Report for Task 2

Group 1

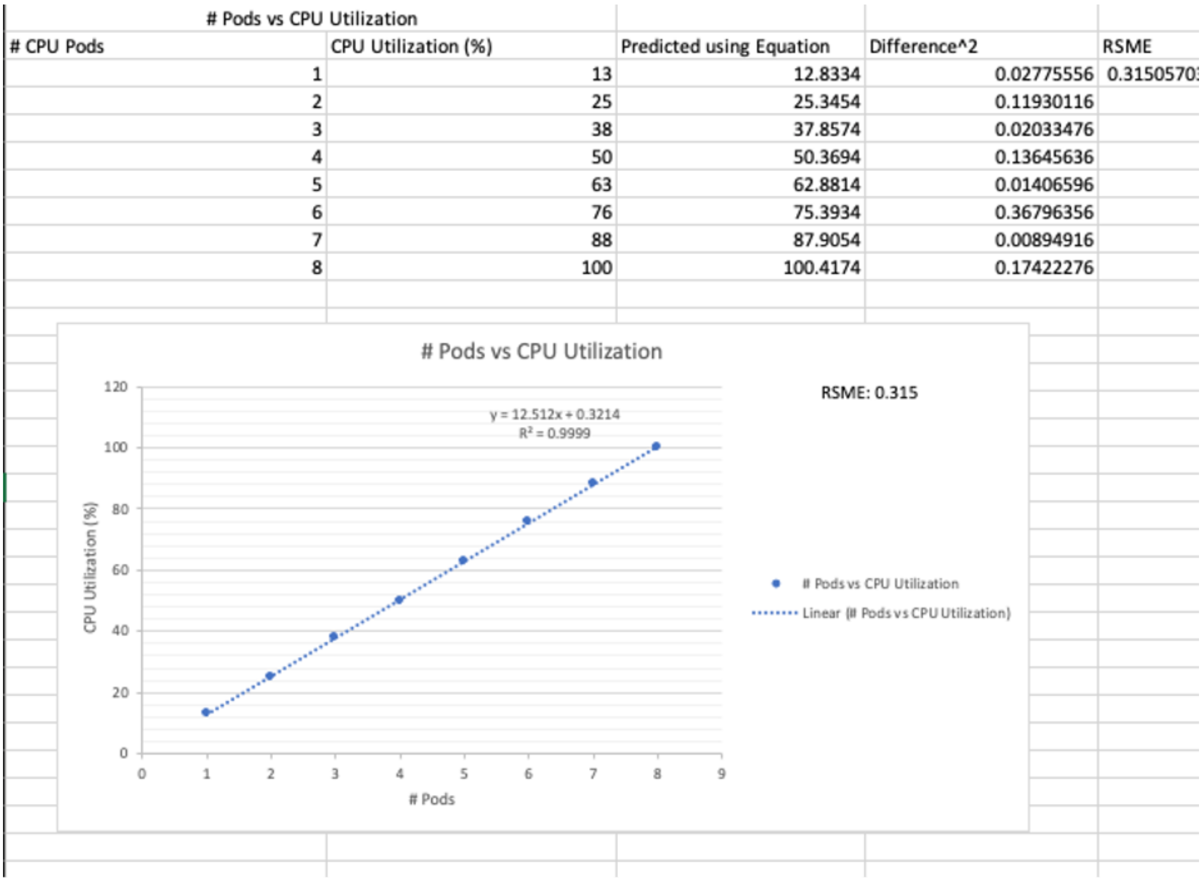
Group Members

Reinier Cruz Carnero
Anurag Chakraborty
Jotsna Gowda
Sumith Reddy Gutha
Darshini Ram Mattaparthi

Controller Design

Discussion process on the various models obtained by team members

Reinier Cruz Carnero:
Y = 12.512x + 0.3214



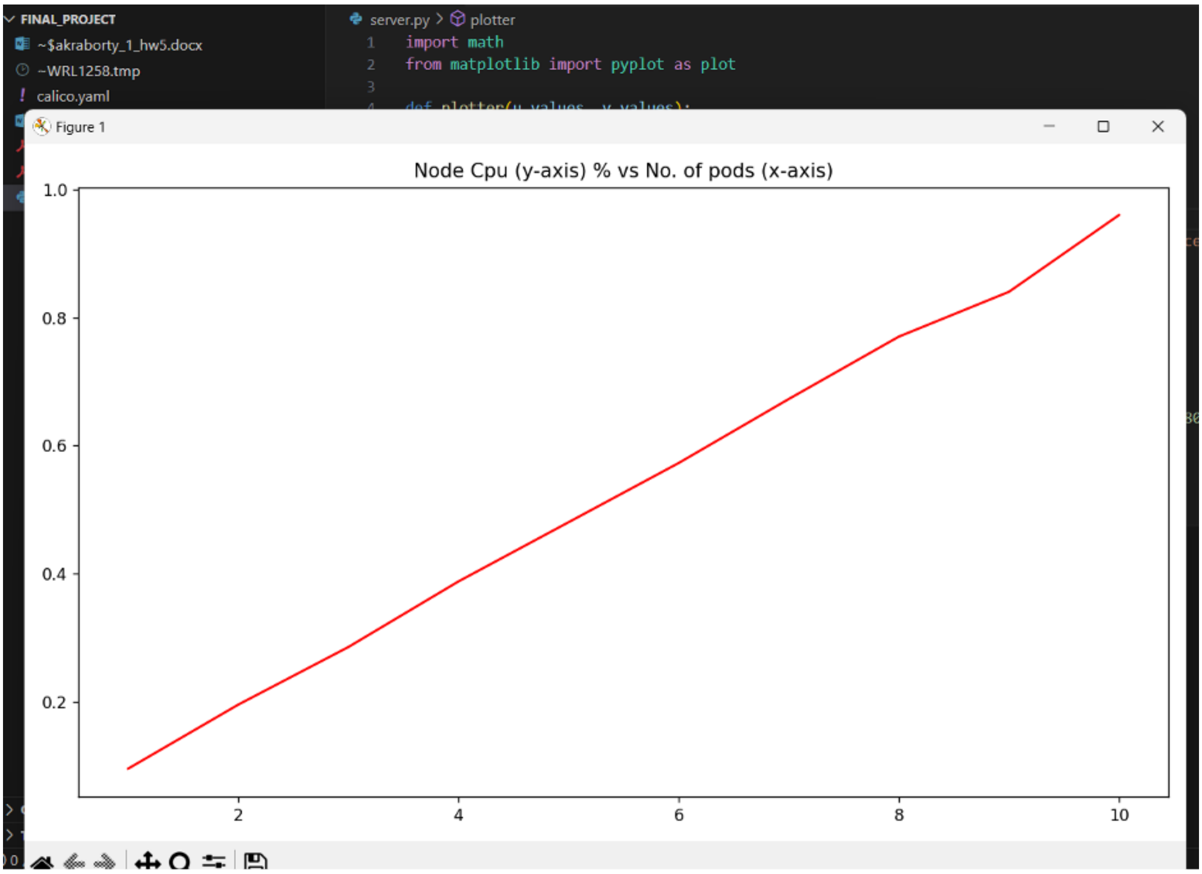
Pods vs CPU Utilization

# CPU Pods	CPU Utilization (%)	Predicted using Equation	Difference^2	RSME
1	13	12.8334	0.02775556	0.3150570
2	25	25.3454	0.11930116	
3	38	37.8574	0.02033476	
4	50	50.3694	0.13645636	
5	63	62.8814	0.01406596	
6	76	75.3934	0.36796356	
7	88	87.9054	0.00894916	
8	100	100.4174	0.17422276	

