across all sampling intervals.*

To quantify energy usage for each trial, the following function is applied:

$$E_{\text{total}} = \sum_{i=2}^{n} P(t_i) \cdot \Delta t_i$$

where $P(t_i)$ represents the effective power usage at sample time t_i, and Δt_i is the time difference between sample times t_i and t_{i-1}. This computation yields the total energy usage in joules for each sampling interval.

Figure 10 plots the total energy consumption in joules across all sampling intervals, providing a comprehensive perspective on energy performance. The results show that smaller sampling intervals significantly increase energy consumption due to higher monitoring overhead. These findings highlight the tradeoff between finer sampling for accuracy and the energy efficiency required for sustained operation in monitoring applications.

5.2 Impact of Sampling Interval on Plugin Runtime

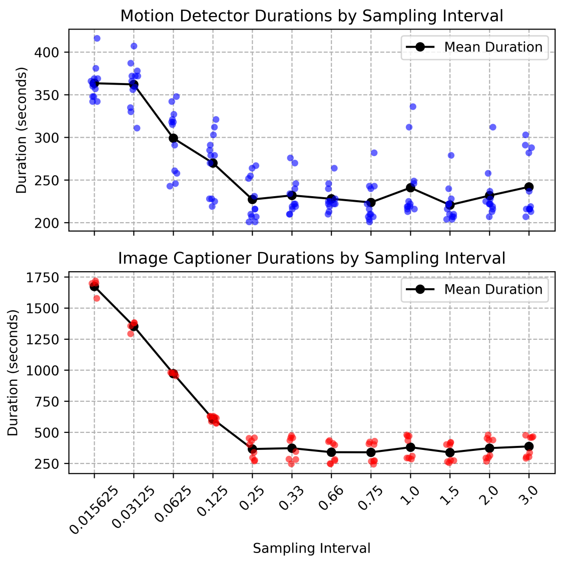

Figure 11: *Application Runtime vs Sampling Interval*

To evaluate the impact of system monitoring on application performance, we analyzed the runtime of the two testing plugins: the motion detector and the image captioner, under each sampling interval. The execution durations were recorded for each interval.

5.2.1 Performance Trends

The motion detector plugin exhibited a gradual improvement in execution time as the sampling interval increased. At very short intervals (e.g., 0.015625 seconds), the overhead introduced by frequent sampling led to execution durations averaging 350–400 seconds. As the sampling interval increased to 0.125 seconds, the mean duration dropped significantly, after which the improvement plateaued, stabilizing below 250 seconds for intervals beyond 0.125 seconds.

Conversely, the image captioner application demonstrated extreme sensitivity to short sampling intervals, with mean durations exceeding 1500 seconds at 0.015625 seconds. This suggests substantial interference caused by intensive monitoring. However, as the sampling interval increased, the execution durations dropped sharply, stabilizing around 350 seconds for intervals greater than 0.125 seconds.

5.2.2 Analysis

The observed trends highlight a tradeoff between sampling interval size and application performance. While frequent sampling provides fine-grained monitoring data, it imposes significant overhead, particularly for computationally intensive applications like the image captioner. This effect diminishes as the sampling interval increases, indicating a point of diminishing returns where the performance gains become negligible.

5.3 Performance and Accuracy Tradeoffs in Monitoring

Evaluating the tradeoff between performance and accuracy requires defining utility functions for each. By integrating these utility functions, we gain valuable insights into the balance between the two.

Since memory usage stayed consistent regardless of sampling interval, we will be analyzing the tradeoff in terms of CPU usage and energy consumption. To capture a holistic understanding of the tradeoff, we use the CPU means across all sampling intervals (Figure 4) and the joules consumed consumed across all sampling intervals (Figure 10).

5.3.1 Performance Utility Function

The performance utility function quantifies the performance of the experiment where the sampling interval is the input. Assuming that the performance is equally important as accuracy, the performance utility function needs to be measured on the same scale as the accuracy. We can achieve this by normalizing the CPU means and joules consumed using Min-Max Normalization of the vector of CPU means and joules consumed.

$$normalize(v) = \frac{v - \min(v)}{\max(v) - \min(v)}$$

The values after normalization will retain the information about the performance and map the values to the range $[0, 1]$. For both CPU means and joules consumed, we can identify lower and upper bounds.

For CPU means, we know that the CPU will never exceed 100% and will never fall below 0%. We also know that as the sampling interval decreases, the CPU means approach it's maximum use capacity. As the sampling interval increases, the CPU means approach it's minimum use capacity.

Similarly, for joules consumed, the joules will not exceed more than the power supply will allow it to pull and while the system runs, the power supply will give the system enough to power to stay on. We also know that as the sampling interval decreases, joules consumption approaches the maximum capacity at which the power supply can operate and as sampling interval increases, joules consumption approaches the minimum capacity at which the power supply will give for the system still stay up and running.

