Revista Facultad de Ingeniería (Rev. Fac. Ing.) Vol. 33, No. 69.
L-ISSN: 0121-1129, e-ISSN: 2357-5328.

DOI: 10.19503/01211129.v33.n69.2024.18057

PERFORMANCE ANALYSIS OF ACCESS AND MOBILITY MANAGEMENT FUNCTION ON A 5G CORE BASED ON CPU USAGE PREDICTIONS

Análisis de desempeño de la función de gestión de acceso y movilidad en un Core 5G basado en predicciones del uso de CPU

Wilmar-Yesid Campo-Muñoz
Ph. D. Universidad del Quindío (Armenia-Quindío, Colombia).
wycampo@uniquindio.edu.co

Jhon-Alexander Amaya-Suárez
M. Sc. Universidad del Quindío (Armenia-Quindío, Colombia).
jaamayas@uqvirtual.edu.co

Juan-Camilo Caviedes-Valencia
M. Sc. Universidad Nacional de Colombia (Bogotá-Distrito Capital, Colombia).
jcaviedesv@unal.edu.co

Received / Recibido: 22-05-2024
Accepted / Aceptado: 31-08-2024

ABSTRACT

The increasing number of mobile devices and the growing demand for services lead to an increase in the access requests per second to the Access and Mobility Management Function (AMF) of the control plane in a Fifth Generation (5G) mobile network. It causes congestion of the function and affects the overall network performance. Therefore, this paper proposes a self-scaling mechanism for the AMF in a 5G core by CPU usage predictions using the Long Short-Term Memory (LSTM) machine learning (ML) technique. The mechanism predicts the percentage of CPU usage in the pod containing the AMF and establishes scaling policies that determine the necessary number of AMF pods. The performance of the AMF is evaluated through success rate, loss rate, and latency of access requests per second in three scenarios: a reactive one with scaling based on current CPU thresholds, a predictive one using CPU predictions, and another using both the scaling policies and the LSTM technique. With the previous scenarios, the AMF is scaled reactively and predictively. Results show that the scaling policies and the ML algorithm significantly improve the performance of the function in terms of success rate and loss rate of access requests per second. An efficient self-scaling of the AMF is achieved, which contributes both to the optimization of computational resources and to improving the availability of the 5G mobile network.

Keywords: AMF; CPU; LSTM; scaling policies; self-scaling.

Esta edición se financió con recursos del Patrimonio Autónomo Fondo Nacional de Financiamiento para la Ciencia, la Tecnología y la Innovación, Francisco José de Caldas, Minciencias

How to cite: W-Y. Campo-Muñoz , J-A. Amaya-Suárez, J. C. Caviedes-Valencia, “Performance Analysis of Access and Mobility Management Function on a 5G Core Based on CPU Usage Predictions”, *Revista Facultad de Ingeniería*, vol. 33, no. 69, e18057, 2024. https://doi.org/10.19053/01211129.v33.n69.2024.18057

RESUMEN

El aumento en el número de dispositivos móviles y la creciente demanda de servicios generan un incremento en las solicitudes de acceso por segundo que llegan a la función de gestión de acceso y movilidad (AMF, *Access and Mobility Management Function*) del plano de control en una red móvil de quinta generación (5G, *Fifth Generation*), lo que provoca congestión en la función y afecta el desempeño general de la red. En este artículo se propone un mecanismo de autoescalado para el AMF en un *core* 5G utilizando predicciones de uso de la CPU obtenidas mediante la técnica de aprendizaje automático (ML, *Machine Learning*) de memoria a largo y corto plazo (LSTM, *Long Short-Term Memory*). El mecanismo predice el porcentaje de uso de la CPU en el *pod* que contiene la AMF y establece políticas de escalado que determinan la cantidad necesaria de *pods* AMF. Se evalúa el desempeño del componente AMF a través de la tasa de éxito, tasa de pérdidas y latencia de las solicitudes de acceso por segundo en tres escenarios diferentes: uno reactivo con escalado basado en límites (*Thresholds*) actuales de CPU, otro predictivo utilizando predicciones de CPU, y otro en el que se involucran tanto las políticas de escalamiento como la técnica LSTM. Con los escenarios anteriores, se escala el AMF de forma reactiva y predictiva. Los resultados muestran que las políticas de escalamiento y el algoritmo de ML mejoran significativamente el desempeño de la función en términos de tasa de éxito y tasa de pérdidas de solicitudes de acceso por segundo. Se logra implementar un autoescalado eficiente del AMF, lo cual contribuye tanto a la optimización de recursos computacionales como a mejorar la disponibilidad de la red móvil 5G.

Palabras clave: AMF; autoescalamiento; CPU; LSTM; políticas de escalamiento.

1. INTRODUCTION

The number of subscribers to fifth-generation 5G mobile networks is projected to reach 4.4 billion by the end of 2027, thus becoming the main mobile access technology [1]. Consequently, the increase in the number of user equipment (UE) will be accompanied by an increase in traffic, both at the user and control levels of the mobile network [2] [3]. This increase especially affects the Access and Mobility Management Function (AMF) of the control plane in a 5G network. The AMF is critical for managing and controlling access and mobility in the UE. However, the increase in access requests per second overloads the function and affects the overall performance of the network because it is the only access point for the control plane of the 5G mobile network [3], [4].

To handle this situation, the 3GPP (3rd Generation Partnership Project) has created the 5G architecture with the purpose of making it more flexible and scalable than its predecessors [5], [6], [7], [8]. As a result, several scaling strategies that allow increasing the capacity of network components depending on the workload they receive have been developed [9] [10]. Nevertheless, these strategies are often reactive and do not consider the future needs of the network [11]. There are no proposals for autoscaling strategies for the AMF in 5G networks in the literature. Then, it is important to note that the increase or decrease in the number of users per second is directly related to the loading or unloading of computational resources such as the Central Processing Unit (CPU) [12], [13].

Accordingly, this paper analyzes the performance of the AMF component in autoscaling scenarios through different metrics such as the success rate, loss rate, and access request latency in three different scenarios. These scenarios are built in an AWS cloud and Kubernetes is used to orchestrate the pods that store the containers; each component of a 5G network architecture is configured there. To automatically scale the number of AMF pods, an autoscaling algorithm is created to predict the CPU usage value. It takes as input a historical sequence of metrics in the form of time series. Also, scaling policies are defined to avoid erroneous scaling and prevent the waste of computational resources such as CPU.

The first scenario is reactive, so when access requests increase, the algorithm scales-out or scales-in AMF pods. The second scenario is predictive as the algorithm already scaled-out or scaled-in the pods in the component when the access requests change. In the third scenario, both scaling policies and the LSTM (Long Short-Term Memory) Machine Learning technique are involved [14], [15], [16]. In this scenario, CPU predictions go through scaling policies to define the number of pods. The LSTM and self-scaling policies improve the performance of the function in terms of success rate and loss rate of access requests.

This article is organized as follows: Section 2 presents the methodology used to conduct this research; Section 3 presents the results and their discussion; Section 4 presents the conclusions and future work derived from this research.

2. METHODOLOGY

Figure 1 presents the architecture where tools and technologies used in the different experimental scenarios are integrated.

