

SESSION-STATE PAYLOAD SIZE AND AUTOSCALING BEHAVIOR OF MVC APPLICATIONS IN A KUBERNETES TESTBED

Meriton Ibraimi, Agon Memeti*

University of Tetova, Tetovo
North Macedonia

* Corresponding Author, e-mail: meritonibraimi01@gmail.com

Abstract: This paper evaluates the scaling behaviour of Model-View-Controller (MVC) applications in a Kubernetes testbed, focusing on external session-state payload size and autoscaling behaviour. A Redis-backed MVC deployment is tested under burst workload with three payload sizes, three replica counts, and three repetitions per condition. A second experiment validates Kubernetes Horizontal Pod Autoscaler (HPA) and Kubernetes Event-driven Autoscaling (KEDA) with Prometheus under a CPU-heavy burst workload. Results show that the 100 KB payload reduces throughput and increases p95 latency and failed-request rates, while additional web replicas do not remove the shared-state cost. Both HPA and KEDA produced real pod scaling. The findings show that MVC scalability depends on payload size, shared state-store capacity, workload shape, and scaling signal.

Key words: cloud-native applications, Kubernetes autoscaling, MVC applications, KEDA, session-state payload size.

1. INTRODUCTION

Cloud-native applications are commonly deployed as containerized workloads managed by orchestration platforms. Kubernetes supports deployment, service discovery, health management, and replica-based scaling, which makes it a common platform for web applications that must react to changes in demand [1, 2]. However, horizontal scaling is not only a platform function. It is also an architectural property of the application. If replicas are not interchangeable, if requests require sticky routing, or if a shared dependency becomes saturated, additional pods may not reduce latency or failed requests.

This issue is important for Model-View-Controller (MVC) applications because many traditional implementations keep user-session data close to the web process. In a stateful configuration, user requests must be routed repeatedly to the same pod or server instance. In a stateless web-layer configuration, session data is moved outside the web pod, for example to Redis, so that any replica can serve a request. The second approach

improves replica interchangeability, but it also introduces a shared state dependency whose communication cost may become visible during burst traffic [3, 4].

Recent autoscaling research has improved the scaling logic of cloud-native systems, but much of this work assumes that application replicas are already prepared for horizontal scaling. The survey in [5] shows that autoscaling has been studied across infrastructure, application architecture, scaling methods, objectives, and behavioural modelling. Other work improves Kubernetes scaling decisions through resource-efficient HPA designs [6] or proactive prediction of request rates [7]. Related work in cloud and virtualized environments also emphasizes resource utilization, workload redistribution, and service-level objectives [3, 4, 8]. Nevertheless, these studies do not fully isolate how MVC session-state size and external session-store dependency affect scaling outcomes.

The present paper therefore focuses on a narrower and more measurable question: how does external session-state payload size influence MVC scaling behaviour in Kubernetes, and can HPA and KEDA produce observable replica changes under burst workload? The paper is structured around two research questions:

RQ1: How do external session-state payload size and replica count affect throughput, p95 latency, and failed-request rate in a Redis-backed MVC deployment under burst workload?

RQ2: Under the selected workload, do Kubernetes HPA and KEDA with Prometheus produce real pod autoscaling behaviour, and how do their observed performance results compare?

The paper makes three contributions. First, it defines the scalability characteristics of MVC applications in terms of replica interchangeability, session-state externalization, payload size, scaling signal, and shared dependency capacity. Second, it presents a Kubernetes-based sensitivity experiment with 27 completed runs that varies external session payload size and replica count. Third, it validates real HPA and KEDA autoscaling behaviour through six additional Kubernetes runs and records pod counts, HPA states, ScaledObject configurations, and Kubernetes events. These additions directly strengthen the experimental contribution and reduce the risk of interpreting the work as a simple emulator.

2. RELATED WORK AND CRITICAL EVALUATION

The survey in [5] provides a broad taxonomy of autoscaling approaches for cloud-native applications. It classifies autoscaling according to infrastructure, application architecture, scaling method, objectives, and behavioural modelling. For the present work, the most relevant point is that autoscaling should not be evaluated independently from application architecture. A scaling controller may react correctly to a metric while the application still fails to benefit from added capacity because state management, routing, or a dependency limits the effect of new replicas.

