# An Autonomous Task-Clustering Manager for Edge Computing Applications Running on Raspberry Pi

Emad Albassam [1]

[1] *Department of Computer Science, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah, Saudi Arabia*

***Abstract*** **– Edge nodes in Edge computing systems are often deployed in close proximity to IoT devices for local data processing and storage. Recently, it has been shown that low-end single-board computers, such as Raspberry Pis, can be utilized as Edge nodes. However, these devices are limited in terms of computational capabilities, especially in cases where they need to perform a large number of concurrent tasks in response to connected IoT devices. Furthermore, these edge nodes may be deployed in unstable or hostile environments with fluctuating power sources, affecting their performance. This paper investigates how task clustering techniques can achieve different configurations of task clusters running on a Raspberry Pi as an Edge node. First, an empirical evaluation is conducted to examine how various task-cluster configurations affect the performance of applications running on these devices. Experimental results show that up to 25% improvement in execution time is obtained using a configuration with an appropriate task clustering configuration compared to a configuration in which each object is scheduled as a separate task. Then, this study describes the design of an autonomous Task-Clustering manager extension based on the MAPE-K feedback loop that runs as a layer on top of existing job schedulers to automatically configure task clusters running on these devices to improve performance.**

***Keywords*** **– Edge computing, IoT, MAPE-K, Task clustering, Raspberry Pi.**

## 1. Introduction

Due to their low costs, low-end single-chip computers such as Raspberry Pis and Arduinos have become attractive for running IoT applications. Recently, it has been shown that such devices can also serve as Edge nodes for providing services to IoT devices [1]. However, one challenge in incorporating these devices on the Edge is their limited computational power in running a significant number of concurrent activities, particularly as more IoT devices are connected to these Edge nodes. Furthermore, it has been shown that conventional task scheduling schemes do not adequately tackle the constraints of applications running on such devices [2]. Therefore, it is essential to carefully design concurrent tasks of an Edge computing application running on these devices to improve their performance.

The Edge Computing architecture (Figure 1) consists of three layers: the cloud layer, the edge layer, and the device layer [3], [4]. The different IoT devices, such as sensors, controllers, and actuators, are located at the Device Layer. The Edge layer consists of computational nodes deployed near IoT devices in the Device layer, where the goal is to provide efficient and fast responses to requests from the Device layer. Services offered by the Edge layer can include real-time data processing, data reduction, and caching. Finally, nodes at the Edge layer can communicate with the Cloud layer, which consists of dedicated nodes for complex data processing, aggregation, integration, and storage that may otherwise be infeasible at edge nodes.

DOI: 10.18421/TEM153-10
https://doi.org/10.18421/TEM153-10

**Corresponding author:** Emad Albassam,
*Department of Computer Science,*
*Faculty of Computing and Information Technology,*
*King Abdulaziz University, Jeddah, Saudi Arabia*
**Email:** ealbassam@kau.edu.sa

*Received: 05 August 2025.*
*Revised: 16 February 2026.*
*Accepted: 23 February 2026.*
*Published: 27 August 2026.*

This study considers the following attributes in edge computing applications: (1) Low-end single-board computers with restricted computational power, such as Raspberry Pis, are deployed to the edge layer, (2) Devices from the device layer connect to Edge nodes (e.g., Raspberry Pis) to obtain services, where the number of connected devices can be large, and therefore, the number of concurrent services executed by Edge nodes can be large as well, (3) Devices in the device layer are dynamic in that new devices can connect to these Raspberry Pis serving as edge nodes, while currently connected devices may disconnect from the Raspberry Pis, and (4) The Raspberry Pis can be underpowered due to environmental factors, which may affect their performance.

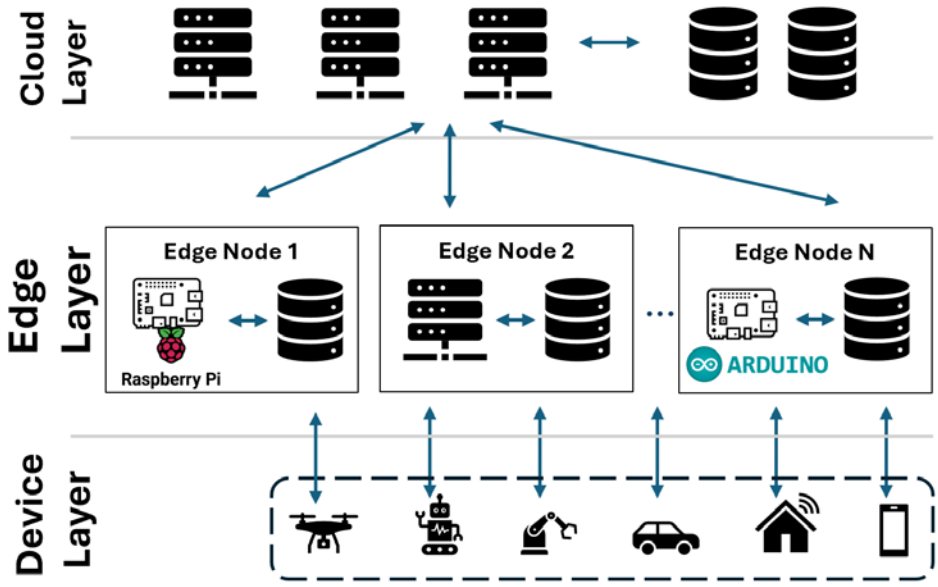


*Figure 1. An overview of the Edge computing architecture*

Based on these attributes, and since these low-end single-chip computers acting as Edge nodes in the Edge Layer may have limited computational power and can provide concurrent services to connected devices in the Device Layer, the overall system performance may degrade without careful design of the application's architecture.