We can fit a function to both the CPU means and energy consumption to represent the performance utility function.

$$U_{\text{perf}}(x) = 1 - \frac{L}{1 + e^{-k(x-x_0)}} + b$$

We fit this curve to our CPU means and energy consumption because the data is representitive of the complement of a logistic function. Fitting the curve to the normalized CPU means and joules consumption data, we have an R^2 value of 0.990 and 0.985 value respectively.

5.3.2 Accuracy Utility function

The accuracy utility function quantifies the accuracy of the experiment where the sampling intervals is an input. Smaller sampling intervals yield more data points which, in turn, put together a more complete and confident understanding of the system health. We can use the number of data points as a way to measure the accuracy of the system health.

The number of data points in each trial is given as:

$$\text{num\_points}(interval) = \frac{7200}{interval}$$

where $\frac{1}{interval}$ models the number of data points collected per second and 7200 is the number of seconds in 2 hours.

We take the number of points of each trial as a vector and normalize the vector. This process retains the structure of the data while scaling the data to the range [0, 1].

To capture the growth of the normalized number of points, we fit a function to represent the accuracy utility function.

$$U_{\text{acc}}(x) = \frac{a}{x} + b$$

Fitting this curve to our normalized number of points, we fit the function with an R^2 value of 0.999.

5.3.3 CPU and Energy Tradeoff

Combining the performance utility functions fit on the normalized CPU means and the normalized joule consumption with the accuracy utility function fit to the normalized sampling interval data. We create two combined utility functions that assist in understanding the tradeoff.

Figure 12 illustrates the performance utility function, accuracy utility function, and combined utility function for both CPU

Utility Function	Local Minimum	Local Maximum	Midpoint
CPU Means	0.063	1.113	0.588
Energy Consumption	0.072	0.616	0.344

Table 1: *Measured values for CPU Means and Energy Consumption.*

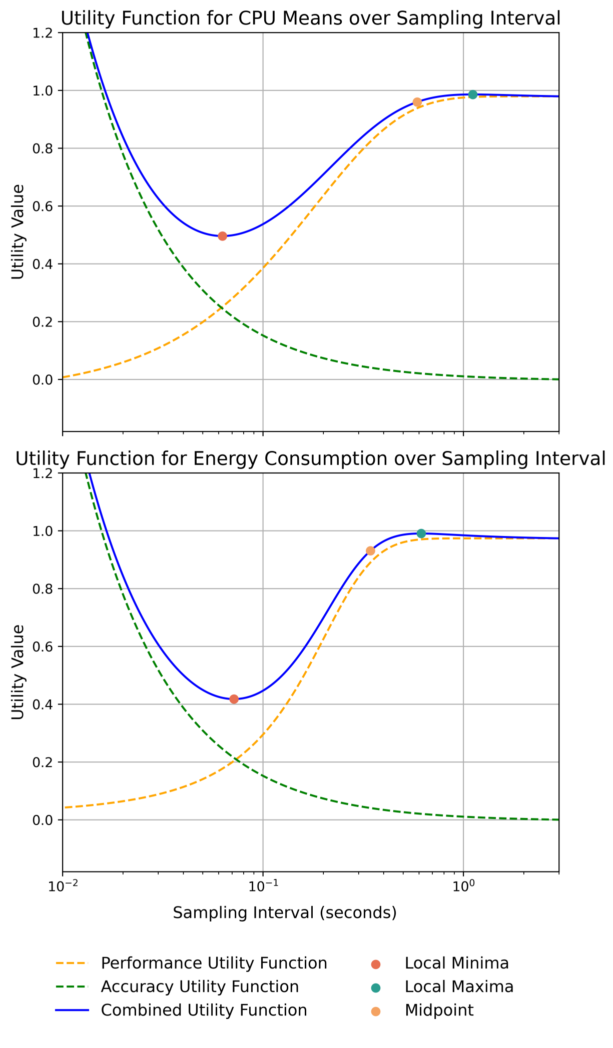

Figure 12: *Utility functions for CPU Means and Joules Consumption*

means and energy consumption. The combined utility function highlights two key points:

1. **Local Minimum:** Represents a suboptimal choice, as it sacrifices significant performance to achieve a higher rate of data collection. Sampling intervals smaller than this point result in diminishing returns for performance.

2. **Local Maximum:** Marks the optimal point, where performance peaks, offering the highest possible data collection rate at peak performance. This is the point of maximum efficiency for both performance and data collection, making it the most desirable option in many scenarios.

The significance of the local minimum lies in its role as a reference point for determining a balanced sampling strategy. By taking the midpoint between the local minimum and the local maximum, the system achieves a strategy that sacrifices some performance for increased accuracy in a controlled and measurable way. This range between the local maximum and the midpoint is particularly advantageous, as it balances key priorities:

- **Fair Trade-Off:** This range provides a balanced compromise, allowing sufficient data collection while maintaining acceptable performance levels. It avoids the extremes of overburdened performance or insufficient data.

- **Performance Efficiency:** Sampling closer to the midpoint remains efficient, avoiding the steep performance cost of the local minimum while still collecting meaningful data.

