SILESIAN UNIVERSITY OF TECHNOLOGY PUBLISHING HOUSE

SCIENTIFIC PAPERS OF SILESIAN UNIVERSITY OF TECHNOLOGY 2023
ORGANIZATION AND MANAGEMENT SERIES NO. 186

DUAL INGRESS ARCHITECTURE DESIGN PATTERN FOR KUBERNETES APPLICATIONS

Piotr P. JÓŹWIAK

Wroclaw University of Science and Technology; piotr.jozwiak@pwr.edu.pl, ORCID: 0000-0002-5325-3728

Purpose: The article focuses on the analysis of the mechanism for exposing Services running on a Kubernetes cluster using an Ingress type definition. It discusses the basics of this mechanism, pointing out its fundamental limitation of being able to use only single technology simultaneously in handling traffic to a web application. The paper presents an architectural pattern that enables the simultaneous integration of two Ingress definitions, combining the advantages of both systems used.
Design/methodology/approach: Available solutions for exposing applications served in the Kubernetes cluster were analyzed. As a result of the research, an enhancement was proposed to allow the use of two services simultaneously, providing broader system functionality.
Findings: An approach was proposed to use two Ingress controllers simultaneously in the form of an external cloud service and an internal Nginx service running on a Kubernetes cluster.
Originality/value: A design pattern is presented along with an example implementation of dual Ingress on an AKS cluster **in Azure.**

Keywords: Kubernetes, Ingress, architectural pattern, limitations, Azure, Application Gateway, Nginx.

Category of the paper: Research paper**,** Technical paper.

1. Introduction

The Kubernetes cluster is becoming a mainstream global technology, according to a survey conducted by the *Cloud Native Computing Foundation* (CNCF). As many as 96% of the organizations surveyed indicated that they are using Kubernetes or are in the process of evaluating its capabilities (CNCF Annual survey, 2021). Of these, more than a quarter of respondents indicated that they are using Kubernetes as a cloud service provided by major cloud operators in the global market.

The growing popularity is linked to the provision of solutions that allow for an easily scalable environment compared to applications running on virtual machines. Kubernetes is gaining in proportion to the increasing popularity of application containerization. It keeps code operational and speeds up the delivery process. The Kubernetes API allows automating a lot of resource management and provisioning tasks. According to IBM, the most important factors influencing the choice of Kubernetes are (Top 7 Benefits of Kubernetes, 2022):

- Container orchestration savings,
- Increased DevOps efficiency,
- Deploying workloads in multicloud environments,
- More portability with less chance of vendor lock-in,
- Automation of deployment and scalability,
- App stability and availability in a cloud environment,
- Open-source benefits of Kubernetes.

In this paper, I focus on presenting the problem of making an application running in a Kubernetes cluster accessible to an external environment using the *Ingress* mechanism. Kubernetes *Ingress* is the basic tool that defines access to an application from the outside. Production use of this mechanism requires support in the form of external resources, usually provided by a cloud operator. In this paper, I outline what the basics of *Ingress* are and its limitations. Additionally, I present a way to circumvent the limitation of single *Ingress* per Kubernetes *Service*, allowing two Ingress mechanisms to be used simultaneously for a designated application service running on a cluster.

2. Kubernetes Ingress basics

An *Ingress* is a native Kubernetes object that defines external access to a Service running on a cluster (Burns et al., 2022). The *Service* object itself groups multiple *Pods* under one common type. The task of the *Ingress* manifest is to define a set of rules that govern inbound connection mapping between *Services*. This mechanism consolidates the routing rule to *Services* into a single resource. This routing is based on layer seven of the ISO/OSI model. Without the *Ingress* mechanism, each *Service* to be accessed outside the cluster would require to use separate definitions of e.g. *LoadBalancers* or *NodePorts*. *LoadBalancer* and *NodePort* exposes a service by specifying that value in the service's type. This limitation is particularly challenging for applications designed in microservices architecture. This definition makes it impossible to expose the entire application under one common URL because each new *LoadBalancer* will receive a separate IP address.

Ingress, on the other hand, is a completely independent resource to your *Service*. This makes it decoupled and isolated from the *Services* you want to expose (Burns et al., 2022; Palmer, 2023). An *Ingress* is used when we have multiple *Services* and we want the outbound requests routed to the *Service* based on URL path. Consider an example with two *Services*, *S1* and *S2* in a cluster. Then, for URL myservice.com/s1 we want to route to the *S1 Service* and accordingly for URL myservice.com/s2 we want to expose *Pods* served from *S2 Service*. These routings will be performed by an *Ingress*. Unlike *NodePort* or *LoadBalancer*, *Ingress* is not actually a type of *Service*. Instead, it is an entry point that sits in front of multiple services in the cluster. Figure 1 shows a diagram of how *Ingress* works showing the links between Kubernetes entities. In the example shown, *Service S1* groups two *Pods* of a single *Web1* web application. This way, traffic can be balanced between multiple *Pods*.