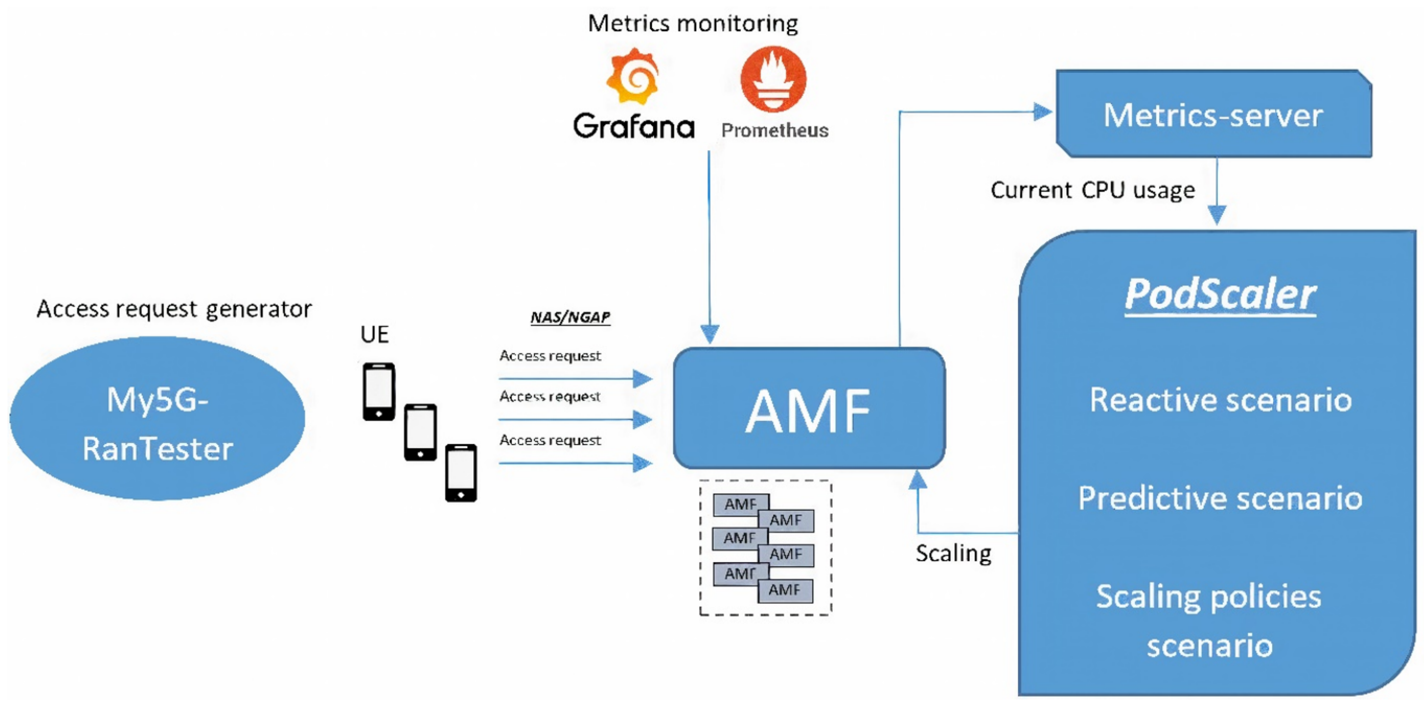


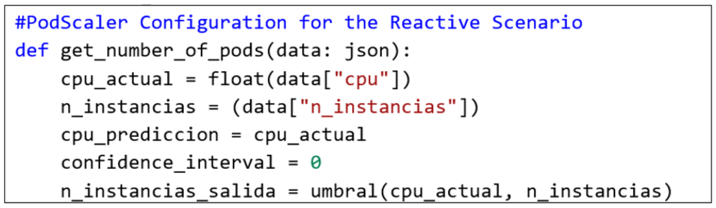
Figure 1. *Testbed architecture.*

Initially, My5G-RanTester simulates user terminals to start generating 5G mobile network access requests to Open5GS. These requests are managed by the AMF pods, which are monitored by Prometheus and Grafana to collect metrics. At the same time, the Kubernetes Metrics Server takes the current CPU usage and sends it to the PodScaler module for testing the experimental scenarios. In a final stage, the dynamic scaling of the AMF pods is executed based on the chosen scenario.

3. RESULTS AND DISCUSSION

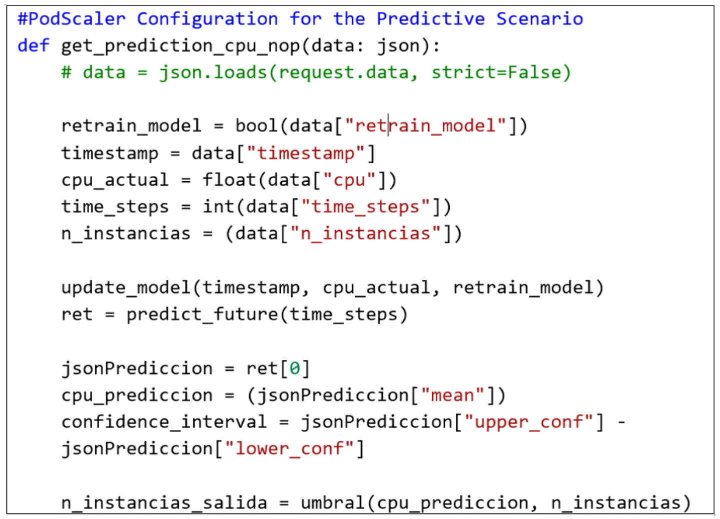
Three experimental scenarios are built: a reactive scenario, a predictive scenario, and a self-scaling policies scenario.

1. Reactive scenario: Here, the AMF performance is validated only with CPU levels, that is, when access requests increase, the AMF pods scales-out or scales-in based on a threshold. Figure 2 presents the configuration of the PodScaler, where variable cpu_prediccion is matched to cpu_actual, and variable n_instancias_salida is matched to the umbral function and sent cpu_actual.
2. Predictive scenario: Here, the AMF performance is validated with levels based on CPU usage predictions. Prior to the increase of access requests, the algorithm has already performed self-scaling actions. This scenario does not include self-scaling policies. Figure 3 presents the configuration of the PodScaler in such a way that the n_instancias_salida variable is matched to the umbral function and sent cpu_prediccion.



```
#PodScaler Configuration for the Reactive Scenario
def get_number_of_pods(data: json):
    cpu_actual = float(data["cpu"])
    n_instancias = (data["n_instancias"])
    cpu_prediccion = cpu_actual
    confidence_interval = 0
    n_instancias_salida = umbral(cpu_actual, n_instancias)
```

Figure 2. *PodScaler configuration for the reactive scenario.*



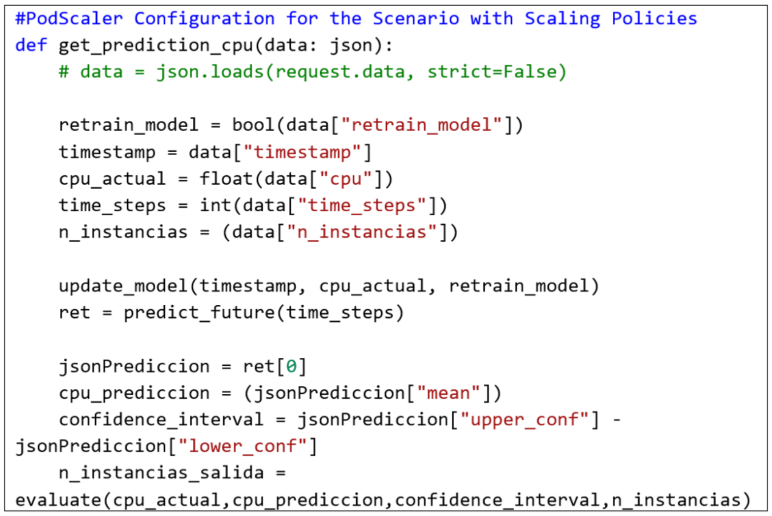
```
#PodScaler Configuration for the Predictive Scenario
def get_prediction_cpu_nop(data: json):
    # data = json.loads(request.data, strict=False)

    retrain_model = bool(data["retrain_model"])
    timestamp = data["timestamp"]
    cpu_actual = float(data["cpu"])
    time_steps = int(data["time_steps"])
    n_instancias = (data["n_instancias"])

    update_model(timestamp, cpu_actual, retrain_model)
    ret = predict_future(time_steps)

    jsonPrediccion = ret[0]
    cpu_prediccion = (jsonPrediccion["mean"])
    confidence_interval = jsonPrediccion["upper_conf"] -
    jsonPrediccion["lower_conf"]

    n_instancias_salida = umbral(cpu_prediccion, n_instancias)
```