- **Applicability:** The strategy is practical for constrained environments, such as embedded systems or energy-sensitive platforms, where moderate monitoring overhead is necessary.

- **Strategic Accuracy:** While not as data-rich as the local minimum, this range still captures trends effectively, enabling informed decision-making and adaptive application adjustments.

The midpoint between the local maximum and local minimum offers a balanced strategy, with a range near the local maximum leaning toward performance efficiency while maintaining meaningful data collection. This approach is well-suited for dynamic or resource-constrained systems.

Table 1 gives the sampling interval values for the points on figure 12. The CPU means and energy consumption utility functions have different midpoints because the CPU and energy consumption react differently towards a sampling intervals.

6 Resource Planning and Scheduling Optimization

Science goals, defined as rule-sets dictating how and when plugins execute on edge nodes, are created by scientists to achieve specific objectives. These goals are managed within the Sage Continuum scheduler, which is responsible for deploying and executing plugins according to the defined rules.

The performance monitoring plugin captures critical resource utilization data, such as CPU usage, memory consumption, and energy metrics. By optimizing the balance between accuracy and performance, the plugin determines an efficient sampling interval that ensures high-quality metrics are published without overloading the system. These metrics are then utilized by the scheduler to make intelligent decisions about plugin execution and resource allocation.

The scheduler leverages this data to dynamically adjust plugin schedules and maintain efficient node operations. For example, when resource utilization approaches critical thresholds, the scheduler could refine execution patterns or adjust task priorities to sustain balanced operations. This ensures that edge nodes operate within optimal limits while achieving the defined scientific objectives.

By combining accurate resource monitoring with adaptive scheduling, the Sage Continuum scheduler ensures efficient resource utilization, sustained reliability, and the successful execution of science goals, all while adapting to real-time workload and system conditions.

7 Conclusion

The research presented here offers a lightweight method to monitor system health on Sage Continuum edge devices. The dedicated plugin captures CPU, memory and power metrics without introducing excessive overhead, enabling efficient performance tracking even in resource-constrained environments.

This study explores the tradeoff between sampling frequency and accuracy in system monitoring, revealing its impact on device behavior and resource usage. It highlights that while finer sampling provides more detailed insights, it comes at the expense of increased computational load, which can hinder real-time responsiveness. The analysis presents a approach to this tradeoff, offering practical guidance for selecting sampling intervals based on the application's needs.

The lightweight monitoring method enables efficient system health tracking without compromising device performance. This capability is particularly valuable for resource-constrained environments like Sage Continuum's edge devices where timely hazard detection relies on real-time sensor data processing.

The insights gained from this research have implications for enhancing the efficiency of distributed monitoring systems like Sage Continuum. They allow developers to strike a balance between accuracy and performance, optimizing resource usage while maintaining system health insights. This approach enables more efficient application scheduling and improved workload management on edge devices.

References

[1] Mahadev Satyanarayanan. "The Emergence of Edge Computing". In: *Computer* 50.1 (2017), pp. 30–39. DOI: 10.1109/MC.2017.9.

[2] SAGE Continuum. *SAGE Continuum Project: An Open Instrumentation and Edge Computing Platform*. https://sagecontinuum.org/about. Accessed: 2024-11-21.

[3] The Linux Kernel Documentation. *Virtual Filesystem (VFS)*. https://docs.kernel.org/filesystems/vfs.html. Accessed: 2024-11-21.

[4] Patrick Mochel. "The sysfs filesystem". In: *Linux Symposium*. Vol. 1. The Linux Foundation San Francisco, CA, USA. 2005, pp. 313–326.

[5] The Linux Kernel Documentation. *I2C Sysfs Interface*. https://docs.kernel.org/i2c/i2c-sysfs.html. Accessed: 2024-11-21.

[6] Colin Burdine, Sean Shahkarami, and Nicola Ferrier. *Motion-detector*. 2024. URL: https://portal.sagecontinuum.org/apps/app/seonghapark/motion-detector.

[7] Gunnar Farnebäck. "Two-Frame Motion Estimation Based on Polynomial Expansion". In: vol. 2749. June 2003, pp. 363–370. ISBN: 978-3-540-40601-3. DOI: 10.1007/3-540-45103-X_50.

[8] Zoran Zivkovic and Ferdinand van der Heijden. "Efficient adaptive density estimation per image pixel for the task of background subtraction". In: *Pattern Recognition Letters* 27.7 (2006), pp. 773–780. ISSN: 0167-8655.

DOI: https://doi.org/10.1016/j.patrec.2005.11.005. URL: https://www.sciencedirect.com/science/article/pii/S0167865505003521.

[9] Ryan Rearden, Seongha Park, and Yongho Kim. *Plugin-image-captioning*. 2024. URL: https://portal.sagecontinuum.org/apps/app/yonghokim/plugin-image-captioning.

[10] Bin Xiao et al. *Florence-2: Advancing a Unified Representation for a Variety of Vision Tasks*. 2023. arXiv: 2311.06242 [cs.CV]. URL: https://arxiv.org/abs/2311.06242.