One factor contributing to optimizing performance is the application's configuration from a software architectural point of view. Although it is possible to design each object of an Edge computing application as a concurrent task, such a design may result in a large number of concurrent and potentially small tasks, causing a system overhead [5]. On the other hand, designing objects to be passive (i.e., non-concurrent) is restricting, especially with the availability of multi-core processors on these devices. Therefore, there is a need for an approach that dynamically adapts the configuration of an Edge computing application in terms of task structuring to an optimal configuration based on the capabilities of these devices.

This paper contributes to the literature by discussing the design of an autonomous task clustering manager capable of automatically configuring the number of concurrent tasks deployed to low-end single-chip Edge nodes, such as Raspberry Pis, at runtime by using task clustering techniques. The manager is designed to complement existing task schedulers.

The research questions addressed by this paper are as follows:

*1) How does applying task clustering techniques affect the execution time of concurrent tasks running on low-end single-board Edge nodes such as Raspberry Pis?*

*2) How can an autonomous task clustering manager for low-end single-board Edge nodes, such as Raspberry Pis, be designed to configure the number of task clusters at runtime to improve execution time using task clustering techniques?*

The remainder of this paper is structured as follows. Section 2 discusses the clustering techniques being investigated in this work. Section 3 contains the literature review and describes how this work is related to prior works. The experimental design and procedures are presented in Section 4. The experimental results are presented in Section 5. The results of this study are discussed in Section 6. Section 7 concludes this work and outlines potential directions for future research.

## 2. Background

Several task clustering techniques have been previously proposed in the area of software architectural design, where the goal is to design a concurrent software architecture that reduces the execution overhead that can be potentially caused by running a large number of concurrent tasks. This paper considers the impact of three task clustering techniques: temporal clustering, sequential clustering, and multiple-instance task inversion clustering [5].

*Temporal clustering* can be applied to candidate tasks that are triggered by the same event (such as a timer event) by combining such tasks into a single task cluster, given that there is no sequential relationship between these tasks. For example, in Edge computing, edge nodes might periodically pull/receive data from various connected devices from the Device layer and then process this data. In this case, such data processing activities can be designed as a task cluster where they are executed in any sequential order using a single thread to reduce overhead potentially caused by allocating a different thread for each activity. Therefore, temporal clustering can be applied in situations in which multiple candidate tasks are activated periodically at the same frequency (e.g., once every 100ms) or in situations in which multiple candidate tasks are activated periodically at different frequencies such that these frequencies are multiples of one another (e.g., every 50ms and 100ms).

*Sequential clustering*, on the other hand, can be used in situations in which there is a sequential order between two or more candidate tasks. The first candidate task in the sequence is initiated periodically or aperiodically. Each candidate task in the sequence then activates the subsequent candidate task upon completion. Such sequential candidate tasks can be consolidated into a single task cluster according to the sequential task clustering technique. As example in edge computing, an Edge node may receive an event from an IoT device from the device layer, triggering a sequence of tasks to be executed by the edge node, such as data pre-processing, analysis, and post-processing.

Since these candidate tasks have sequential dependencies, it is possible to structure them into a task cluster where they are executed by a single thread, one after another, to decrease overhead compared to allocating a separate thread for each sequential task.

*Multiple-instance task inversion clustering* allows for the grouping of multiple instances of a specific type into a single task cluster, instead of allocating a different task for each of these instances. In edge computing, for instance, the edge node might keep track of the state of multiple UAVs (Unmanned Aerial Vehicles) by using a state machine. Since multiple UAVs are involved, the edge node needs multiple instances of the same state machine type. In multiple-instance task inversion clustering, it is possible to group such multiple instances into a task cluster. In this case, when the task cluster receives an event, the cluster forwards the event to the designated instance embedded within it for further processing.

The approach in this work considers the design of an autonomous task clustering manager capable of incorporating these task clustering techniques dynamically at run-time to improve the performance of Edge computing applications running on Raspberry Pi as edge nodes.

## 3. Related Works

Task clustering in Edge computing applications has been the subject of extensive recent studies tackling a variety of constraints and problems in this domain. A prior work by [6] investigated the issue of edge node mobility on unmanned aerial vehicles (UAVs) and suggested a method for task offloading in such settings. In order to optimize job allocation for Edge micro-clusters, [7] introduced a linear-based model and suggested a metaheuristic particle swarm optimization. In addition, a method for optimizing task clustering based on the degree of similarity of task features in order to determine the best execution timings for tasks that were submitted for scheduling [8]. In addition, several techniques have been proposed for clustering based on similarity measurements [9], [10]. Such clustering techniques rely extensively on similarity analysis, for example, in data-intensive applications [11], [12]. In this regard, the proposed approach in this study considers how the task clustering techniques discussed in Section 2 can be identified and applied without incurring overhead related to similarity analysis, which is more suited for Edge computing applications that operate on low-end computers, such as the Raspberry Pis.

The issue of running concurrent applications on low-end single-board computers, like the Raspberry Pi, has also been examined in earlier studies.

For instance, [13] proposes a clustering technique employing four Raspberry Pis running Apache Hadoop tools for parallel computation. In addition, [14] demonstrated how to set up a Raspberry Pi cluster with a portion of the OpenStack Swift orchestration framework. However, these approaches are considered technology-dependent. On the other hand, this study investigates a technology-independent approach for task clustering.

The effectiveness of Raspberry Pis as Edge nodes was highlighted in a number of studies. For instance, these devices were used in flood forecasting systems to manage the real-time collection and processing of lightning signals [15]. A prior evaluation of task scheduling algorithms in edge data centers consisting of Raspberry Pi edge devices show that with current scheduling algorithms and due to low power or memory, over 25% of tasks fail to run [2]. In this respect, this study tackles the challenge of improving the performance of Raspberry Pis as edge nodes by proposing an extension manager that runs on top of existing task schedulers for dynamic and improved task clustering.