# Pods vs CPU Utilization
RSME: 0.315
y = 12.512x + 0.3214
R² = 0.9999
CPU Utilization (%)
120
100
80
60
40
20
0
0
1
2
3
4
5
6
7
8
9
# Pods
● # Pods vs CPU Utilization
······· Linear (# Pods vs CPU Utilization)

Anurag Chakraborty :

$y(k+1) = -0.119\ y(k) + 0.105\ u(k)$



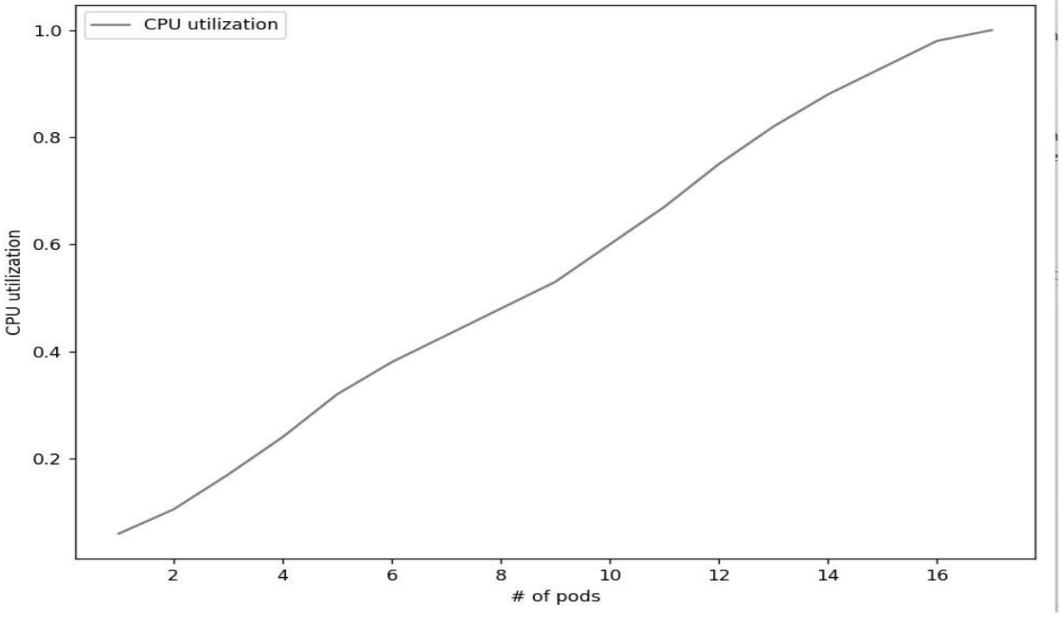
```
import math
from matplotlib import pyplot as plot

def plotter(u_values, y_values):
```

Jotsna:

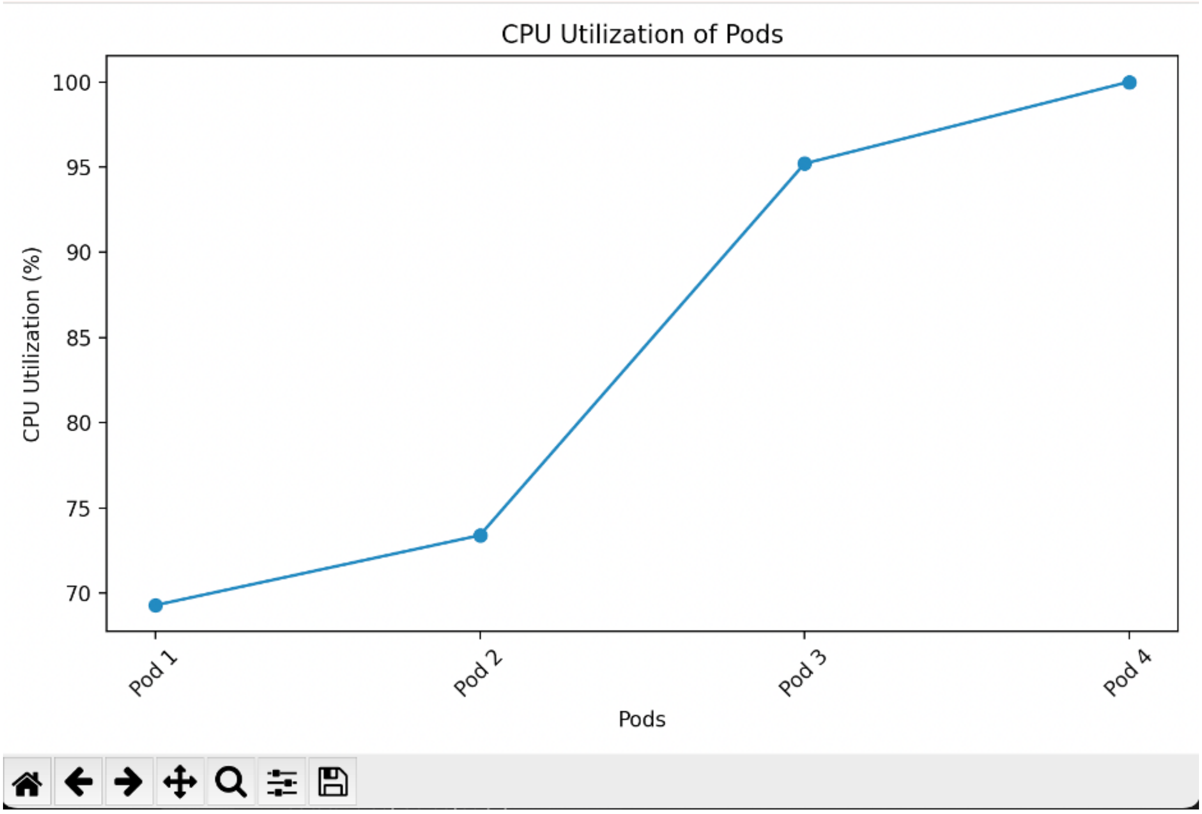
y(k+1)=0.032*y(k)+0.059*u(k)