Figure 3. *PodScaler configuration for the predictive scenario.*

3. Self-scaling policies scenario: Here, the self-scaling policies and the LSTM model are used together. CPU predictions are integrated with scaling policies to determine the appropriate number of pods, seeking to improve resource efficiency and availability of the 5G mobile network. In the following lines of code, the PodScaler is configured with the n_instancias_salida variable matched to the evaluate

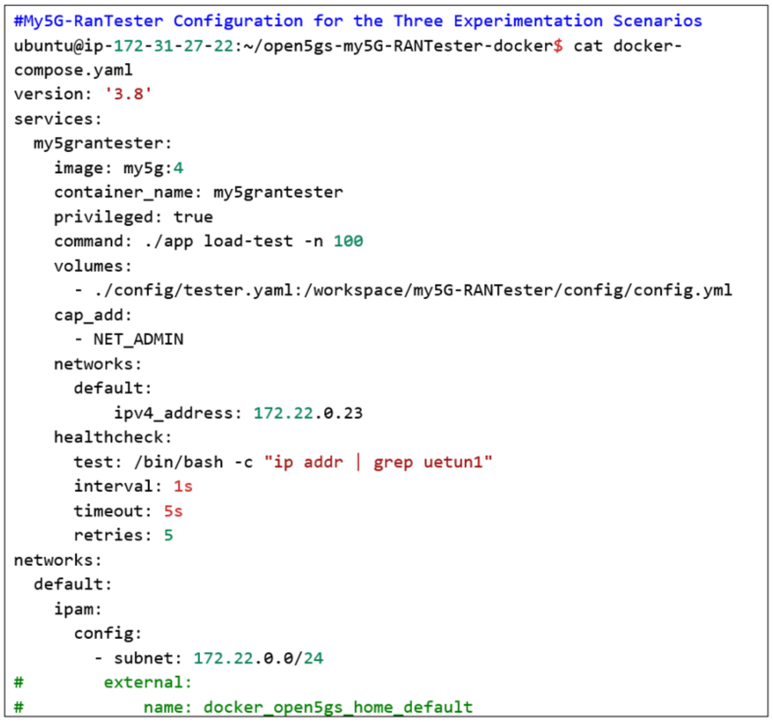
function that contains the scaling policies. Figure 4 shows the configuration of the PodScaler with the n_instancias_salida variable matched to the evaluate function.

Once the PodScaler has been configured for each experimental scenario, the access request generator is configured. In all scenarios, My5G-RanTester is used to generate a validation interval of eight hours and to send 10000 access requests per second to the network, as shown in Figure 5. Subsequently, it is used to evaluate the performance of the AMF component (Figure 6).



```
#PodScaler Configuration for the Scenario with Scaling Policies
def get_prediction_cpu(data: json):
    # data = json.loads(request.data, strict=False)

    retrain_model = bool(data["retrain_model"])
    timestamp = data["timestamp"]
    cpu_actual = float(data["cpu"])
    time_steps = int(data["time_steps"])
    n_instancias = (data["n_instancias"])

    update_model(timestamp, cpu_actual, retrain_model)
    ret = predict_future(time_steps)

    jsonPrediccion = ret[0]
    cpu_prediccion = (jsonPrediccion["mean"])
    confidence_interval = jsonPrediccion["upper_conf"] -
jsonPrediccion["lower_conf"]
    n_instancias_salida =
evaluate(cpu_actual,cpu_prediccion,confidence_interval,n_instancias)
```

Figure 4. *PodScaler configuration for the self-scaling policies scenario.*



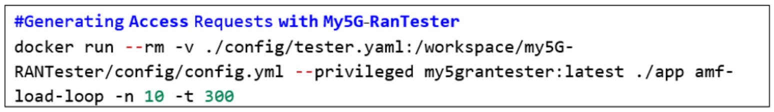
```
#My5G-RanTester Configuration for the Three Experimentation Scenarios
ubuntu@ip-172-31-27-22:~/open5gs-my5G-RANTester-docker$ cat docker-
compose.yaml
version: '3.8'
services:
  my5grantester:
    image: my5g:4
    container_name: my5grantester
    privileged: true
    command: ./app load-test -n 100
    volumes:
      - ./config/tester.yaml:/workspace/my5G-RANTester/config/config.yml
    cap_add:
      - NET_ADMIN
    networks:
      default:
          ipv4_address: 172.22.0.23
    healthcheck:
      test: /bin/bash -c "ip addr | grep uetun1"
      interval: 1s
      timeout: 5s
      retries: 5
networks:
  default:
    ipam:
      config:
        - subnet: 172.22.0.0/24
#        external:
#            name: docker_open5gs_home_default
```

Figure 5. *My5G-RanTester configuration for all scenarios.*

The access requests generated by My5G-RanTester create the total CPU peaks in the AMF pods shown in Figure 7 for the reactive, predictive, and self-scaling policies, respectively. The vertical axis of the figure represents the variation of CPU in millicores (mcs), with a scale of maximum 24 mcs. To facilitate the analysis, the 24 mcs are evenly distributed across 4 pods, resulting in 6 mcs per pod. It is possible to take more or less than 4 pods in the AMF component to handle total CPU usage and the dynamics of analyzing network metrics to evaluate the performance of the AMF will be the same. Therefore, a limit of 6 mcs per AMF pod is set in the PodScaler.

With the testbed working, the performance of the AMF component is validated according to its CPU usage in all scenarios.

The following 6 metrics are evaluated in every scenario: (i) total sum of CPU usage of the AMF pods over time; (ii) Total number of AMF pods over time; (iii) Total CPU usage vs Total number of AMF pods over time; (iv) Successful access requests to the 5G network; (v) Failed access requests to the 5G network; and (vi) Average latency of successful access requests over time.



```
#Generating Access Requests with My5G-RanTester
docker run --rm -v ./config/tester.yaml:/workspace/my5G-
RANTester/config/config.yml --privileged my5grantester:latest ./app amf-
load-loop -n 10 -t 300
```