In the area of job scheduling, several surveys have been conducted to classify the abundant amount of work in job schedulers, e.g., [16]. Many such approaches rely on thread pool technology that is maintained by a job manager to optimize the scheduling of submitted jobs [16], [17], [16]. The work by [18] analyzed the most widely used task scheduling techniques. Their results show that task scheduling algorithms enhance task execution performance with the use of suitable algorithms, including meta-heuristic algorithms. Prior works have also clarified key factors for choosing a scheduling algorithm by conducting a comparative analysis of current heuristic-based job scheduling methods [19]. This study complements existing job scheduling approaches as it runs as a layer on top of the job schedulers to further optimize jobs within task clusters before or after they are submitted to the job scheduler.

Compared to prior works, this research attempts to reduce the overhead incurred by the complex similarity analysis proposed by many existing works, since such complex similarity analysis may incur overhead on low-end, single-board computers running as edge nodes. In addition, the proposed approach is technology-independent. It can be added as a layer on top of existing job schedulers to complement their work by determining the best configuration of task clusters for a specific type of job.

## 4. Methodology

The methodology of this study relies on empirical experimentation by means of simulations to evaluate the impact of different task clustering techniques on the performance of Edge computing applications running on Raspberry Pis.

The details of the experimentation design are as follows.

The experiments are executed on a Raspberry Pi 3 Model B operated by Windows 10 IoT as its operating system. The Raspberry Pi consists of a 64-bit quad-core CPU running at 1.2 GHz with 1GB of RAM. The simulator is implemented using the .NET environment and the C# programming language. Therefore, the conducted experiments run simulated applications on an actual Raspberry Pi device.

The controlled experiments involve running a simulated application consisting of the following independent variables, which are set at the start of each experiment as configuration parameters:

- *Experiment type (P1)*: This parameter represents the clustering technique to be applied in the simulation, which can be either Temporal Clustering (TC), Sequential Clustering (SC), or Multiple-Instance Task Inversion Clustering (IC). For more details related to clustering techniques, interested readers can refer to [5].
- *Number of candidate tasks (P2)*: represents the number of tasks executed on the Raspberry Pi in the experiments. The number of tasks can be 8, 16, 32, 64, 128, 256, 512, 1024, or 2048. Each task simulates a workload that the Edge node performs according to the selected experiment type.
- *Workload size (P3)*: The simulated workload for each task is based on the black-scholes program of the PARSEC 3.0 benchmark suite for chip-microprocessors [20], [21]. Each task runs this program based on the simdev input, either 30 times (representing a *small* workload) or 60 times (representing a *larger* workload).
- *Tasks per task cluster (P4)*: To experiment with different cluster sizes given the selected number of tasks in the run, the number of tasks grouped into task clusters is varied using this configuration parameter. For example, if the number of tasks in temporal clustering experiments is set to 64. Then, it is possible to experiment with 2 clusters (each containing 32 tasks), 4 clusters (each containing 16 tasks), or 8 clusters (each containing 8 tasks). This study also considers the case in which each candidate task runs separately as an independent task (i.e., no clustering technique is applied).
- *Thread pool size (P5)*: This variable controls how many worker threads there are in the thread pool, which can be 32 or 128.
- *Input power (P6)*: This configuration parameter considers whether the Raspberry Pi is either (1) receiving sufficient power (connected to a 5.2V 2.5A power source) or (2) underpowered (connected to a 5.2V with up to 0.5A power source).

For each possible configuration (i.e., combinations of parameter values for P1-P6), the configuration is run 30 times, and the execution time of each experiment run is recorded:

- *Execution time*: The elapsed time from starting all tasks/task clusters in the experiment (according to the configuration of the independent variables) until all these tasks/task clusters are completed.

It should be noted that this study focuses on the worst-case scenario in these experiments, in which the system executes a number of tasks equal to the value set for *P2*. This allows for a better understanding of the impact of clustering techniques on system performance.

### *4.1. The Task Clustering Manager (TCM)*

Figure 2 presents a high-level view of the design of the proposed Task Clustering Manager (TCM). The manager acts as an intermediary between the Edge computing application and the underlying job scheduler provided by the programming framework or the operating system. The TCM design is based on the well-known MAPE-K model for autonomous systems [22]. This section describes each activity of the MAPE-K model performed by TCM.

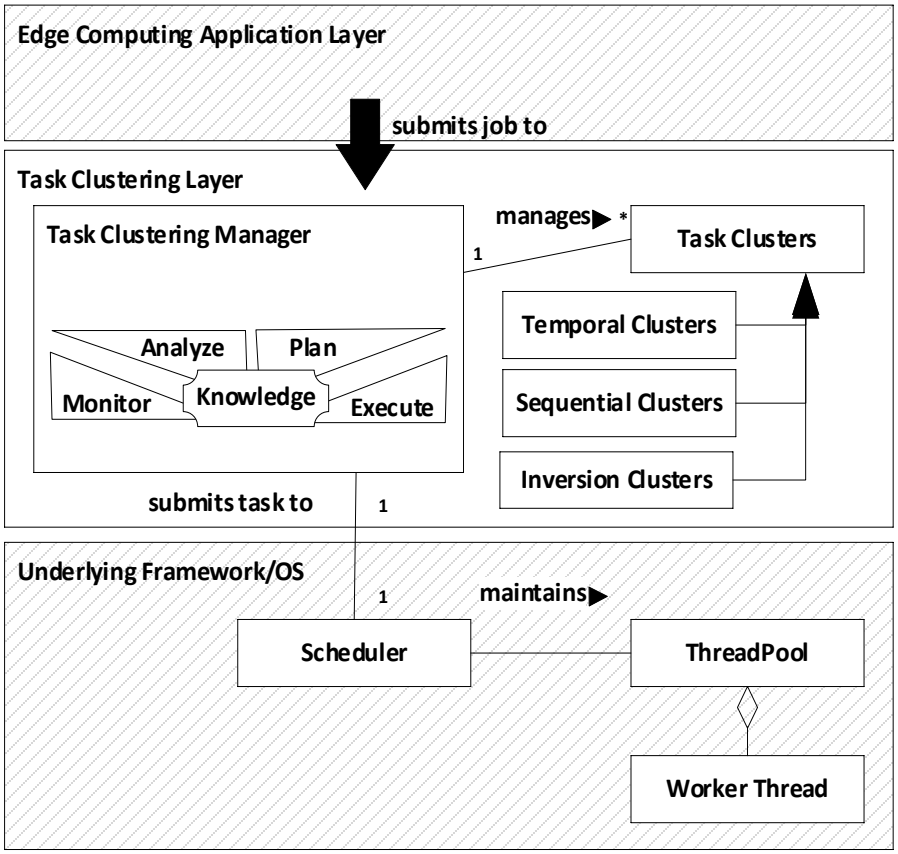


*Figure 2. Design overview of Autonomous Task Clustering Manager*

At design time, task types are marked with one of the marker interfaces shown in Figure 3:

- *Temporal*: A task type marked with this interface indicates to the TCM that it satisfies the Temporal Task Clustering criterion.
- *Sequential*: A task type marked with this interface indicates to the TCM that it satisfies the Sequential Task Clustering criterion.
- *Inversion*: A type marked with this interface indicates to the TCM that it satisfies the Multiple-Instance Task Inversion Clustering criterion.

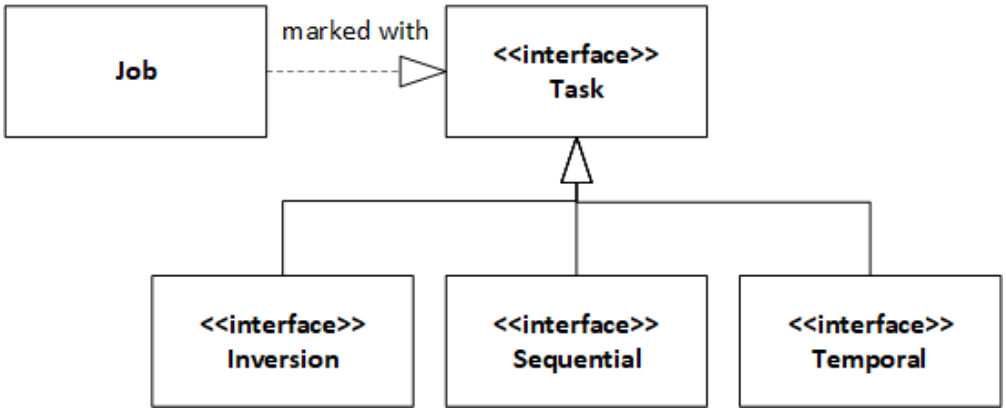


*Figure 3. Marker interfaces*

At run-time, the Edge computing application submits the tasks it needs to execute to the Task Clustering Manager. The Task Clustering Manager maintains several task clusters and assigns the tasks it receives to one of these clusters based on the task's marker interface. Furthermore, the Task Clustering Manager actively monitors and optimizes the number of active task clusters as explained next.

TCM's *monitoring* activity periodically monitors currently active task clusters. The monitoring activity awakens the Analysis activity if (1) The number of active tasks exceeds a predefined threshold $t$ or (2) Existing task clusters of the same type (i.e., temporal, sequential, or inversion) have an unbalanced number of candidate tasks inside them.

The responsibility of the *analysis* activity is to determine whether the current configuration of the Edge computing application requires reconfiguration. To accomplish this, the analysis activity performs the following rules:

- If (1) the number of currently active tasks running on the Raspberry Pi is greater than the value $c$ (which is set to be equal to the number of CPU cores) and (2) There are currently multiple task clusters of the same type (i.e., Temporal, Sequential, and Inversion clusters), then the analysis activity notifies the planning activity to reduce the number of clusters to match $c$.
- Otherwise, if (1) The number of currently active tasks running on the Raspberry Pi is lower than $c$ and (2) There are currently task clusters with more than one candidate task inside them, then the analysis activity notifies the planning activity to increase the number of task clusters to match the value $c$.
- Otherwise, if current task clusters have an unbalanced number of tasks, the analysis activity notifies the planning activity to balance the number of candidate tasks in task clusters.

The planning activity then crafts the plan for increasing or decreasing the number of task clusters under the management of TCM as follows. If the action determined by the analysis activity is to reduce the number of task clusters, then:

1. The planning activity determines the task cluster $C_{low}$, which currently has the lowest number of candidate tasks.
2. The planning activity determines the set $T$ of task clusters that have the same clustering type as $C_{low}$ (i.e., temporal, sequential, or inversion) in which it is possible to distribute the tasks from $C_{low}$ into $T$ according to Task Clustering Criteria (section 2).
3. The planning activity adds an action to (1) Distribute the tasks of $C_{low}$ over $T$ and (2) Remove $C_{low}$.
4. The planning activity repeats steps 1-3 to reduce the number of task clusters until it reaches the value $c$ or until no further clustering can be achieved.

On the other hand, if the action determined by the analysis activity is to increase the number of task clusters, then:

1. The planning activity determines the task cluster $C_{high}$, which currently has the highest number of tasks.
2. The planning activity adds the action of creating a new task cluster and moving half of the tasks from $C_{high}$ to this new cluster.
3. The planning activity repeats steps 1-2 to increase the number of task clusters until it reaches the value $c$ or until no more new task clusters can be created.

The planning activity can also create an action to redistribute the tasks over clusters in case of unbalanced clusters. Once the plan is determined, the planning activity activates the execution activity to initiate the reconfiguration process and passes the determined plan for execution.

The *execution* activity is responsible for executing the plan received from the previous activity with minimal interruptions. To accomplish this, the execution activity coordinates with task clusters affected by reconfiguration as follows:

1. If the tasks of a currently active temporal Cluster need to be reconfigured, the execution activity instructs this cluster to *passivate*. As a result, this cluster allows all currently active tasks within it to complete, so they are waiting for the next activation event. At this moment, the cluster has become *quiescent* [23], such that tasks are suspended until reconfiguration is completed. The execution activity then moves tasks according to the determined plan and activates the clusters.
2. The actions for reconfiguration of inversion clusters are similar to the previous case since inversion tasks are activated by external events rather than periodic events.
3. For sequential clusters, the execution activity instructs affected clusters to passivate. As a result, each sequential cluster allows the currently running sequential tasks to be completed. At this point, the execution activity proceeds in planned execution. Finally, the execution activity instructs sequential clusters to resume task execution.