3)The plot for number of pods vs CPU Utilzation,



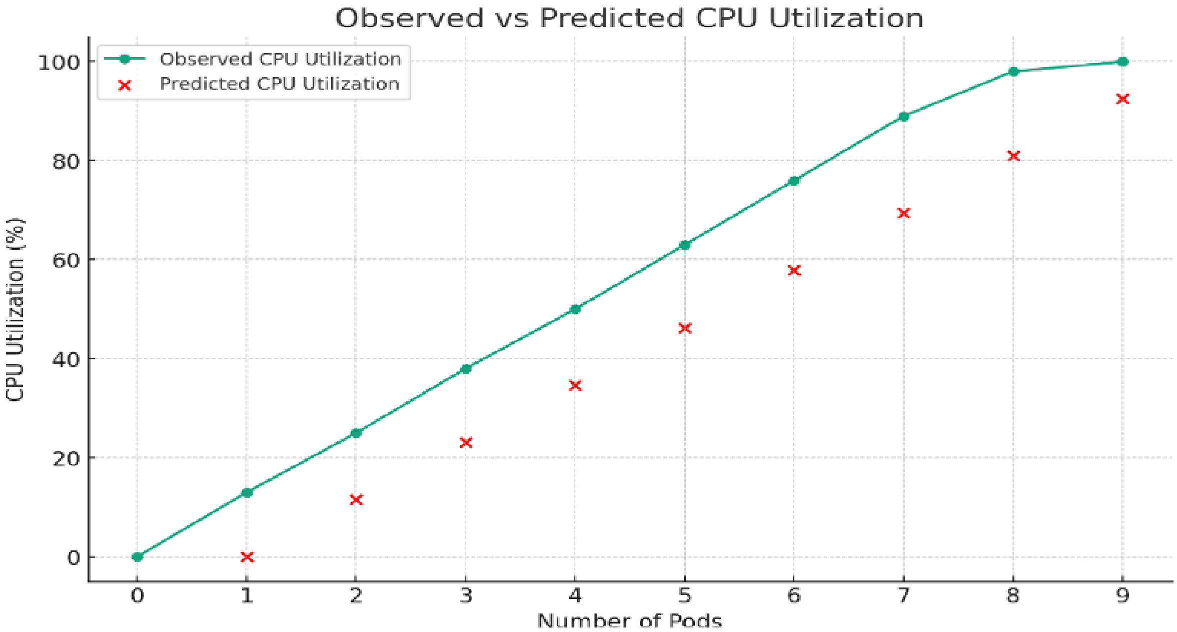
Darshini:

y(k+1)=-0.136*y(k)+15.2711*u(k)

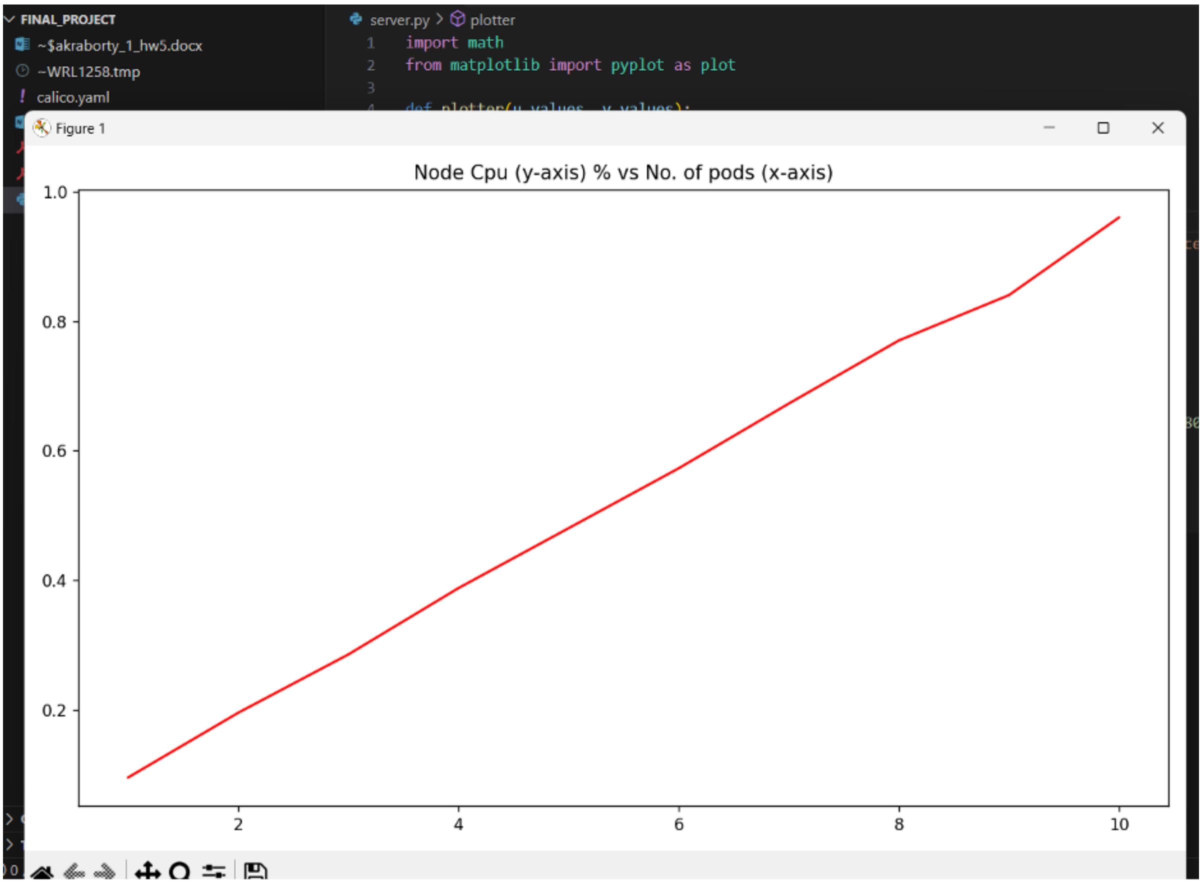


Sumith:

y(k+1)=0.0211*y(k)+11.3208*u(k)



Why the chosen model was considered the best.



```
import math
from matplotlib import pyplot as plot

def plotter(u_values, y_values):
```

Model with Averaging Utilization:

The selected model adopted an approach where it aggregated or averaged various stressor metrics to compute an overall utilization metric. This aggregated utilization metric was likely a representation of the overall stress or load on the Kubernetes cluster, considering multiple stressor types.