Figure 6. *Access Requests generated using My5G-RanTester.*

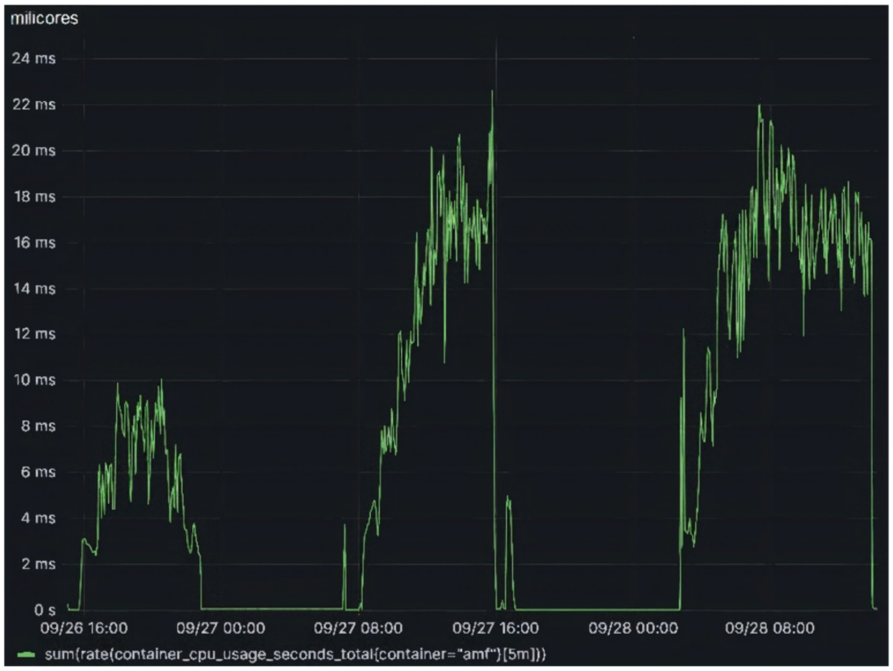
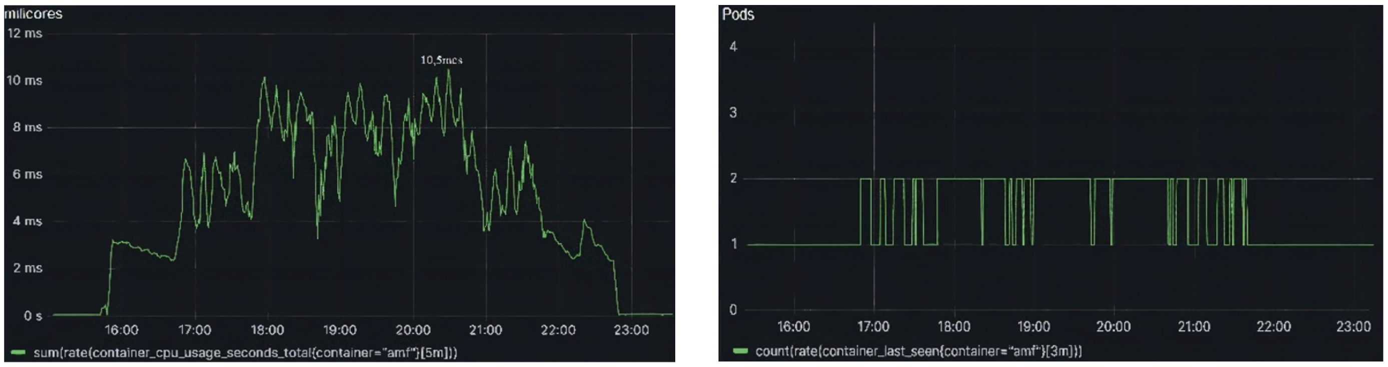


Figure 7. *Total CPU usage for reactive, predictive, and self-scaling policies scenarios, respectively, using My5G-RanTester.*

A. Analysis of the Reactive Scenario

Metric (i). Figure 8. (a) presents the total sum of CPU usage of the AMF pods over time. A peak of approximately 7 hours is observed. The 10000 access requests per second generate a variable CPU usage as access requests are generated over the validation interval. The highest CPU value is 10.5 mcs. Moreover, there is a trend at the peak, as increases in CPU are evident as time goes by. The curve is also cyclical, that is, fluctuations do not have a fixed pattern but occur in longer cycles. Irregularity is also observed, as there are outliers in the validation interval, the CPU drops down to a value of 3.3 mcs after being at 9 mcs in a short time and rises again to the same value almost instantly. Finally, it is not possible to evaluate seasonality because there is one single peak of CPU generated in the validation interval.

Metric (ii). Figure 8. (b) presents the total number of AMF pods in the validation interval. It is observed that, to supply the CPU usage, a maximum of two AMF pods are required, each one with 6 mcs of CPU. It is evident how the PodScaler in the reactive scenario scales-out and scales-in AMF pods as the CPU usage passes the threshold of 6 mcs. Given the fluctuations in CPU usage in a short time, the PodScaler can work in the same way, creating and removing AMF pods.



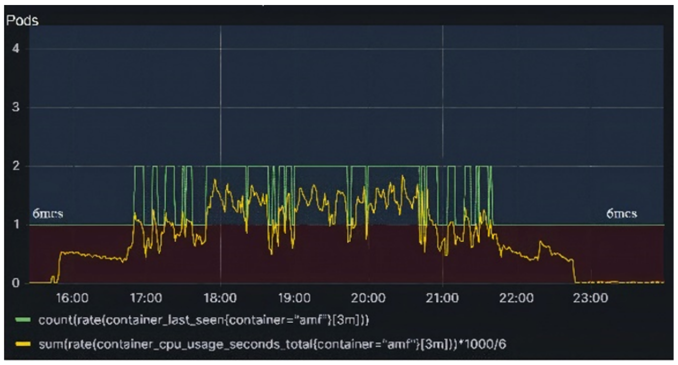
a. *b.*

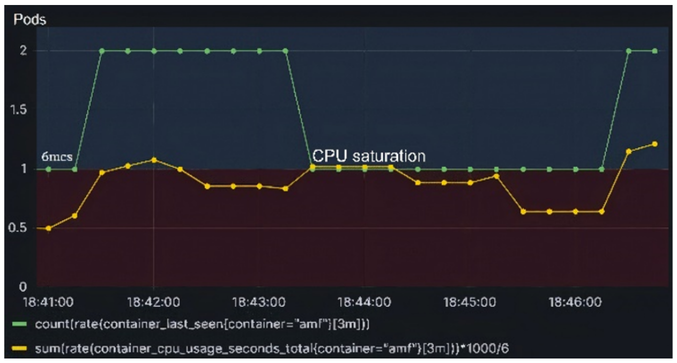
Figure 8. *(a) CPU Usage; (b) Number of AMF pods in the validation interval. Reactive scenario.*

Metric (iii). Figure 9. (a) presents the total CPU usage vs number of AMF pods over time. It is important to note that when CPU usage surpasses the threshold of 6 mcs, the PodScaler creates an additional pod to satisfy CPU needs, and vice versa, when CPU usage falls below the threshold, the PodScaler deletes a pod. Also, when a slight fluctuation in CPU usage occurs due to an increase in the access requests per second, the PodScaler does not generate an additional pod, thus resulting in a saturation in the AMF pod. This situation is observed in Figure 9. (b), which is frequently repeated when CPU usage fluctuates rapidly.

This CPU saturation in the AMF pod generates failed access requests since the AMF component is not operational to attend new incoming requests. Therefore, this experimental scenario is called reactive because when access requests per second increase, so does CPU usage, the PodScaler reacts by creating an additional pod. Likewise, when access requests per second decrease and CPU usage is below the threshold of 6 mcs, the PodScaler reacts by deleting an AMF pod. Consequently, a predictive scenario would solve the present problem, that is, the PodScaler has already created an additional pod when the increase in CPU usage above 6 mcs is about to happen.

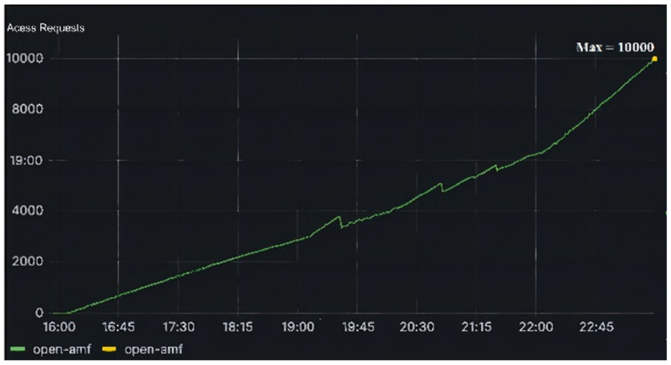
Metrics (iv) and (v): Successful and Failed access requests. For these metrics, it is necessary to calculate the success rate. Figure 10. (a) shows the total number of access requests generated in the validation interval, and Figure 10. (b) shows 8663 successful access requests. Therefore, the percentage of successful access requests is 86.63% (8663/10000). The remaining 13.37% is due to the CPU saturation experienced in one of the AMF pods.