## 5. Results

This section presents the simulation experimental results to assess the performance impact of different task clustering configurations by varying the configuration parameters discussed previously in Section 4, including (1) the thread pool size, (2) workload size, and (3) power input.

### *5.1. Results of Varying the Thread Pool Size*

The objective of this set of experiments is to assess whether the thread pool's size impacts performance under different cluster configurations. Therefore, the workload size is set to *small*, and the power input is set to *sufficient power*. All other parameters are varied in each run. Figure 4 shows the results of varying the size of the thread pool. Results when the thread pool size is 32 are shown in the left column of Figure 4, and results when the thread pool size is 128 are shown in the right column. It should be noted that the chart indicates the number of task clusters. When the number of jobs is too low (e.g., 8), some configurations are not applicable since there are not enough jobs for clusters.

As seen in this figure and for all clustering techniques, when the number of tasks is small (i.e., 8-64 tasks), the performance of all configurations is relatively similar.

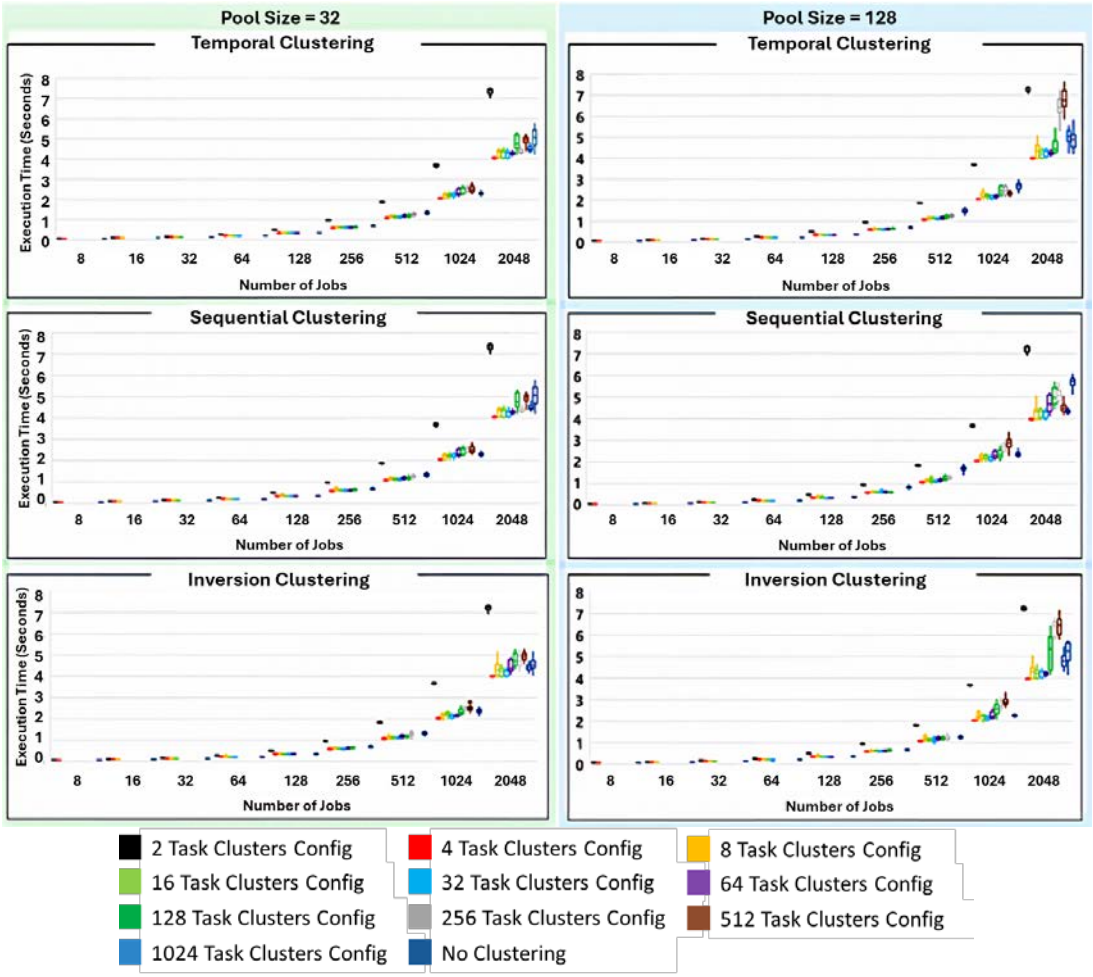


*Figure 4. Results of varying the number of worker threads in the thread pool*

However, as the number of tasks reaches 128, having two task clusters (each containing 64 tasks) starts showing performance degradation. This performance degradation becomes significant when there are more than 128 tasks. On the other hand, all other cluster configurations exhibited relatively equal results when the number of tasks was between 8 and 512.

In addition, as the number of tasks reaches the maximum permitted value (i.e., 2048 tasks), a specific cluster configuration (the red-marked configuration consisting of 4 clusters) shows better results than all other configurations. Furthermore, this cluster shows minimal fluctuation in terms of execution time compared to the other cluster configurations.

It can be concluded from these results that a four-cluster configuration is optimal in this set of experiments. This can be attributed to the fact that the Raspberry Pi contains a quad-core processor. Therefore, setting the number of tasks to less than four is restrictive, and a significant increase in the number of tasks can cause an overhead.

Furthermore, it can be seen that increasing the size of the thread pool has not improved performance and may have caused an increase in performance fluctuation in the different runs. This can be justified as many threads are blocked, waiting to be scheduled by the scheduler when a resource becomes available.