Generalization and Adaptability - By considering various stressors and creating a composite metric, this model aimed to capture a broader range of system conditions. It emphasizes adaptability, as it doesn't solely focus on specific stressors but considers a wider spectrum, allowing for a more generalized response.

Reduced Overfitting - By not simply increasing replicas of specific stressor pods, it attempts to mitigate overfitting. Overfitting occurs when a model learns specific details or noise in the training data that may not generalise well to new or diverse conditions. Averaging utilization across different stressors helps reduce this risk.

Models Increasing Replicas for Specific Stressor Pods:

Other models seemed to adopt a strategy of increasing the number of replicas for individual stressor pods in response to elevated utilization metrics specific to those stressors.

Overfitting Risk - Increasing replicas of specific stressor pods might lead to models that are overly tailored to handle those stressors. This strategy might result in models that perform well for those specific stressors but struggle to adapt to or generalize well across different stressor types or scenarios not explicitly accounted for during training.

Comparative Analysis:

Divergent Strategies - The selected model employing an averaged utilization approach aimed for a more holistic view of the system's load, considering various stressors and their collective impact. Other models, by increasing replicas for specific stressors, might excel in handling those stressors but could lack versatility in accommodating diverse stressor types.

Consideration for Generalization – The selection involves a trade-off between specialization for specific stressors and generalization to handle a wider range of stressor scenarios. While increasing replicas for specific stressors may provide precise handling, it risks limited adaptability to novel stressor types, potentially resulting in a lack of robustness and generalization.

Model Selection and Adaptability:

The selection of the model adopting an averaging utilization approach indicates a prioritization of adaptability and generalization to diverse stressors within the Kubernetes environment. This model aims to avoid overfitting to specific stressors, seeking a more versatile solution that can handle a broader spectrum of stressor scenarios for improved system resilience and performance under various conditions.

Explanation for the type of controller used and why it was chosen.

PID controllers are feedback mechanisms used in control systems. PID stands for proportional, integral, and derivative control. To reduce mistakes and get the system closer to the intended set point, it continuously modifies input. The derivative term predicts future error changes by considering their rate of development, the integral term gradually removes steady-state errors, and the proportional term reacts instantly to changes in errors. Kp, Ki, and Kd enable the controller to be adjusted to account for different system dynamics. To streamline the system and accomplish the intended goal, the team decided to use PID control.

Continuous Closed-Loop Control:
The PID controller offers a continuous closed-loop control mechanism that continuously monitors CPU utilization and adjusts job allocation in real-time.

Proportional, Integral, and Derivative Control:
The PID controller integrates proportional, integral, and derivative control actions, offering a comprehensive approach to system management. Provides immediate response to changes in CPU utilization, enabling quick adjustments. Eliminates steady-state errors over time, ensuring long-term stability and accuracy in maintaining the desired CPU utilization. Anticipates future changes in CPU utilization, aiding in anticipating trends and enhancing responsiveness.

Adaptable and Versatile Control:
The PID controller allows for adjusting the parameters (Kp, Ki, Kd) to fine-tune and optimize its behavior based on observed system responses. Provides adaptability to varying system conditions and different stressor scenarios, as it can adjust its control action based on real-time feedback.

Handling Multiple Stressors and Dynamic Changes:
In a Kubernetes environment with diverse stressors and changing workloads, the PID controller's continuous feedback loop can dynamically respond to fluctuations in CPU utilization caused by various stressors. Helps in handling various stressor types without being explicitly programmed for each, offering a more generalized approach to resource management.

Overfitting Avoidance and Generalization:
Compared to models solely increasing replicas for specific stressors, the PID controller's continuous adaptation aims to avoid overfitting to specific stressor scenarios. The PID controller, with its dynamic adjustments based on feedback, aims for a more generalized approach to resource management, potentially providing adaptability to a wider range of stressor scenarios.

The PID controller's ability to continuously monitor and dynamically adjust job allocation based on CPU utilization, its multi-faceted control actions, adaptability to changing environments, avoidance of overfitting, and potential for generalization make it a suitable choice for managing resources within a Kubernetes cluster in response to diverse stressors and dynamic workloads. Its characteristics align well with the need for adaptability, responsiveness, and versatility in such an environment, as indicated by the context provided in the codes and explanations.

Assumptions made for the controller's design.

The controller primarily relies on a proportional control mechanism to adjust the number of running jobs based on CPU utilization. The assumption that job allocation should vary proportionally with CPU utilization changes. Fixed Maximum Job Limit assumption is that there exists a fixed maximum limit (max_jobs) for concurrent job execution. The controller assumes this limit as a boundary for job allocation, not allowing jobs to exceed this predefined threshold. Assumption of a continuous closed-loop control system where CPU utilization is continuously monitored and used as feedback for adjusting job allocation. Real-Time Adjustment Assumption is a real-time adjustment of job allocation based on the current CPU utilization metrics. Utilization Threshold assumes a threshold or limit for CPU utilization (explicit or implicit) to trigger job allocation or deallocation. Real-Time Responsiveness assumes an immediate adjustment in job allocation in response to changes in CPU utilization. No Delay Assumption implies that the controller adjusts job allocation as soon as it detects variations in CPU utilization without any delay.

The assumptions made for the controller's design predominantly revolve around a proportional integral derivative control mechanism, a fixed job limit, continuous closed-loop control based on CPU utilization, immediate responsiveness to changes, dependency on CPU utilization as the primary metric, and a simplified control algorithm for job allocation adjustments. These assumptions form the foundation for the controller's design logic and behavior within the job management system.

This initial configuration assumes that the system starts with all available resources and modifies itself in response to feedback. However, the controller stops processing and displays an error message if the CPU utilization data is not available. This recognizes the importance of CPU utilization data and stops here in the absence of this vital information.

This particular simulation is designed for the node ["node1.reinierc-176345.ufl-eel6871-fa23-pg0.utah.cloudlab.us"]. The controller is concentrated on resource management for this specific node, so the assumptions are dependent on its availability.

Criteria and Constraints considered:

- Optimize the performance of the Kubernetes cluster. Dynamically manage the number of jobs based on CPU utilization to ensure efficient resource allocation.

- Efficiently utilize available resources in the Kubernetes cluster. Monitor CPU usage and adjust job allocation to maintain optimal resource utilization while avoiding overloading nodes.

- Efficiently manage jobs within the Kubernetes environment. Control the number of concurrently running jobs, ensuring they don't exceed predefined thresholds (max_jobs) while leveraging available resources effectively.

- Implement a closed-loop control system for job allocation. Continuously monitor CPU utilization, dynamically adjust job allocation, and potentially incorporate a controller for automatic adjustment based on utilization metrics.

- Handle potential errors or job completion scenarios. Monitor job completion status and terminate the script appropriately if jobs are completed, ensuring a graceful exit without errors.