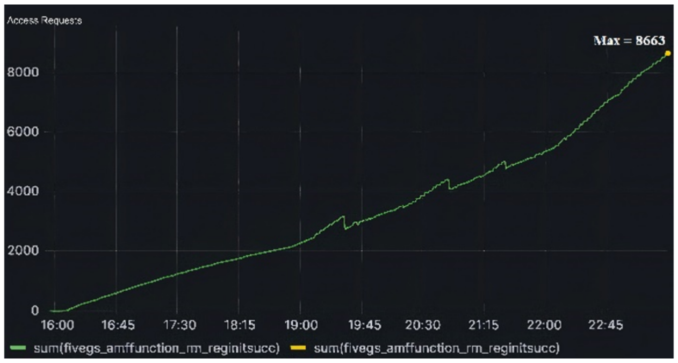




a. *b.*

Figure 9. *(a) Total CPU usage vs number of AMF pods with 6 mcs threshold; (b) CPU saturation in the AMF pod. Reactive scenario.*





a. *b.*

Figure 10. *(a). Total access requests generated; (b). Successful access requests. Reactive scenario.*

Metric (vi): Figure 11 shows the average total latency of successful access requests during the validation interval. It is evident that the maximum value of an access request was 316 ms, and the average latency is approximately 120 ms.

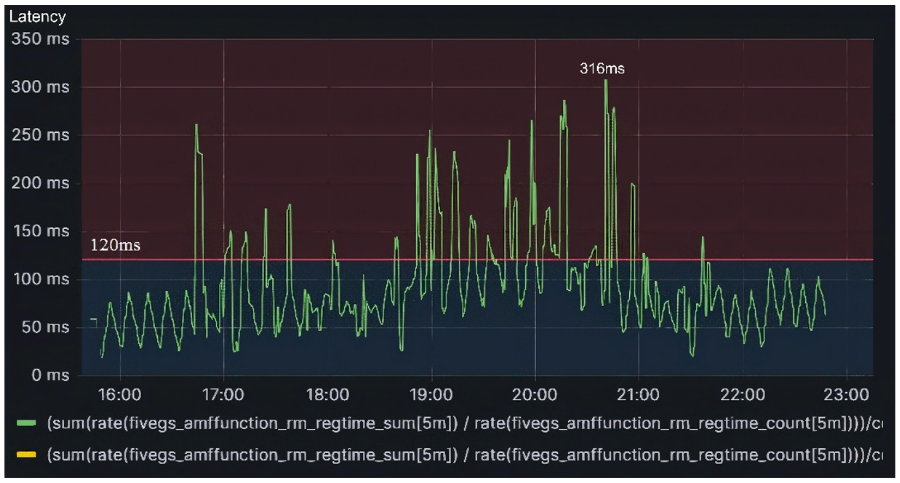


Figure 11. *Total average latency in successful access requests. Reactive scenario.*

B. Analysis of the Predictive Scenario

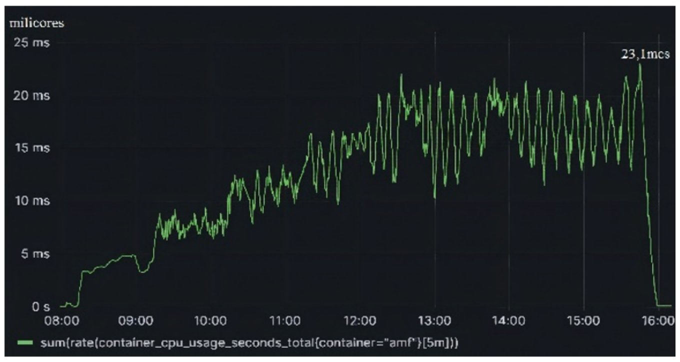
Metric (i). Figure 12. (a). presents the total CPU usage of the AMF pods over time. An increase in its measurement is observed for 8 hours. The 10000 access requests per second generate a variable CPU usage as access requests are generated over time. It is possible to note that the highest CPU value is 23.1 mcs. Moreover, there is a trend at the peak where gradual increases in CPU are evident in time. In addition, the curve is cyclical. Irregularity is also observed, as there are outliers in the validation interval, the CPU drops down to 10.8 mcs after being at 21.1 mcs in a short time and rises again to 21.3 mcs almost instantly. Finally, it is not possible to evaluate seasonality since there is only one peak of CPU generated as a validation interval.

Metric (ii): Figure 12. (b) presents the total number of AMF pods in the validation interval. Unlike the reactive scenario where only 2 pods are needed, in this predictive scenario a maximum of 6 AMF pods are required to supply CPU usage, each with 6 mcs. The rest of the analysis of this metric is the same as the previous scenario.

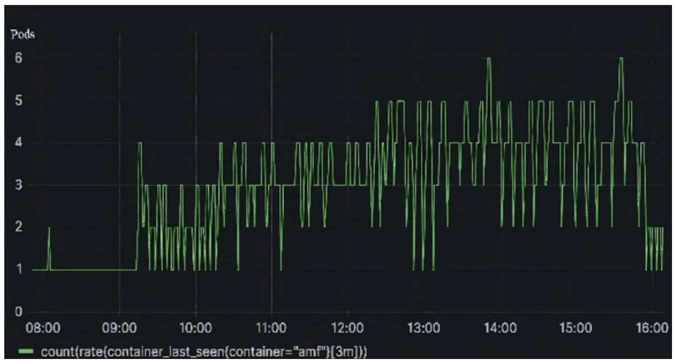
Metric (iii): Figure 13. (a) presents the total CPU usage vs the total number of AMF pods over time. It is important to note that when CPU usage surpasses the threshold of 6 mcs, the PodScaler creates an additional pod to meet the CPU needs, and vice versa, when CPU usage falls below the threshold, the PodScaler deletes a pod. However, when a slight fluctuation in CPU usage occurs due to a considerable increase in access requests per second, the PodScaler has previously created an additional pod to avoid a saturation in the immediately preceding AMF pod. This behavior is observed in Figure 13. (b).

It is evident that there are no CPU saturations in the AMF pods because the scenario is completely predictive. In this way, the pods are already scaled at the time of increasing access requests per second. Moreover, more pods are created to supply the CPU usage. Figure 18 presents a demarcation in pod number 4. This shows that 4 pods were enough to meet the needs of CPU usage, but the PodScaler configured with a predictive scenario created unnecessary additional pods, thus generating an excess. This is solved by combining the above prediction with scaling policies to avoid using unnecessary computational resources.

Metrics (iv) and (v): Successful and Failed access requests. For these metrics, it is necessary to calculate the success rate. Figure 14. (a) shows the total number of access requests generated in the validation interval, and Figure 14. (b) shows that 9712 successful access requests are submitted. Therefore, the percentage of successful access requests is 97.12% (9712/10000), which is better than the previous scenario.

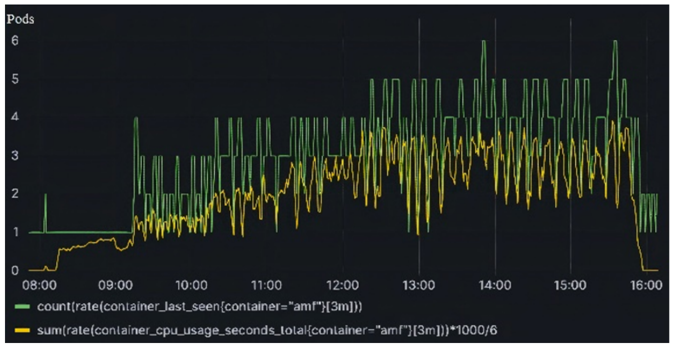


a.

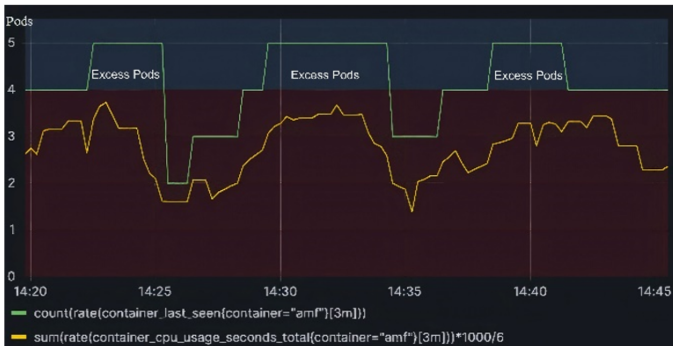


b.