### *5.2. Results of Varying the Workload Size*

To evaluate the effect of workload size, the experiments are repeated by changing the workload size from small to larger while the power input is still set to sufficient power and fixing the thread pool size to 128. Similar to the previous set of experiments, in each run, all other parameters are varied to observe the performance of the simulated application. The effects of increasing the workload are displayed in Figure 5. This graph shows that as the number of tasks increases, the performance gap between various cluster setups becomes more noticeable. For example, in the case where P2=2048, the four-cluster configuration outperformed the two-clusters configuration by approximately 75%.

On the other hand, the four-cluster configuration outperformed the configuration without any clustering (i.e., each task is assigned its own thread) by approximately 25%. Therefore, the observation in this regard is consistent with the previous observation in that a four-clusters configuration outperformed all other configurations due to the same reason, which is having a quad-core processor.

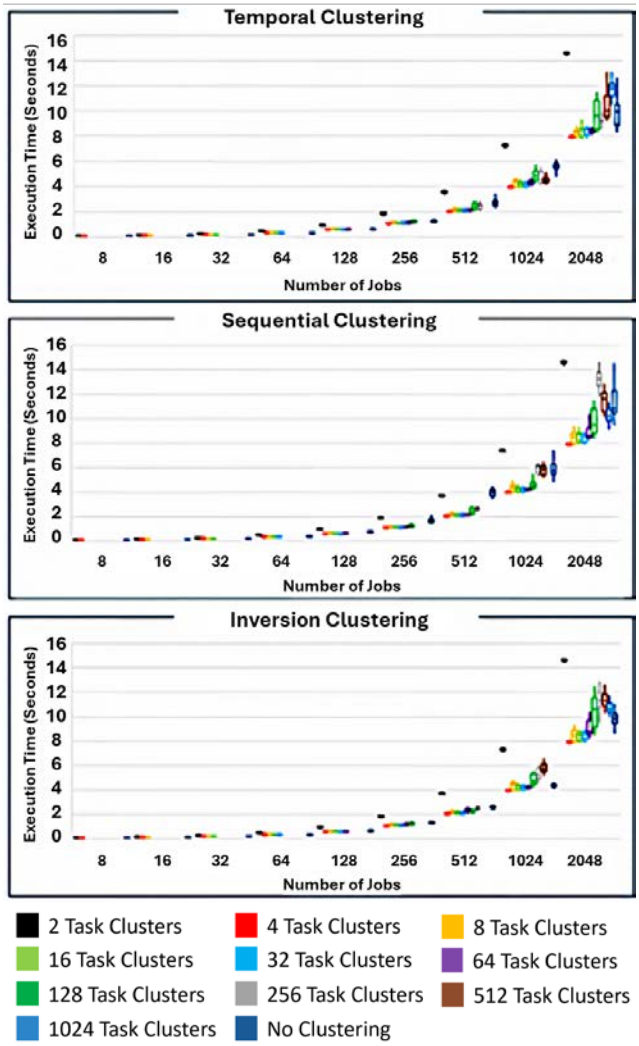


*Figure 5. Results of varying the workload size from small to large. Thread pool size = 128*

### *5.3. Results of Varying the Power Input*

In this set of experiments, the impact of reducing the Raspberry Pi's power input is examined. Therefore, the power input configuration parameter is set to underpowered while the workload size is set to small, and the thread pool size is 128. All other parameters are varied in different runs. When the Raspberry Pi is underpowered, significant performance degradation is observed at the higher range of a number of tasks, as shown in Figure 6 (compare the results with the results previously shown in Figure 4).

Surprisingly, the performance of the restricted two-cluster configuration is in the same range as other cluster configurations. This could result from the Raspberry Pi adjusting its computational resources based on input power. Despite this observation, the two-cluster configuration exhibited a large performance fluctuation compared to other cluster configurations, especially the four-cluster configuration, which showed more stable results in all runs.

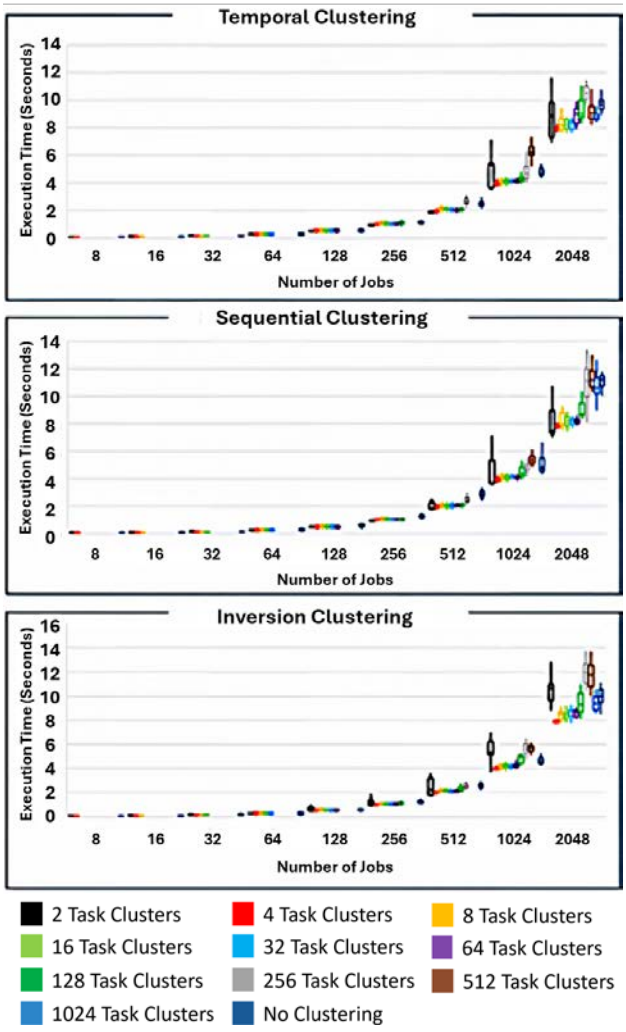


*Figure 6. Results of changing the power input from sufficient power to underpowered. Thread pool size = 128*

Therefore, the results in this set of experiments are consistent with the previous results reported in sections 5.1 and 5.2 in that a four-cluster configuration has shown better and more consistent performance across different experiments.