- Maximum number of allowable jobs (max_jobs) is initially set to 2, implying a constraint on the maximum concurrent job execution.

- Code includes a controller mechanism for managing max_jobs based on CPU utilization. This implies a potential constraint in integrating or enabling this controller functionality within the current context.

- Exhibit continuous monitoring behavior for CPU usage and job management, suggesting ongoing monitoring without a defined termination condition or stopping mechanism.

- Interacts with the Kubernetes cluster to gather CPU and job-related metrics, assuming the availability and accessibility of these resources within the cluster.

- Desired CPU Utilization is assumed to be 90%.

The design process seems centered around optimizing performance, managing resources, and controlling job allocation based on CPU utilization metrics. Criteria and Constraints include limitations on the number of jobs, potential controller integration complexities, and a focused scope within a specific namespace. Constraints collectively shape the job management and resource utilization strategy within the Kubernetes environment.

Controller implementation

We implemented a PID controller in Python as part of the code provided for Kubernetes cluster simulations.

Initialization:
prev_error: It stores the previous error value initially set to 0.
integral_error: Represents the integral of the error over time, initialized to 0.
SamplingInterval: Indicates the time interval set to 1 unit (presumably seconds).

PID Controller Function:
pid_controller function implements a PID controller with proportional, integral, and derivative terms to compute a control adjustment based on the provided error and controller gains (Kp, Ki, Kd).

Computations:
Calculates the integral term as the sum of the previous integral error and the current error multiplied by the sampling interval (integral = integral_error + error * dt).
Derivative term is computed as the change in error over time (derivative = (error - prev_error) / dt).
Computes the control adjustment (adjust) using the PID formula: Kp * error + Ki * integral + Kd * derivative.

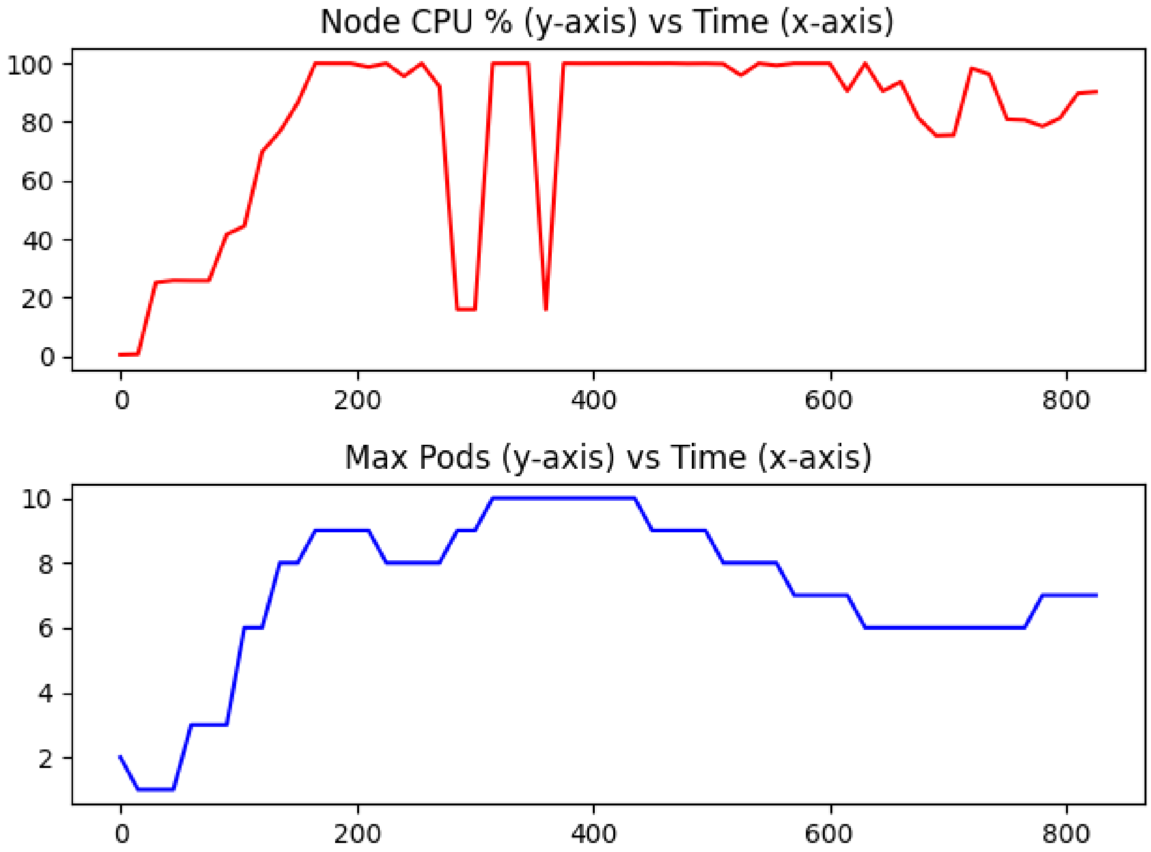
Controller Function:
controller function utilizes the PID controller to compute the control signal based on the average CPU usage and predefined controller gains (Kp, Ki, Kd).

Local Variables:
kp, ki, kd: Specific values for the proportional, integral, and derivative gains.
integral_error and prev_error are declared as global variables to maintain their values across function calls.
reference_point: Set as 0.9, representing the desired CPU usage.
error: Computes the current error as the difference between the reference point and the average CPU usage.

Controller Execution:
Calls the pid_controller function with the current error, previous error, and other controller parameters (gains and interval).
Obtains the control adjustment u and the updated integral error from the PID controller function.
Updates prev_error with the current error for the next iteration.
Prints the computed control adjustment.
If the computed control adjustment u is less than 1, the function returns 0; otherwise, it returns u.

It implements a PID controller (pid_controller) and a controller function (controller) that utilizes the PID controller to compute a control signal based on the error between a reference point and the average CPU usage. The PID controller handles proportional, integral, and derivative terms to generate a control adjustment (u) used for system control. The controller function outputs the control adjustment after applying certain conditions.

Results:



Conclusion:

We ran the controller for 50 randomized jobs. Initially our max_pods were set to 2, meaning 2 jobs would get picked first. The CPU % obtained from these 2 jobs was then fed as input to the PID controller and our observation was that for a reference value of 0.9, the controller returned a higher max_pod value. This went on until the CPU% reached 100% as shown in the graphs above. The controller did run a few more loops with similar max_pods with a 100% CPU utilization, and then it slowly started to return lower max_pods so that the CPU% would also reduce. We noticed that this process took some time to settle at around the reference value.