A resource-efficient HPA design is proposed in [6]. The approach combines centralized and decentralized architectural ideas and aims to improve resource allocation for microservice applications, especially in resource-constrained environments. The contribution is important because it shows that CPU-based scaling alone is often

insufficient and that coordination across services can improve scaling behaviour. In this paper, the same general concern is translated to MVC workloads: scaling the web tier alone may not solve latency if session-state access becomes the bottleneck.

A proactive autoscaling design based on Prophet and Long Short-Term Memory models is presented in [7]. That work uses request-rate forecasting to estimate future pod demand before overload occurs. Request rate is especially relevant because it is closer to user demand than CPU utilization. However, prediction and multi-metric control still assume that additional replicas can effectively serve additional requests. In MVC systems with externalized session state, this assumption must be tested rather than taken for granted.

Resource-management problems are also discussed in multi-cloud and virtualized environments. Workload scheduling for resource utilization maximization is studied in [3], while service-level agreement-aware scheduling is discussed in [4]. Similar concerns about utilization and workload redistribution in virtual data centres are examined in [8]. These studies support the general view that performance depends on the relationship between workload, resources, and system architecture, not only on the existence of a scaling mechanism.

Kubernetes HPA is designed to automatically adjust the replica count of a workload based on specified metrics [11]. KEDA extends Kubernetes scaling by connecting event sources and external metrics to Kubernetes scaling decisions; the Prometheus scaler can use a Prometheus query and threshold as a trigger [12]. These mechanisms are useful for this study because they allow a direct comparison between CPU-based scaling and request-rate-based scaling in the same Kubernetes deployment.

Session management documentation further supports this focus. ASP.NET Core distributed caching documentation describes a distributed cache as an external service shared by multiple app servers and used to improve application performance and scalability in cloud or server-farm scenarios [13]. Redis documentation also describes Redis-backed session stores as a common way to share session state across multiple application instances [14]. These sources justify externalizing session state, but they also motivate measuring the cost of the external store rather than assuming that externalization is always beneficial.

The gap addressed in this paper is therefore specific: the literature contains many scaling policies, but less evidence is available on how MVC session-state externalization behaves when the amount of session data increases and multiple replicas share the same external state store. The evaluation tests this directly rather than only validating the expected statement that external state access introduces overhead.

3. SCALABILITY CHARACTERISTICS OF MVC APPLICATIONS

For an MVC application, scalability means more than increasing the number of running pods. A scalable MVC deployment should have at least five characteristics. First, replicas should be interchangeable, meaning that any pod can serve a request without breaking user-session continuity. Second, routing should not depend permanently on sticky sessions unless this dependency is accepted as part of the architecture. Third, the scaling signal should reflect the real pressure experienced by the

application, such as request rate, latency, or failed requests, not only CPU utilization. Fourth, external dependencies such as Redis or a database must have enough capacity to support the increased number of web replicas. Fifth, external session-state payload size should remain controlled because very large payloads can increase serialization, network transfer, and state-store access overhead.

This interpretation clarifies why an MVC application may fail to benefit from scaling. In-memory session storage can require sticky routing and reduce replica interchangeability. External session storage can remove that constraint, but it shifts part of the workload to the state store. Therefore, horizontal scalability depends on the balance between the benefit of more web pods and the cost of shared session access, especially when the external session payload becomes large.

4. METHODOLOGY AND EXPERIMENTAL DESIGN

The evaluation uses a local Kubernetes testbed instead of relying only on an emulator-style setup. The application is an ASP.NET Core MVC/API workload deployed as a Kubernetes Deployment. The web-layer variant evaluated in the new experiments uses Redis as an external session store. The testbed also includes a Kubernetes Service exposed through NodePort, Prometheus for application metrics, Metrics Server for CPU metrics, Kubernetes HPA, KEDA, and k6 as the workload generator. Source code and experimental artefacts are provided in [9, 10].

The tested endpoint is `/work/session?stateKb=x&cpuMs=y`. For each request, the application reads session data, updates a small counter, writes a session payload of the requested size, performs optional CPU work, and returns a JSON response containing the pod identity and measured endpoint information. This endpoint makes it possible to control two variables: the session-state payload size and the amount of CPU work per request. Figure 1 shows the Kubernetes testbed architecture used to evaluate external session-state payload size and autoscaling behaviour.