### *5.4. TCM Evaluation*

To evaluate TCM's capability of runtime adaptation, multiple simulation scenarios are conducted by means of experimentation to observe the behavior of TCM during (1) normal job submissions and (2) runtime reconfiguration of existing task clusters.

In each executed scenario, detailed log records of the execution are collected. The collected log records are then analyzed and compared with the expected behavior of TCM in maintaining the number of task clusters to the desired value (set 4) to achieve the experimental results discussed previously in section 5.

As an example of such a scenario, an Edge computing application submits 2048 temporal tasks to TCM. Execution logs (Figure 7) show that TCM created four temporal task clusters when it received the first four submitted tasks to match the value *c*. As a result, each task is assigned to a different cluster. With each subsequent task submitted to TCM, the action is to distribute these tasks over the four clusters instead of creating new ones. This is because TCM attempts to keep the number of task clusters matching the value *c*, which is set to 4.

Eventually, the application reached a configuration consisting of four temporal clusters, each containing 512 tasks, which is the desired behavior. This scenario also simulates the case in which all tasks of an existing task cluster terminate normally. Therefore, the number of active task clusters changes from 4 to 3. Inspection of the execution log shows that, eventually, the monitoring activity is activated to evaluate how many active tasks are running. Since this number is less than the desired value, the monitoring activity activates the analysis activity. Based on the rules in section 4.1, the analysis activity determines the need to increase the number of task clusters to match the number of CPU cores. As a result, the analysis activity activates the planning activity to plan this action. The planning activity then determines an appropriate plan by selecting a task cluster $C_{high}$ with the largest number of tasks, creating a new task cluster, and migrating half of the tasks from $C_{high}$ to this new cluster.

```
[Monitoring] number of active tasks is 3 which exceeds the threshold 4.
[Monitoring] activating Analysis activity.
[Analysis] the average number of jobs in task clusters is 512 jobs.
[Analysis] based on analysis, there are 3 task clusters with many clustered jobs.
[Analysis] activating Planing activity to increase the number of task clusters.
[Planning] determining plan to increase the number of task clusters.
[Planning] C_high =  temporal cluster with 512 jobs.
[Planning] adding action to move half jobs from C_high to a new Temporal cluster.
[Execution] suspending source cluster.
[Execution] source cluster suspended.
[Execution] created a new temporal cluster.
[Execution] moved jobs to new cluster.
[Execution] started new cluster.
[Execution] resumed source cluster.
[Execution] clusters= (Task1 with 512 jobs), (Task2 with 512 jobs), (Task3 with 256 jobs), (Task4 with 256 jobs)
```

*Figure 7. Example execution log of the behavior of TCM to increase the number of task clusters at runtime*

Finally, the execution activity performs these actions by (1) Instructing cluster $C_{high}$ with the largest number of tasks to passivate (i.e., allowing it to complete all currently executing tasks), (2) Creating a new task cluster, (3) Moving half of the tasks from $C_{high}$ with the largest number of tasks to the new cluster, (4) Running the new cluster, and (5) Resuming $C_{high}$ with the largest number of tasks. As a result, the new configuration now has 4 active task clusters to better match the number of CPU cores, which is the desired behavior.

## 6. Discussion

Simulation experiment results show that introducing the autonomous task clustering manager (TCM) to dynamically adapt task clustering configuration improved the performance of the Edge computing application running on the Raspberry Pi. TCM attempts to reduce the overhead of previously proposed methods, which perform complex similarity analysis of tasks.

To accomplish this, tasks submitted to the task clustering manager are marked to indicate whether they can be clustered under temporal, sequential, or inversion clustering techniques. Such marking of task types requires minor human effort. However, this effort is at the design time of the Edge computing application. At run-time, the task manager can cluster tasks without human intervention. Therefore, this is an acceptable trade-off compared to approaches that attempt to perform extensive similarity analysis at run-time because of the Raspberry Pi's limited computing capabilities.

As shown previously in section 5, the results show that no significant performance improvement is obtained when P2 (i.e., number of concurrent tasks) is low. However, as the number of concurrent tasks significantly increases, clustering tasks into an optimal configuration can result in observable performance improvement. This indicates that the Raspberry Pi is capable of running concurrent tasks, confirming prior results [1], [15].

However, as the number of concurrent tasks increases, automatic task clustering can reduce the overhead on these devices.

Therefore, the proposed approach is applicable in situations where the Edge computing application running on Raspberry Pis comprises numerous small tasks that are executed concurrently, due to incurring more execution overhead (e.g., in terms of scheduling and context switching overhead) that outweighs the benefits of concurrent execution.

Since the proposed task clustering manager is designed as an extension that can run on existing task schedulers, this approach complements prior works in task schedulers rather than replacing them.

Furthermore, this study focuses on specific types of tasks that can be clustered when clustering conditions are met. The proposed approach assumes that these conditions are implicitly satisfied since only tasks that meet these conditions will be marked during design time. Furthermore, it is assumed that these tasks are independent of each other. Therefore, future works are needed to consider relaxing these assumptions by investigating inter-task dependencies.

## 7. Conclusion

This research empirically evaluates the impact of three task clustering techniques (temporal, sequential, and multiple-instance task inversion clustering) on the performance of edge computing systems running on Raspberry Pis as Edge nodes. Simulation experimental results show that although these devices can handle concurrent task execution, performance can be notably improved when the number of concurrent tasks becomes large by applying clustering techniques to control the number of tasks based on the number of CPU cores. Experiment results show that an approximately 25% improvement in execution time can be achieved with a configuration that uses an appropriate task cluster configuration compared to a configuration that does not apply these techniques, where every task in the system is mapped to a task. Furthermore, this study presents the design for an autonomous task manager incorporating the MAPE-K loop to reconfigure the task architecture at runtime.