Report for Task 3

<u>Group 1</u>

Group Members

Reinier Cruz Carnero
Anurag Chakraborty
Jotsna Gowda
Sumith Reddy Gutha
Darshini Ram Mattaparthi

Introduction

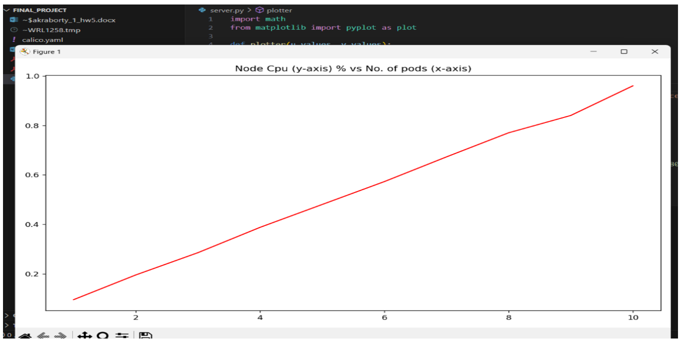
The task focuses on designing and implementing a Global Controller system tailored for effectively managing nodes within a cluster environment. The primary objective is to select an optimal controller capable of overseeing the total number of nodes while maintaining the cluster's overall CPU utilization below an 80% fixed threshold.

The Global Controller is designed to read from a job queue file periodically, every 15 seconds, and make decisions based on CPU utilization metrics. It possesses the functionality to call the local controllers which determine MAX_PODS for each node. This enables the Global Controller to assign jobs to nodes based on their current pod capacity, ensuring optimal resource utilization, and is equipped to handle node management within the cluster by facilitating the addition or removal of nodes. It involves the initiation and operation of both local and global controllers, along with various monitoring components within a node. Jobs from the job queue are periodically serviced by the cluster, and the controllers determine MAX_PODS and overall cluster node count based on CPU utilization. Simulated scenarios test the Global Controller's response to node failures, dynamic workload changes, and cluster resizing, ensuring its adaptability and effectiveness in managing cluster resources efficiently.

Controller Design:

The selected model adopted an approach where it aggregated or averaged various stressor metrics to compute an overall utilization metric:

y(k+1) = -0.119 y(k) + 0.105 u(k)



```
import math
from matplotlib import pyplot as plot
```

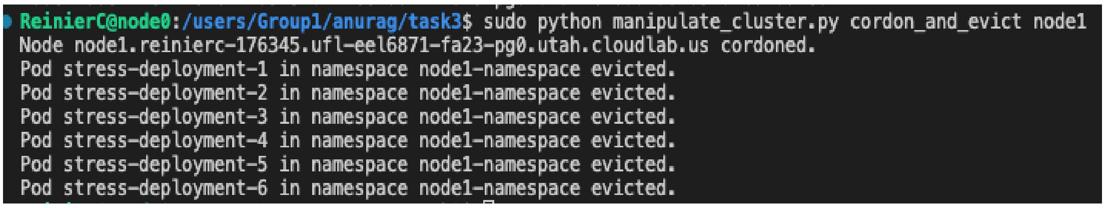
Rule based controller

In the context of managing nodes within a cluster environment and controlling CPU utilization, a rule-based controller was chosen.

For the fixed CPU utilization thresholds, periodic job queue readings, and node management actions, a rule-based controller was chosen for its suitability in handling these explicit conditions and executing corresponding actions efficiently. Its simplicity, transparency, and adaptability aligned well with the specified project requirements, making it a fitting choice for the task. Upon detecting the killing of nodes or changes in the cluster's composition, this controller swiftly responded by reallocating tasks and job assignments to the available nodes. By recognizing alterations in the cluster's state, the rule-based controller adeptly rerouted tasks to ensure continuous job processing and optimal resource utilization.

Simulation Scenarios F and G:

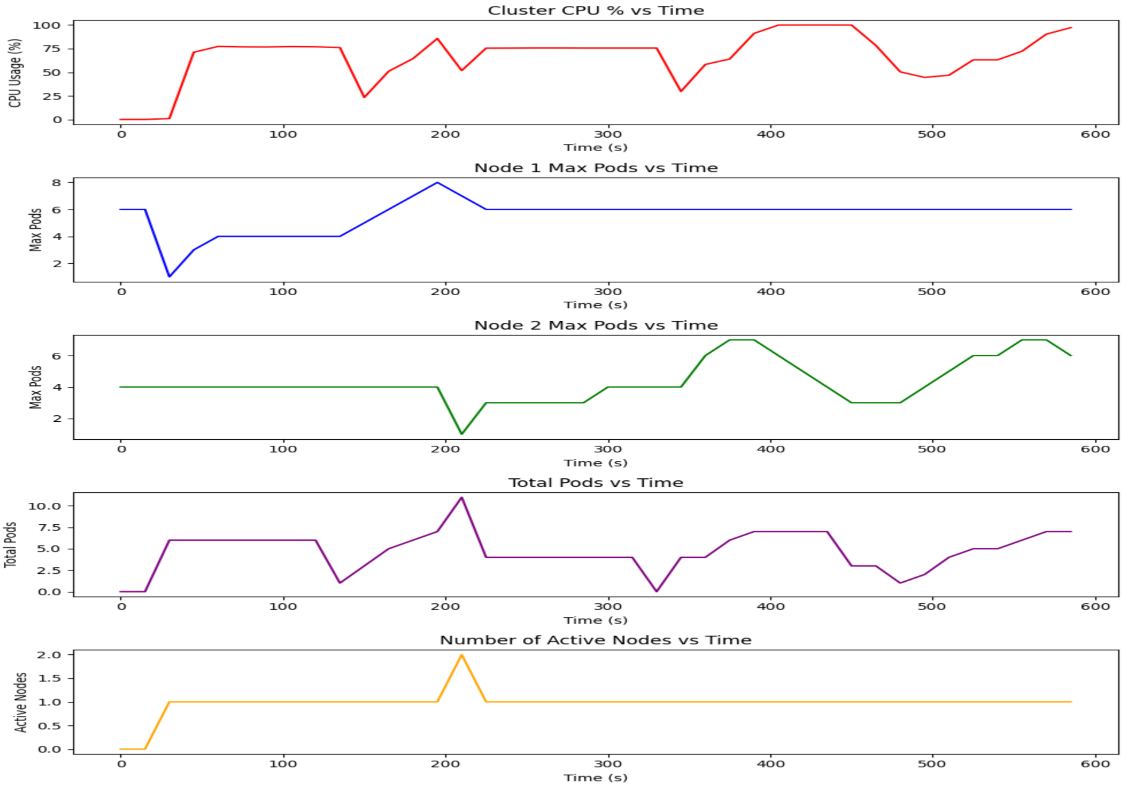
Simulation F Command, Ran after node 1 was at capacity



```
ReinierC@node0:/users/Group1/anurag/task3$ sudo python manipulate_cluster.py cordon_and_evict node1
Node node1.reinierc-176345.ufl-eel6871-fa23-pg0.utah.cloudlab.us cordoned.
Pod stress-deployment-1 in namespace node1-namespace evicted.
Pod stress-deployment-2 in namespace node1-namespace evicted.
Pod stress-deployment-3 in namespace node1-namespace evicted.
Pod stress-deployment-4 in namespace node1-namespace evicted.
Pod stress-deployment-5 in namespace node1-namespace evicted.
Pod stress-deployment-6 in namespace node1-namespace evicted.
```