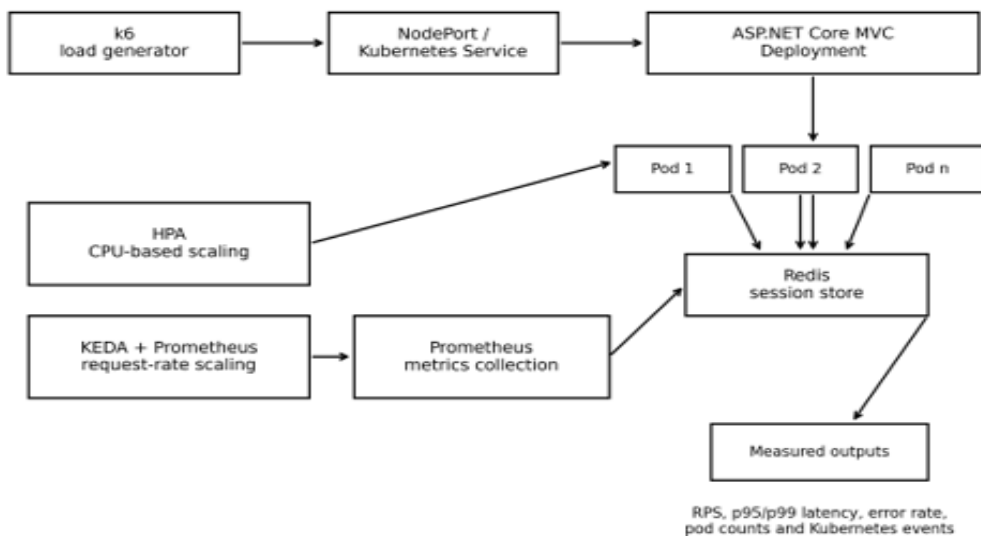


Figure 1. Kubernetes testbed for MVC session-state payload and autoscaling evaluation

Two experiments are performed. Experiment 1 is a session-state sensitivity experiment. It isolates the effect of external session-state payload size and replica count by using fixed Kubernetes replica counts. The replica count is manually set to 2, 4, and 6 so that the effect of the number of replicas can be separated from the dynamic behaviour of an autoscaler. Session-state sizes of 1 KB, 10 KB, and 100 KB are tested under the same burst workload, and each condition is repeated three times. This produces 27 completed runs.

Experiment 2 is an autoscaling validation experiment. It is designed to confirm that the work includes real Kubernetes autoscaling behaviour. A 10 KB session-state payload and CPU work of 80 ms are used to create sufficient load. The HPA configuration scales the deployment from a minimum of 2 replicas to a maximum of 6 replicas based on CPU utilization. The KEDA configuration uses Prometheus request-rate metrics and creates a scaling decision through a KEDA ScaledObject. Each policy is repeated three times. The purpose of this experiment is not to claim universal superiority of one autoscaler, but to verify whether real pod scaling occurs and to compare the observed behaviour under the selected workload. Table 1 summarizes the controlled factors, values, number of runs, and main purpose of the two experiments.

Table 1. Experimental design

Experiment	Controlled factors	Values	Runs	Main purpose
1	Session payload; replicas	1, 10, 100 KB; 2, 4, 6 replicas	27	Measure external session-state overhead
2	Autoscaling policy	HPA/CPU; KEDA/Prometheus	6	Validate real Kubernetes scaling behaviour

Table 2 presents the testbed configuration and execution parameters used to support reproducibility.

Table 2. Testbed configuration and execution parameters

Item	Configuration
Kubernetes testbed	Docker Desktop Kubernetes, local single-node cluster, namespace mvc-scale
Application	ASP.NET Core .NET 8 MVC/API workload deployed as Kubernetes Deployment
Session store	Redis 7 Kubernetes Service used for external session state
Service exposure	Kubernetes Service exposed through NodePort 30081
Metrics	Prometheus for request metrics; Metrics Server for CPU metrics
Workload generator	k6 ramping burst workload; three repetitions per condition
Experiment 1 settings	Fixed replicas 2, 4, 6; payload sizes 1, 10, 100 KB; CPU work 10ms
Experiment 2 settings	HPA min/max 2-6 replicas, CPU target 30%; KEDA Prometheus threshold 10 requests/s; payload 10 KB; CPU work 80ms

The measured metrics are requests per second (RPS), p95 latency, average latency, maximum latency, failed-request rate, and the number of completed HTTP requests. For Experiment 2, Kubernetes evidence is also recorded through pod states, HPA status,

ScaledObject configuration, and Kubernetes events before and after each run. Only completed k6 runs with a valid summary file are included in the analysis. Interrupted attempts are excluded from the reported results.