Figure 12. *(a) CPU usage; (b) Number of AMF pods in the validation interval. Predictive scenario.*

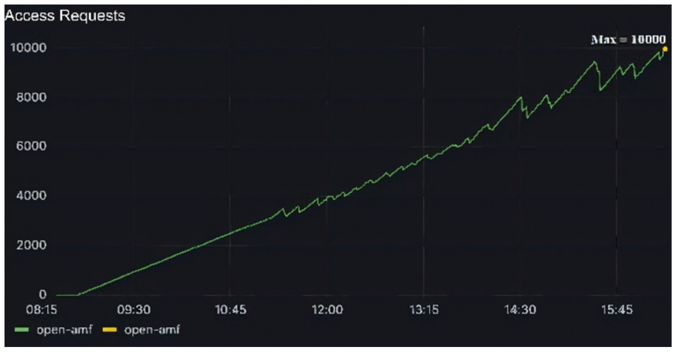


a.

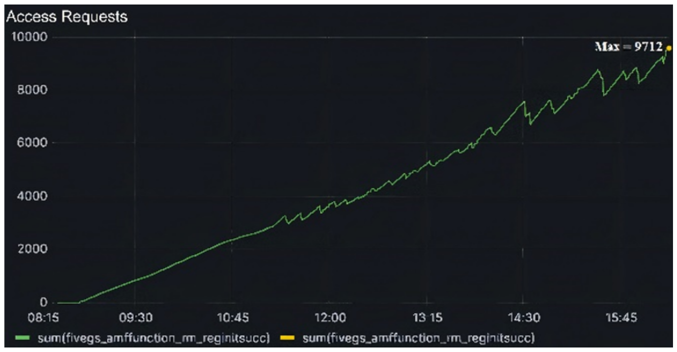


b.

Figure 13. *(a) Total CPU usage vs number of AMF pods with 6 mcs threshold; (b) Amount of AMF pods anticipate CPU usage. Predictive scenario.*



a.



b.

Figure 14. *(a) Total access requests generated; (b) Successful access requests. Predictive scenario.*

Then, failed access requests are 2.88%; a consequence of this scenario being completely predictive, which radically avoids the CPU saturation problem experienced by an AMF pod in the reactive scenario. An additional problem arises, the predictive scenario wastes computational resources by creating more pods than necessary to meet CPU needs.

Metric (vi). Figure 15 shows the total average latency of successful access requests during the validation interval. It is noted that the maximum value taken by an access request was 2.05 seconds, and that average latency is approximately 500 ms.

C. Analysis of the self-scaling policies scenario

Metric (i). In Figure 16. (a), the total CPU usage of the AMF pods over time presented an increase in its measurement for 10 hours. The 10000 access requests per second generate a variable CPU usage over time, where the highest CPU value is 23.1 mcs. Moreover, there is an exceptional behavior where gradual increases in CPU are evidenced over time. In addition, the curve is cyclical. Irregularity is also observed, as there are outliers in the validation interval. The CPU drops to 11 mcs after being at 18 mcs in a short time and rises again almost instantly. Also, it is not possible to evaluate seasonality since there is only one peak of CPU generated in the validation interval.

Metric (ii). Figure 16. (b) presents the total number of AMF pods in the validation interval. Unlike the previous scenarios, to meet the demand for CPU usage, a maximum of four AMF pods are required, each with 6 mcs. The rest of the analysis of this metric is the same as the previous scenarios.

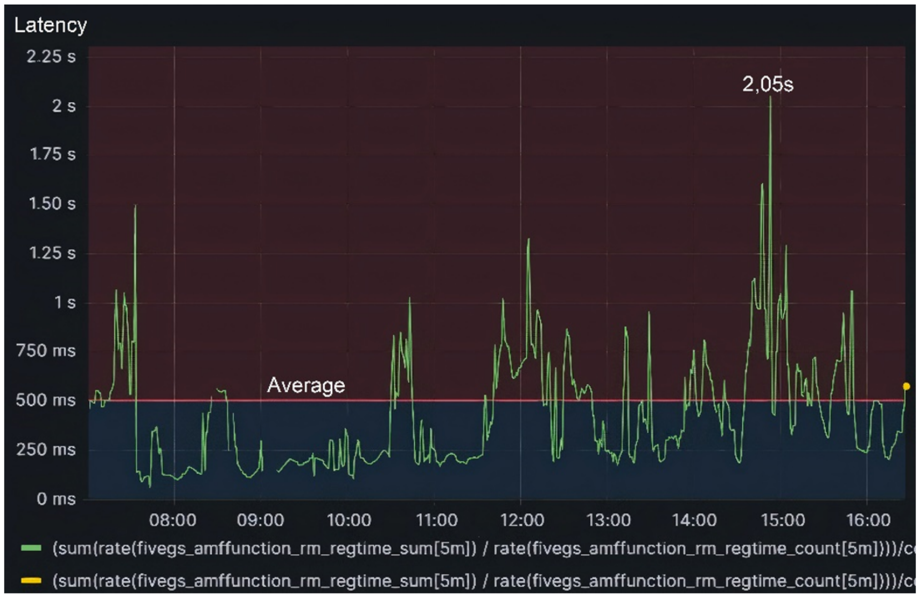
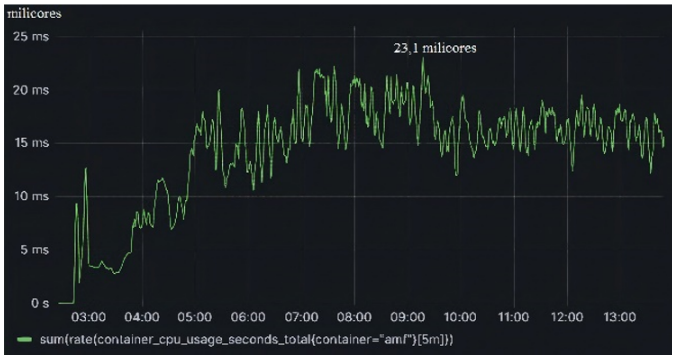
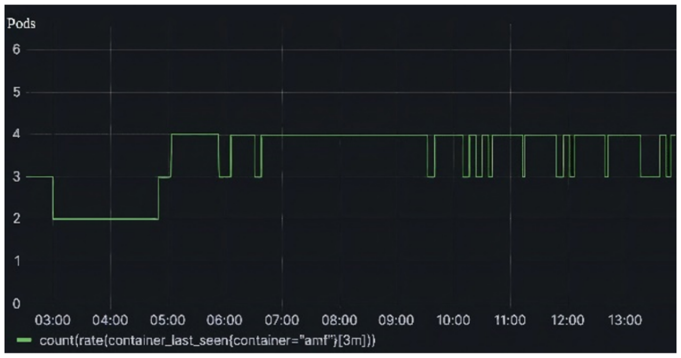


Figure 15. *Total average latency for successful access requests. Predictive scenario.*





a. *b.*

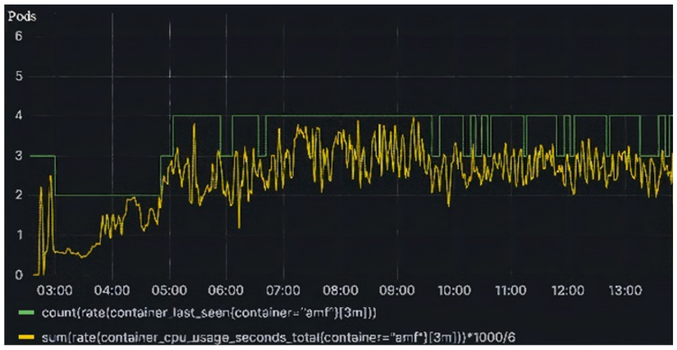
Figure 16. *(a) CPU usage; (b) Number of AMF pods in the validation interval. Self-scaling policies scenario.*