Relating graphs to the simulation scenarios:

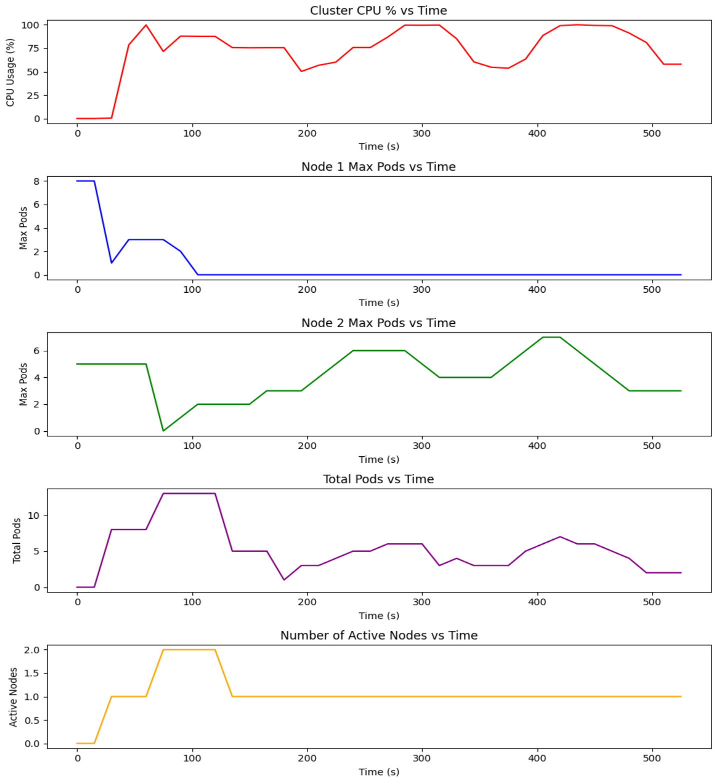
- The peak in the Cluster CPU % graph may correspond to the point where node-1 is running at capacity and cannot take new jobs, prompting the addition of node-2 to the cluster.
- The drop to one active node in the "Number of Active Nodes vs Time" graph likely represents the manual removal of node-1, after which only node-2 is active.
- Finally, the scale-down to one active node towards the end of the simulation represents the completion of all jobs on node-1 and the global controller's decision to reduce the number of active nodes in the cluster.



Execution stopped at 35 jobs

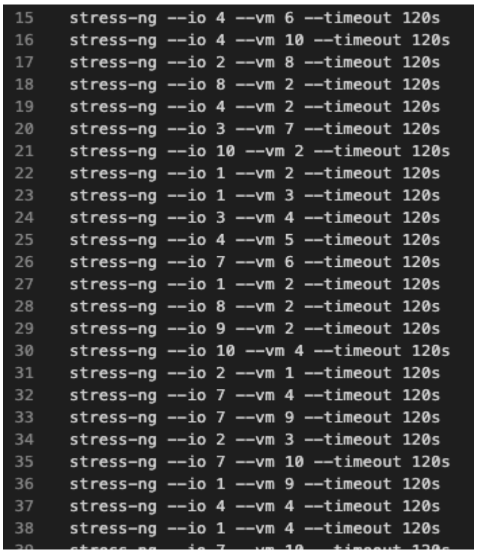
We set up the cluster environment with only node-1 operational. Designed job assignments in a way that saturates node-1's capacity. Then scheduling two additional jobs on node-2. Manually initiated the removal of node-1 from the cluster and monitored the Global Controller's behavior. The Global Controller detects the absence of node-1 and responds by abstaining from assigning any new jobs to it. The Global Controller redirects new job assignments exclusively to node-2, the remaining available node in the cluster, ensuring uninterrupted job processing. The peak seen in the graph is where node 2 gets two jobs and then node1 is removed shortly after.

For Scenario G:



Execution stopped at 30 jobs

We set up the cluster environment with both node-1 and node-2 operational. Allowed both nodes to reach capacity. Manually set a command to not allow new jobs to be placed on node1. We let all the jobs in node-1 complete their processing. After a designated time from the completion of jobs in node-1, the controller automatically initiated a scaling-down action. Triggered the Global Controller to scale down the number of active nodes in the cluster. Ensured that after the scaling-down process, only node-1 remains active.



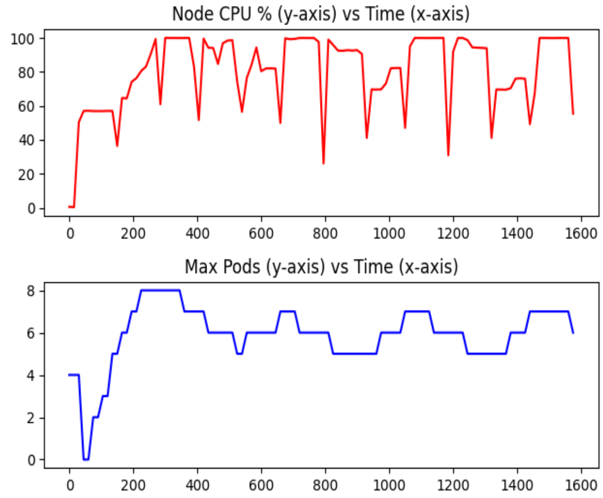
```
stress-ng --io 4 --vm 6 --timeout 120s
stress-ng --io 4 --vm 10 --timeout 120s
stress-ng --io 2 --vm 8 --timeout 120s
stress-ng --io 8 --vm 2 --timeout 120s
stress-ng --io 4 --vm 2 --timeout 120s
stress-ng --io 3 --vm 7 --timeout 120s
stress-ng --io 10 --vm 2 --timeout 120s
stress-ng --io 1 --vm 2 --timeout 120s
stress-ng --io 1 --vm 3 --timeout 120s
stress-ng --io 3 --vm 4 --timeout 120s
stress-ng --io 4 --vm 5 --timeout 120s
stress-ng --io 7 --vm 6 --timeout 120s
stress-ng --io 1 --vm 2 --timeout 120s
stress-ng --io 8 --vm 2 --timeout 120s
stress-ng --io 9 --vm 2 --timeout 120s
stress-ng --io 10 --vm 4 --timeout 120s
stress-ng --io 2 --vm 1 --timeout 120s
stress-ng --io 7 --vm 4 --timeout 120s
stress-ng --io 7 --vm 9 --timeout 120s
stress-ng --io 2 --vm 3 --timeout 120s
stress-ng --io 7 --vm 10 --timeout 120s
stress-ng --io 1 --vm 9 --timeout 120s
stress-ng --io 4 --vm 4 --timeout 120s
stress-ng --io 1 --vm 4 --timeout 120s
stress-ng --io 7 --vm 10 --timeout 120s
```

Sample from jobs.txt

It includes a variety of job types with different execution times and stress test jobs that run for 120 seconds each. The timeout for stress tests is 120 seconds, emphasizing the high-load nature of these stress test tasks while complying with a specific execution duration and timeout limit.