5. RESULTS

Table 3 presents the median results from Experiment 1. The table reports median RPS, median p95 latency, the observed p95 range, median failed-request rate, the observed failed-request range, and the median number of HTTP requests. The most important result is not a perfectly monotonic relationship across all payload sizes, but the clear degradation observed for very large external session payloads, especially at 100 KB.

Table 3. Session-state sensitivity results under burst workload

Rep.	State size (KB)	RPS median	p95 median (ms)	p95 range (ms)	Error median (%)	Error range (%)	Requests median
2	1	72.36	5905.96	2098.08-6102.97	0.797	0.192-0.827	21713
2	10	79.37	5500.55	1899.10-5910.64	0.298	0.173-2.117	23817
2	100	38.86	9504.50	393.29-32198.75	6.509	0.960-95.251	11662
4	1	56.74	5500.83	4309.18-26894.79	1.016	0.137-8.278	17022
4	10	83.45	4086.50	1997.82-4312.03	0.344	0.096-0.465	25036
4	100	34.02	12154.52	10104.27-24693.52	2.135	0.015-29.338	10211
6	1	70.35	5599.45	2599.55-7594.05	0.621	0.147-3.369	21110
6	10	89.99	2104.08	1801.17-4492.55	0.115	0.100-0.195	27005
6	100	35.86	12288.34	5602.37-27409.11	3.859	1.719-79.919	10760

For 2 replicas, the 100 KB state size reduced the median throughput to 38.86 RPS and increased the median p95 latency to 9504.50 ms, with a median failed-request rate of 6.509%. For 4 replicas, the 100 KB condition produced 34.02 RPS, a median p95 latency of 12154.52 ms, and a median failed-request rate of 2.135%. For 6 replicas, the 100 KB condition produced 35.86 RPS, a median p95 latency of 12288.34 ms, and a median failed-request rate of 3.859%. These values show that increasing the number of web replicas does not remove the effect of a large, shared session payload.

The 10 KB condition was the most stable in this testbed; with 6 replicas, it reached 89.99 RPS, 2104.08 ms p95 latency, and 0.115% failed requests. This is not a universal optimum, but in this environment moderate session payloads were handled better than large payloads. The high variability of the 100 KB condition suggests a sensitivity threshold where the external session store and network transfer become limiting factors

under burst load. Figure 2 visualizes the median p95 latency values from Table 3 by session-state size and replica count.

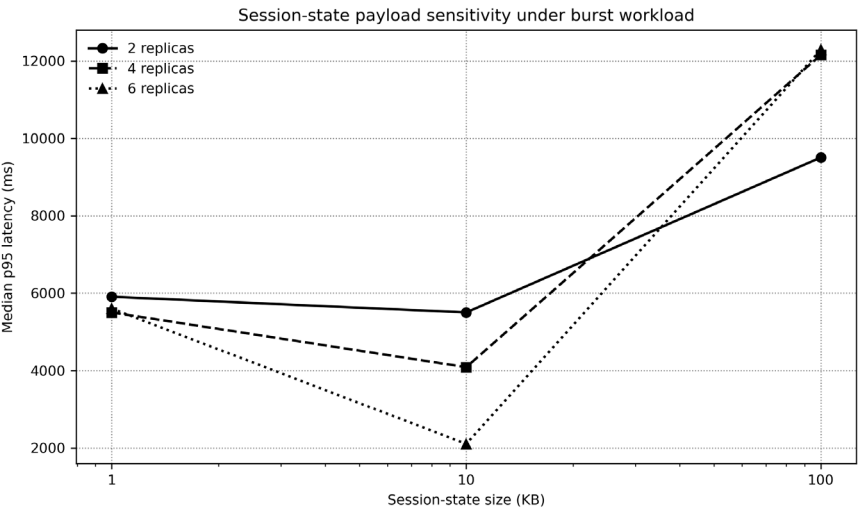


Figure 2. Median p95 latency by session-state size and replica count

To make the overhead easier to interpret, Table 4 normalizes the effect of increasing the external session payload from 10 KB to 100 KB. Across all replica counts, the 100 KB condition reduced median RPS by 51.0-60.2% and increased median p95 latency by 72.8-484.0%. The error-rate multiplier also increased, although the baseline error rate at 10 KB was low. This normalization shows that adding web replicas did not compensate for the shared-state cost created by the large payload.