Future work includes investigating and evaluating the impact of these clustering techniques on the power consumption of these low-end, single-board devices. In particular, further studies can be conducted to consider whether the Raspberry Pi also conserves power with different cluster configurations. The results in this study show that the Raspberry Pi's performance behaviour significantly changes when it is underpowered. A possible future direction is investigating proper task clustering approaches under such circumstances.

**References:**

[1]. Gizinski, T., & Cao, X. (2022). Design, implementation and performance of an edge computing prototype using raspberry pis. *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*, 0592-0601. Doi: 10.1109/CCWC54503.2022.9720848

[2]. Stan, R. G., et al. (2021). Evaluation of task scheduling algorithms in heterogeneous computing environments. *Sensors*, *21*(17), 5906. Doi: 10.3390/s21175906

[3]. Cao, K., et al. (2020). An overview on edge computing research. *IEEE access*, *8*, 85714-85728. Doi: 10.1109/ACCESS.2020.2991734

[4]. Liu, F., et al. (2019). A survey on edge computing systems and tools. *Proceedings of the IEEE*, *107*(8), 1537-1562. Doi: 10.1109/JPROC.2019.2920341

[5]. Gomaa, H. (2016). *Real-time software design for embedded systems*. Cambridge University Press.

[6]. Mousa, M. H., & Hussein, M. K. (2022). Efficient UAV-based mobile edge computing using differential evolution and ant colony optimization. *PeerJ Computer Science*, *8*, e870. Doi:10.7717/peerj-cs.870

[7]. Yousif, A., Bashir, M. B., & Ali, A. (2024). An evolutionary algorithm for task clustering and scheduling in IoT edge computing. *Mathematics*, *12*(2), 281. Doi:10.3390/math12020281

[8]. Irani, J., Pise, N., & Phatak, M. (2016). Clustering techniques and the similarity measures used in clustering: A survey. *International journal of computer applications*, *134*(7), 9-14.
Doi: 10.5120/ijca2016907841

[9]. Saxena, A., et al. (2017). A review of clustering techniques and developments. *Neurocomputing*, *267*, 664-681. Doi: 10.1016/j.neucom.2017.06.053

[10]. Dautov, R., & Distefano, S. (2019). Automating IoT data-intensive application allocation in clustered edge computing. *IEEE Transactions on knowledge and Data Engineering*, *33*(1), 55-69.
Doi: 10.1109/TKDE.2019.2923638

[11]. Guo, T., Yu, K., Aloqaily, M., & Wan, S. (2022). Constructing a prior-dependent graph for data clustering and dimension reduction in the edge of AIoT. *Future Generation Computer Systems*, *128*, 381-394.
Doi:10.1016/j.future.2021.09.044

[12]. Nugroho, S., & Widiyanto, A. (2020). Designing parallel computing using raspberry pi clusters for IoT servers on apache Hadoop. *Journal of Physics: Conference Series*, *1517*(1), 012070. IOP Publishing.
Doi:10.1088/1742-6596/1517/1/012070

[13]. Miori, L., Sanin, J., & Helmer, S. (2017). A platform for edge computing based on Raspberry Pi clusters. *British International Conference on Databases*, 153-159. Springer International Publishing.
Doi: 10.1007/978-3-319-60795-5_16

[14]. Hassan, A., et al. (2022). Performance Evaluation of Raspberry Pi as an IOT Edge Signal Processing Device for a Real-Time Flash Flood Forecasting System. *International Journal of Advanced Computer Science and Applications*, *13*(10).
Doi: 10.14569/IJACSA.2022.01310100

[15]. Luo, Q., et al. (2021). Resource scheduling in edge computing: A survey. *IEEE communications surveys & tutorials*, *23*(4), 2131-2165.
Doi: 10.1109/COMST.2021.3106401

[16]. Bahadur, F., Umar, A. I., & Khurshid, F. (2018). Dynamic tuning and overload management of thread pool system. *International Journal of Advanced Computer Science and Applications*, *9*(11).
Doi: 10.14569/IJACSA.2018.091162

[17]. Abdelrehim, M. M., Hossny, E., & Omara, H. F. (2023). A Survey of the Most Common Task Scheduling Algorithms. *2023 International Telecommunications Conference (ITC-Egypt)*, 126-133.
Doi: 10.1109/ITC-Egypt58155.2023.10206179

[18]. Mahmoud, H., et al. (2020). A comparative study of heterogenous task-based scheduling techniques in a cloud environment. *2020 International Conference on Innovative Trends in Communication and Computer Engineering (ITCE)*, 1-6.
Doi: 10.1109/ITCE48509.2020.9047806

[19]. Amos, B. (2022). *PARSEC Benchmark Suite 3.0*. Github. Retrieved from: https://github.com/bamos/parsec-benchmark [accessed: 11 January 2026].

[20]. Zhan, X., Bao, Y., Bienia, C., & Li, K. (2017). PARSEC3. 0: A multicore benchmark suite with network stacks and SPLASH-2X. *ACM SIGARCH Computer Architecture News*, *44*(5), 1-16.
Doi: 10.1145/3053277.3053279

[21]. Kephart, J. O., & Chess, D. M. (2003). The vision of autonomic computing. *Computer*, *36*(1), 41-50.
Doi: 10.1109/MC.2003.1160055

[22]. Kramer, J., & Magee, J. (1990). The evolving philosophers problem: Dynamic change management. *IEEE Transactions on software engineering*, *16*(11), 1293-1306.
Doi: 10.1109/32.60317

[23]. Porter, J., & Albassam, E. (2020). A decentralized approach to architecture-based self-protecting software systems. *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, 169-175. Doi: 10.1109/CCWC47524.2020.9031205