Results:

Task 2 Graph



Explanation

1. Node CPU % vs Time Graph: Shows the CPU utilization percentage of the node over time. The utilization fluctuates considerably, with peaks often reaching close to 100% and troughs going down to around 40%. These peaks and troughs represent the varying workload being processed by the node. But on average over the entire run the utilization is close to 80%.

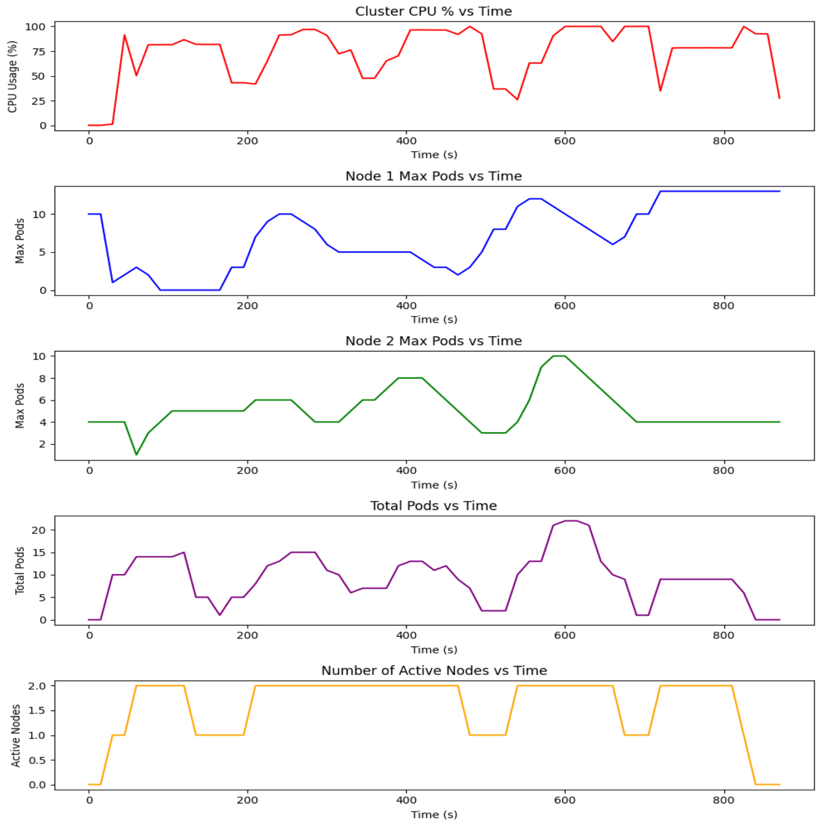
2. Max Pods vs Time Graph: Displays the maximum number of pods that the local controller has determined can be run on the node over time, based on the CPU utilization. The value of Max Pods changes throughout the simulation, which indicates the controller's response to the node's CPU utilization.

- The local controller's adjustment of the Max Pods is demonstrated by the changes in the Max Pods graph. The controller increases or decreases this value in response to the CPU utilization of the node, as shown in the first graph.
- When the Max Pods value is level, this suggests that the maximum number of pods is running on the node and no new jobs are started until there is a drop in CPU usage, allowing for more capacity.
- The controller increasing the value of Max Pods can be seen in the upward adjustments on the Max Pods graph. For example, around the 200-second mark, there is a sharp increase from 2 to 6 Max Pods. Conversely, the controller decreasing the value of Max Pods can be observed in the downward adjustments, such as the drop from 6 to 4 Max Pods just before the 400-second mark.

- Throughout the simulation, the CPU utilization data is collected, shown in the Node CPU % graph, and the Max Pods values are also tracked, as shown in the Max Pods graph. These metrics provide insight into the node's performance and the local controller's decisions in real-time. The run ends when all 75 jobs have been processed.

Task 3 Graph

General Run, with MAX_PODS initially set to 10 for node 1



The graph describes the collection of time-series data plotting different metrics relevant to a computer cluster's performance and resource management over time.

1. Cluster CPU % vs Time Graph: Shows the CPU utilization of the entire cluster over time. The local and global controllers would use this information to make decisions about resource allocation and scaling. The fluctuation in CPU usage indicates varying workload demands on the cluster.

2. Node 1 Max Pods vs Time Graph: It represents the maximum number of pods that can be scheduled on Node 1 based on its CPU utilization. As described in the spec point c, the local controller likely adjusts this value according to the current utilization of the node.

3. Node 2 Max Pods vs Time Graph: Similar to the Node 1 graph, this shows the maximum number of pods for Node 2. The changes over time suggest dynamic adjustment by its local controller.

4. Total Pods vs Time Graph: It displays the total number of pods across the cluster. It’s useful for monitoring the workload being processed by the cluster. As jobs are serviced every 15 seconds, this number would periodically change.

5. Number of Active Nodes vs Time Graph: It indicates how many nodes are active at any given time. It starts with one active node, steps up to two, and at some points, drops back to one.

Conclusion:

In this task, a system was devised with a primary emphasis on optimizing resource utilization and the main aim is to implement a global controller. This was achieved by dynamically adjusting the maximum number of pods (max_pods) based on real-time CPU utilization data. Our closed-loop control mechanism employs a controller that continuously monitors, provides feedback, and dynamically modifies max_pods to attain and sustain the desired CPU utilization level.

The simulation scenarios show the controller's adaptability to varying workloads and resource availability within a Kubernetes cluster. As time progresses, our system adjusts the number of deployed pods in correlation with utilization increases until it reaches the maximum limit set by max_pods. Over time, we observed through plotted data that CPU utilization fluctuates, consistently oscillating between rises and falls around the 80% utilization threshold we established.

Team Member’s Contribution

Reinier Cruz Carnero: globalContoller.py, stats.py, states.py, pseudocode for task2 controller
Anurag Chakraborty: pseudo code for task3 globalController.py, coded jobs.py, stats.py and plotter.py
Jotsna Gowda: coded task3 globalContoller.py and contributed to stats.py, plotter.py
Sumith Reddy Gutha: localController.py and manipulate_cluster.py
Darshini Ram Mattaparthi: Testing, Simulation and report