Table 4. Normalized effect of increasing session payload from 10 KB to 100 KB

Replicas	RPS change	p95 latency change	Error-rate multiplier
2	-51.0%	+72.8%	21.8x
4	-59.2%	+197.4%	6.2x
6	-60.2%	+484.0%	33.6x

Table 5 summarizes Experiment 2. Both policies were executed in Kubernetes with a minimum of 2 replicas and a maximum of 6 replicas. The table reports median performance across three runs and the observed pod scaling range. Kubernetes events confirmed successful rescaling for both policies.

Table 5. Autoscaling validation results under CPU-heavy burst workload

Policy	Scaling trigger	Runs	Observed scaling	RPS median	p95 median (ms)	Error median (%)
HPA	CPU utilization	3	2-6 replicas	155.24	697.98	0.000
KEDA	Prometheus RPS	3	2-5/6	117.19	1110.48	0.002

Under the selected CPU-heavy workload, HPA reached a median throughput of 155.24 RPS, a median p95 latency of 697.98 ms, and a median failed-request rate of

0.000%. KEDA reached a median throughput of 117.19 RPS, a median p95 latency of 1110.48 ms, and a median failed-request rate of 0.002%. These results should not be interpreted as a general claim that HPA is always better than KEDA. The workload included CPU work specifically to ensure that HPA could react to CPU utilization. The important result for the present paper is that both HPA and KEDA produced observable Kubernetes scaling behaviour. HPA scaled from 2 to 6 replicas in all three runs, while KEDA scaled from 2 to 6 replicas in two runs and from 2 to 5 replicas in one run.

Kubernetes events recorded during the validation runs are important for interpreting the results. In the HPA runs, successful rescale events were associated with CPU utilization above the target. In the KEDA runs, the KEDA ScaledObject produced an HPA driven by the Prometheus metric. Therefore, the evaluation includes real Kubernetes autoscaling objects and pod rescaling, rather than only a manually controlled or emulator-style representation of scaling.

6. DISCUSSION

The results answer RQ1 by showing that external session-state overhead is not a constant value. It depends on payload size and on how many web replicas share the external session store. Very large payloads, especially 100 KB, produced lower RPS, higher p95 latency, and higher failed-request rates than the 10 KB payloads. This provides a more specific contribution than simply stating that external state access is slower. The experiment identifies external session payload size as a measurable scalability factor.

The results answer RQ2 by confirming that HPA and KEDA generated real Kubernetes scaling behaviour in the testbed. This is important because a cloud-native scaling claim should not rely only on a simplified local emulator. The validation uses Kubernetes Deployment objects, Services, pods, HPA, KEDA, Prometheus, and Kubernetes events. The experiment therefore demonstrates both application-level performance behaviour and platform-level scaling behaviour.

The practical implication is that MVC applications should not be declared scalable only because they run in containers or because an autoscaler is configured. Before scaling policies are compared, the application must be checked for replica interchangeability, session-state size, state-store capacity, and the relationship between the scaling metric and the actual bottleneck. In some cases, adding replicas may increase pressure on Redis or another shared dependency rather than reducing latency.

The results also show why the choice of scaling signal matters. HPA reacted well because the validation workload intentionally included CPU work, while KEDA reacted through the Prometheus request-rate metric. Different workload shapes, queries, or thresholds could change the relative results; therefore, policy effectiveness must be evaluated together with application architecture and workload characteristics.

7. THREATS TO VALIDITY

The first limitation is that the experiments were executed on a local Kubernetes testbed rather than a production multi-node cluster, which improves control but limits generalization. The second limitation is that the application is intentionally small and

focuses on session handling rather than a full business workflow. Production MVC systems may include additional database, cache, authentication, and network dependencies.

The third limitation is that Redis was used without extensive production tuning, so the results should be read as evidence of sensitivity to external session-state overhead rather than as a Redis benchmark. The fourth limitation is that each condition was repeated three times. Future work should use more repetitions, confidence intervals, additional workload shapes, latency-based metrics, and mixed CPU/request-rate policies.

8. CONCLUSION

This paper examined the scaling behaviour of MVC applications in a Kubernetes testbed with a focus on external session-state payload size and autoscaling. The evaluation shows that externalizing session state is not automatically sufficient for scalability. Very large external session payloads, especially 100 KB, reduced throughput and increased p95 latency and failed-request rates under burst workload. The experiment also showed that increasing the number of web replicas does not necessarily remove the cost of a shared state dependency.

A second validation experiment confirmed real Kubernetes autoscaling behaviour. HPA and KEDA both caused the deployment to scale from two replicas to five or six replicas during burst workload execution. Under the selected CPU-heavy workload, HPA showed lower median p95 latency than KEDA, but this should be interpreted as workload-specific rather than universal. The main conclusion is that MVC scalability depends on the interaction between application architecture, session-state size, external state-store capacity, workload shape, and autoscaling policy.