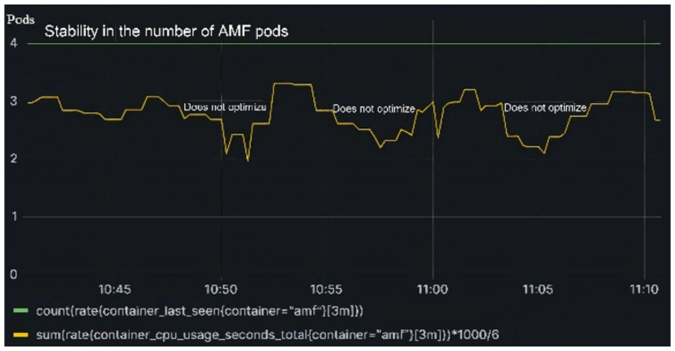
Metric (iii): Figure 17. (a) presents the total CPU usage vs number of AMF pods over time. It is observed that, during all CPU usage, the PodScaler does not create additional pods to supply the CPU usage; in other words, it does not waste computational resources by generating more pods than required. This situation is observed in Figure 17. (b). It is evident then that there are no CPU saturations in the AMF pods nor are computational resources wasted. This is because the scenario is predictive but combined with scaling policies that decide the increase or decrease of AMF pods.

Furthermore, pods are not decreased when the CPU is less than the threshold in time intervals close to 10 minutes. That is, CPU resources are not optimized at intervals of less than 10 minutes.

Metrics (iv) and (v): Successful and Failed access requests. For these metrics, it is necessary to calculate the success rate. Figure 18. (a), shows the total number of access requests generated in the validation interval, and Figure 18. (b) shows that 9420 successful access requests are submitted. Therefore, the percentage of successful access requests is 94.20% (9420/10000), and failed access requests are 5.8%. Scaling policies correct the over-resource problem previously seen in the predictive scenario as there is stability in the number of AMF pods.

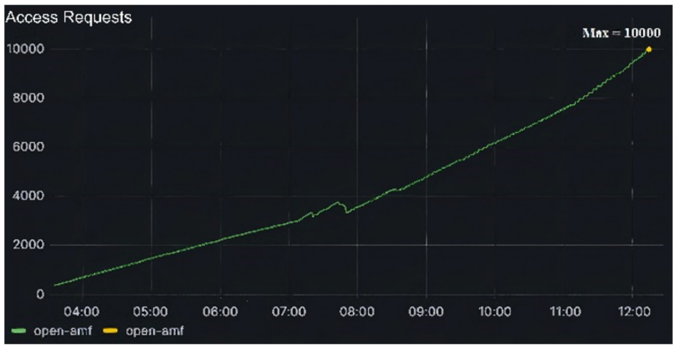
Metric (vi). Figure 19 shows the average total latency of successful access requests during the validation interval. It is noted that the maximum value taken by an access request was 1.28 seconds, and that on average the latency curve is approximately 200 ms. In other words, it takes an average of 0.2 seconds for an access request to be successful.

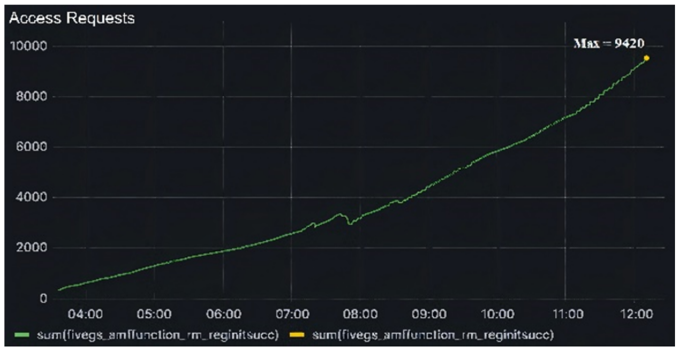




a. *b.*

Figure 17. *(a) Total CPU usage vs number of AMF pods with 6 mcs threshold; (b) Stability of the number of AMF pods in the face of variations in CPU usage. Self-scaling policies scenario.*





b.

Figure 18. *(a) Total access requests generated; (b) Successful access requests. Self-scaling policies scenario.*

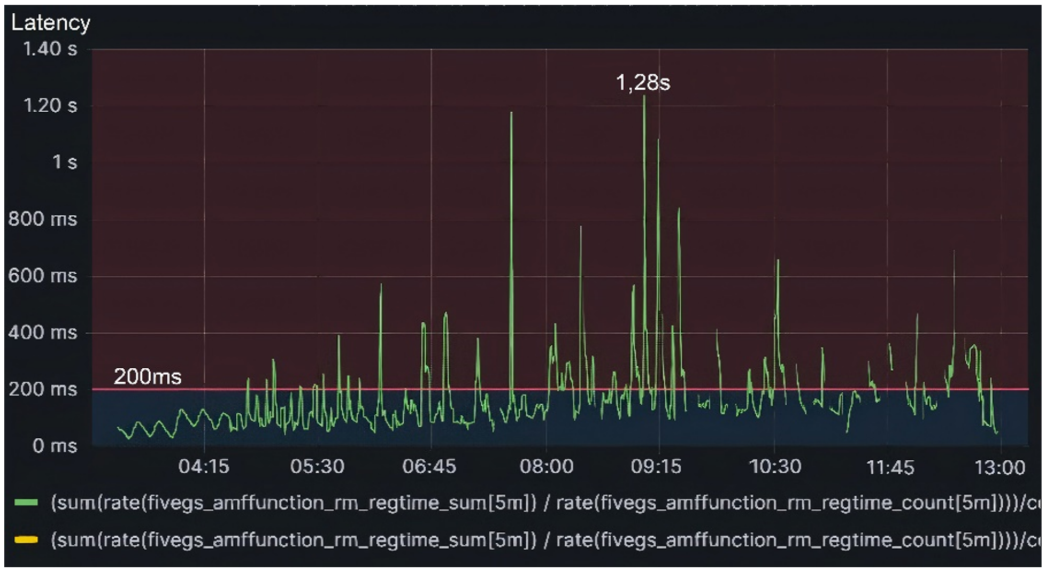


Figure 19. *Average total latency in successful access requests. Self-scaling policies scenario.*

4. DISCUSSION

To evaluate the performance of the AMF component in various contexts, experiments were performed in three different scenarios.

- Total CPU usage vs Total number of AMF pods over time. In the reactive scenario, CPU saturations are generated in the AMF pods, causing the unavailability of the 5G mobile network as the AMF component is not operational. In the predictive scenario, the CPU saturation problem is corrected, but it generates an excess of AMF pods that use computing resources inefficiently. In other words, it contributes to the availability of the 5G mobile network but compromises efficiency in the use of computing resources. Finally, in the predictive scenario with scaling policies, the problem of CPU saturation and excess of AMF pods is corrected; however, it does not optimize CPU resources in time intervals of less than 10 minutes. In other words, it contributes to the availability of the 5G mobile network without sacrificing computing resources, but there is no maximum efficiency.
- Success rate of 5G mobile network access requests. In the reactive scenario, the success rate is 86.63% —the lowest value of the three scenarios—due to the frequent saturations in the CPU usage by AMF pods. It leads to unavailability of the AMF because it is not operational during that saturation period. In the predictive scenario, the success rate is 97.12% —the highest value of the three scenarios— since the autoscaling mechanism creates AMF pods even when they are not needed. Therefore, this scenario uses the most computational resources inefficiently. In the predictive scenario with scaling policies, the success rate is 94.2%, which corresponds to an intermediate success rate. The stability in the number of AMF pods is maintained, preventing CPU saturation and computing resources from being wasted, it also provides the most balanced success rate, as there is no CPU saturation or excess of AMF pods.