Future work should extend the testbed to a multi-node Kubernetes cluster, add database-backed scenarios, evaluate more session-state sizes, include latency-based KEDA triggers, and test irregular spikes and long-duration ramps.

REFERENCES

- [1] Zhong, Z., Xu, M., Rodriguez, M. A., Xu, C., Buyya, R. Machine learning-based orchestration of containers: A taxonomy and future directions. *ACM Computing Surveys*, vol.54, no.10s, 2022, Article 211. DOI: 10.1145/3510415.
- [2] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J. Borg, Omega, and Kubernetes. *ACM Queue*, vol.14, no.1, 2016, pp. 70-93. DOI: 10.1145/2898442.2898444.
- [3] Nelli, A., Jogdand, R. Dragonfly soft-computing approach for workload scheduling resource utilization maximization using multi-cloud platform. *International Journal on Information Technologies and Security*, vol.15, no.1, 2023, pp. 27-38. DOI: 10.59035/MBUL3714.
- [4] Jambigi, M. V., Vijay Kumar, M. V., Ashoka, D. V., Prabha, R. Service level agreement aware energy optimized scheduling algorithm for cloud computing environment. *International Journal on Information Technologies and Security*, vol.14, no.1, 2022, pp.17-28.

- [5] Xu, M., Wen, L., Liao, J., Wu, H., Ye, K. Auto-scaling approaches for cloud-native applications: A survey and taxonomy. arXiv:2507.17128v1, 23 Jul 2025. URL: <https://arxiv.org/abs/2507.17128> (Visited on 16.05.2026).
- [6] Ahmad, H., Treude, C., Wagner, M., Szabo, C. Smart HPA: A resource-efficient horizontal pod auto-scaler for microservice architectures. *2024 IEEE 21st International Conference on Software Architecture (ICSA)*, Hyderabad, India, June 2024, pp. 46-57. DOI: 10.1109/ICSA59870.2024.00013.
- [7] Guruge, P. B., Priyadarshana, Y. Time series forecasting-based Kubernetes autoscaling using Facebook Prophet and Long Short-Term Memory. *Frontiers in Computer Science*, vol.7, 2025. DOI: 10.3389/fcom.2025.1509165.
- [8] Kravets, O. Ja., Mutina, E. I., Zaslavskaya, O. Yu., Redkin, Yu. V., Rahman, P. A., Aksenov, I. A., Kamil, K. W. A. K. Improving the efficiency of using the resources of virtual data centers with a heterogeneous structure by repositioning virtual machines. *International Journal on Information Technologies and Security*, vol.17, no.1, 2025, pp. 3-12. DOI: 10.59035/MMYV5441.
- [9] Ibraimi, M. MVC scale test source code repository. GitHub, 2026. URL: <https://github.com/meriton01/mvc-scale-test> (Visited on 26.06.2026).
- [10] Ibraimi, M. Experimental result files for MVC autoscaling evaluation. Google Drive, 2026. URL: <https://drive.google.com/drive/folders/12z0YYp3O5pMLtTAdi2lSJ3KzhzA6W1Or?usp=sharing> (Visited on 16.05.2026).
- [11] Kubernetes Documentation. Horizontal Pod Autoscaling. URL: <https://kubernetes.io/docs/concepts/workloads/autoscaling/horizontal-pod-autoscale/> (Visited on 28.06.2026).
- [12] KEDA Documentation. Prometheus scaler. URL: <https://keda.sh/docs/2.20/scalers/prometheus/> (Visited on 28.06.2026).
- [13] Microsoft Documentation. Distributed caching in ASP.NET Core. URL: <https://learn.microsoft.com/en-us/aspnet/core/performance/caching/distributed> (Visited on 28.06.2026).
- [14] Redis Documentation. Redis session store with C#/.NET. URL: <https://redis.io/docs/latest/develop/use-cases/session-store/dotnet/> (Visited on 28.06.2026).

Information about the authors:

Meriton Ibraimi - master's student, University of Tetova, North Macedonia; interests: cloud-native applications, Kubernetes autoscaling and digital transformation.

Agon Memeti - Professor, University of Tetova, North Macedonia; interests: web technologies, distributed computing and information systems.

Manuscript received on 17 May 2026; revised on 28 June 2026