- Failure rate of 5G mobile network access requests. In the reactive scenario, the failure rate is 13.37% —the highest loss rate of the three scenarios— because access requests are not managed or attended due to the unavailability of the AMF. In the predictive scenario, the failure rate is 2.88% —the least likely to decrease the number of AMF pods— which leads access requests to be lost. The predictive scenario with scaling policies has a failure percentage of 5.8% because there is no tendency to delete pods, i.e., scaling is done with policies that, depending on metric comparisons, influence the right scaling decision.
- Average latency of successful access requests over time. In the reactive scenario, the average latency value is 120 ms and its maximum is 316 ms. These latency values are the lowest of the three scenarios because there is not a prediction process. For the predictive scenario, the average latency value is 500 ms and its maximum is 2.05 seconds. It is the scenario with the highest average latency of a successful access request, and the one that generates the most AMF pods. When a pod is created, it takes a while for its state to be operational so that new access requests are attended. For the predictive scenario with scaling policies, the average latency value is 200 ms, and its maximum value is 1.28 seconds while maintaining the stability in the number of pods to meet CPU requirements. This confirms the balance of a self-scaling policies scenario.

5. CONCLUSIONS

A predictive autoscaling based on ML techniques is implemented to self-scale instances of the AMF component of an 5G network to predictively adapt to CPU variations. It contributes to an efficient use of computational resources by improving network availability. This approach, in addition to responding to current network demands, anticipates potential saturations or inefficiencies, thus allowing for proactive and efficient scaling of AMF pods in a Kubernetes environment. Therefore, these policies maximize the availability of the 5G mobile network and avoid the inefficient use of valuable computational resources.

The reactive scenario proved to be the option with the lowest success rate in access requests, reaching only 86.63%. Although it has the lowest latency, with an average value of 120 ms and a maximum of 316 ms, this is overshadowed by the constant CPU saturations reaching a maximum CPU of 10.5 mcs, which lead to the unavailability of the 5G mobile network. In consequence, a pure reactive approach is not enough to keep a 5G network available and efficient due to its tendency to react only to current problems without anticipating future demands.

The predictive scenario showed the best success rate, reaching 97.12%. However, this success implies a high cost in terms of computational resources, as it tends to oversize the number of AMF pods, reaching a maximum CPU value of 23.1 mcs. Although it guarantees 5G network availability, it does so at the expense of efficiency by sacrificing resources unnecessarily and with a notable increase in latency time with an average value of 500 ms and a maximum of 2.05 seconds.

The scenario that combines predictions of the LSTM model and scaling policies finds a balance between the two previous scenarios. It manages to maintain network stability without using computational resources inefficiently. Although it does not fully optimize CPU usage in short intervals of less than 10 minutes, since it reaches a maximum CPU value of 23.1 mcs, it represents a compromise between resource efficiency and network availability. Thus, the success rate of access requests to the 5G mobile network is 94.25% and the average latency value is 200 ms with a maximum of 1.28 seconds.

AUTHORS' CONTRIBUTIONS

Wilmar-Yesid Campo-Muñoz: Conceptualization Investigation, Formal analysis, Methodology, Supervision, Writing - review and editing. **Jhon-Alexander Amaya-Suárez:** Conceptualization, Data curation, Methodology, Resources, Software, Validation, Visualization, Writing - original draft. **Juan-Camilo Caviedes-Valencia:** Conceptualization, Investigation, Methodology, Validation, Writing - review and editing.

ACKNOWLEDGMENTS

To the University of Quindío for its support in the execution of this research through project 1160 of the *Vicerrectoría de Investigaciones*.

REFERENCES

[1] Ericsson, *Explore the Ericsson Mobility Report*, 2024. https://www.ericsson.com/en/reports-and-papers/mobility-report/reports/june-2024

[2] I. Alam et al., "A Survey of Network Virtualization Techniques for Internet of Things Using SDN and NFV," *ACM Computing Surveys*, vol. 53, no. 2, e337944, 2020. https://doi.org/10.1145/3379444

[3] J. L. Chavez-Picon, W. Y. Campo-Muñoz, G. E. Chanchí-Golondrino. "Arquitectura para implementación de servicios de video sobre redes móviles mediante redes definidas por software y segmentación de red," *Revista Colombiana de Tecnologías de Avanzada (RCTA)*, vol. 2, no. 42, e2651. https://doi.org/10.24054/rcta.v2i42.2651

[4] A. Chouman, D. M. Manias, A. Shami, "A Reliable AMF Scaling and Load Balancing Framework for 5G Core Networks," in International Wireless Communications and Mobile Computing (IWCMC), Marrakesh, Morocco, 2023, pp. 252-257. https://doi.org/10.1109/IWCMC58020.2023.10182447

[5] H. Atarashi, M. Iwamura, S. Nagata, T. Nakamura, A. Toskala, "5G Targets and Standardization," in *5G Technoly 3GPP Evolution to 5G-Advanced*, pp. 13-26, 2024.

[6] A. Toskala, M. Poikselkä, "5G Architecture," in *5G Technoly 3GPP Evolution to 5G-Advanced*, pp. 67-86, 2024.

[7] S. Christakis, N. Makris, T. Korakis, S. Fdida, "On the Automated Scaling of User Plane Function for 5G: An Experimental Evaluation," in *Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)*, Belgium, 2024, pp. 979-984. https://doi.org/10.1109/EuCNC/6GSummit60053.2024.10597110

[8] C. Rotter, T. Van Do, "A Queueing Model for Threshold-Based Scaling of UPF Instances in 5G Core," *IEEE Access*, vol. 9, pp. 81443-81453, 2021. https://doi.org/10.1109/ACCESS.2021.3085955

[9] T. Lorido-Botran, J. Miguel-Alonso, J. A. Lozano, "A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments," *Journal of Grid Computing*., vol. 12, no. 4, pp. 559-592, 2014. https://doi.org/10.1007/s10723-014-9314-7

[10] J. C. Valencia, "*Modelo predictivo para la asignación elástica de recursos sobre entornos NFV/SDN basados en OpenStack*," Grade Thesis, Universidad Nacional de Colombia, Bogotá, Colombia, 2021. https://repositorio.unal.edu.co/handle/unal/79636

[11] I. Alawe, A. Ksentini, Y. Hadjadj-Aoul, P. Bertin, "Improving Traffic Forecasting for 5G Core Network Scalability: A Machine Learning Approach," *IEEE Network*, vol. 32, no. 6, pp. 42-49, 2018. https://doi.org/10.1109/MNET.2018.1800104

[12] I. Alawe, A. Ksentini, Y. Hadjadj-Aoul, P. Bertin, A. Kerbellec, "On evaluating different trends for virtualized and SDN-ready mobile network," in *IEEE 6th International Conference in Cloud Networking*, 2017. https://doi.org/10.1109/CloudNet.2017.8071534

[13] M. A. Alam, S. Khatibi, "CPU resource usage analysis for downlink PDCP processing in CRAN," in *IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, 2020. https://doi.org/10.1109/PIMRC48278.2020.9217294

[14] H. D. Trinh, L. Giupponi, P. Dini, "Mobile Traffic Prediction from Raw Data Using LSTM Networks," *IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, 2018, pp. 1827-1832. https://doi.org/10.1109/PIMRC.2018.8581000

[15] L. Nashold, R. Krishnan, *Using LSTM and SARIMA Models to Forecast Cluster CPU Usage*, 2020. https://api.semanticscholar.org/CorpusID:219711129

[16] H. Ge, Y. Huo, Z. Wang, P. Xie, T. Wei, "VNF Instance Dynamic Scaling Strategy Based on LSTM," *Advances in Intelligent Systems and Computing*, vol. 1274, pp. 335-343, 2021. https://doi.org/10.1007/978-981-15-8462-6_39