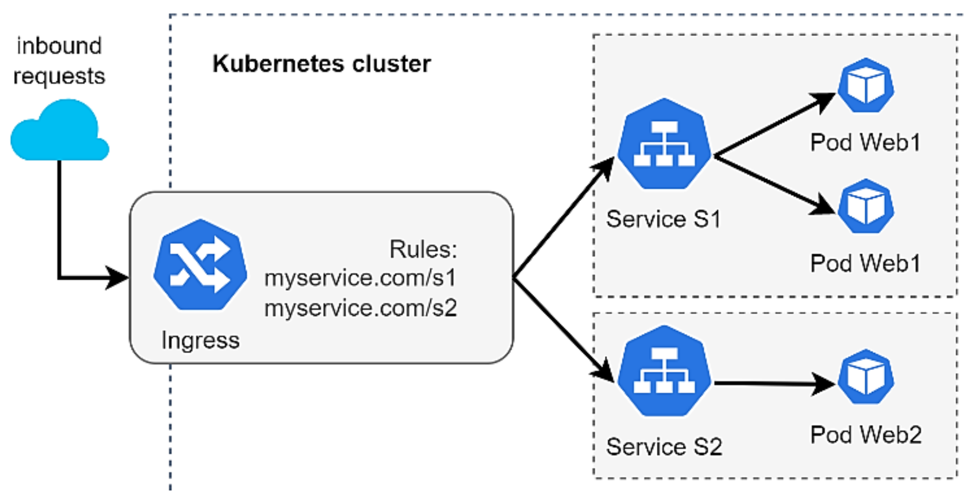

Figure 1. Diagram showing the links between Kubernetes Ingress, Services and Pods.

To use the above mechanism, it must be supported by the environment on which Kubernetes is running. This is because, to function, *Ingress* needs to access the network interface on which external traffic will be handled. By default, the cluster does not have this access, as this is resolved differently in each environment/cloud. Thus, it is the duty of Kubernetes administrators to provide and configure the appropriate mechanism. It is done by installing the appropriate *Ingress Controller*. The most popular *Ingress Controller* is the *Nginx Ingress* (Nginx docs, 2023). This is evident from the direct support by major cloud providers such as AWS, Azure and Google Cloud. The necessary support that cloud operators provide is related to the implementation of specific *Ingress Controllers* that tie the *Nginx* server to external services available in each cloud. In the case of AWS EKS, *Nginx* works alongside the *AWS Load Balancer* (Provide externall access…, 2023). In the case of Microsoft Azure AKS, the native service called *Azure Load Balancer* is used (Microsoft learn: Create an ingress, 2023). Correspondingly, Google Cloud integrates the *Google Cloud L4 Load Balancer* with *Nginx* (Ingress with Nginx, 2023). As can be seen, each of the indicated cloud operators by default uses its own service based on load balancer functionality to integrate with Nginx-based Ingress. This limitation will be further elaborated in the next section.

3. Motivation for research

However, the solution described above has a drawback. When using *Nginx Ingress*, we bind ourselves to a specific physical implementation that the cloud operator uses to integrate with cluster. Typically, it is some kind of a layer four based load balancer. In my example, I will focus on the solution offered by Microsoft Azure. *Load Balancer* provided by Azure is not the only service that can be used here. Other services that Azure offers are *Traffic Manager*, *Front Door* or *Application Gateway*. Each of these services has its own specialized application. The *Load Balancer* itself in Azure operates at layer four of the ISO/OSI model. In Azure, we also find a more tailored service for the requirements of web application/RESTful traffic, which is the *Application Gateway* (AppGw) (Microsoft learn: What is Azure Application Gateway, 2023). *AppGw* has the advantage of working on layer seven of the protocol. However, its advantages do not end there, as *AppGw* in Azure can be extended with a few additional functionalities, such as *Web Application Firewall* (WAF), autoscaling, high availability, URL-based routing, SSL encryption, SSL termination, Cookie-based affinity. With the above in mind, Microsoft Azure provides an *Ingress Controller* for Kubernetes clusters that directly uses *AppGw*. This service is called *AGIC* and is a direct alternative to *Nginx+Load Balancer*. The question here is which solution to use? Officially, you have to decide on one of these services, so when designing the system architecture, the designer has to decide between *Ingress* based on *Nginx+Azure Load Balancer* or *Ingress* based on *Application Gateway*.

Both solutions have many important functionalities. Some of these are available in both solutions. However, some functionalities are only available in one of the solutions exclusively. An example of this is the WAF available in *AGIC*, which is not available in the pure *Nginx Ingress Controller* solution. The WAF functionality is highly desirable for applications with strong IT system security requirements. A web application firewall is highly effective for detecting or preventing web attacks, leveraging the OWASP ModeSecurity Core Rule Set. For example, it can protect web applications from cross-site scripting and SQL injection attacks. On the other hand, *Nginx* provides many functionalities that are not directly available in *AGIC*. Using *Nginx* as an *Ingress Controller* allows you to take advantage of Single Sign-on (SSO) or provides a powerful mechanism for dynamic reconfigurations possible directly from Kubernetes manifests.

With the above in mind, there are situations where the architectural design would indicate the need to use both solutions simultaneously. Such a configuration offers the possibility to implement richer functionality by combining the features of both solutions. However, there is no documented method to achieve this. There is no such *Ingress* manifest definition that could integrate both solutions into a common functionality. In the following section, I present the architecture design pattern with its example implementation that combines both solutions. The key task of the discussed solution is to enable control of the entire system via Kubernetes manifests. Additionally, discussed architecture proposal gains a more elaborate model for the division of responsibilities between the DevOps and Security teams which is discussed in the next chapter.

4. Architecture pattern proposition for Dual Ingress Controller

To address the need to use both *Ingress Controllers* simultaneously, I present a two-tier *Ingress* architecture for a service running on an Azure AKS cluster. The simultaneous use of the *Application Gateway* and the *Nginx Ingress Controller* provides broader functionality and greater flexibility over standard solutions. Figure 2 shows an architectural diagram of the presented system.

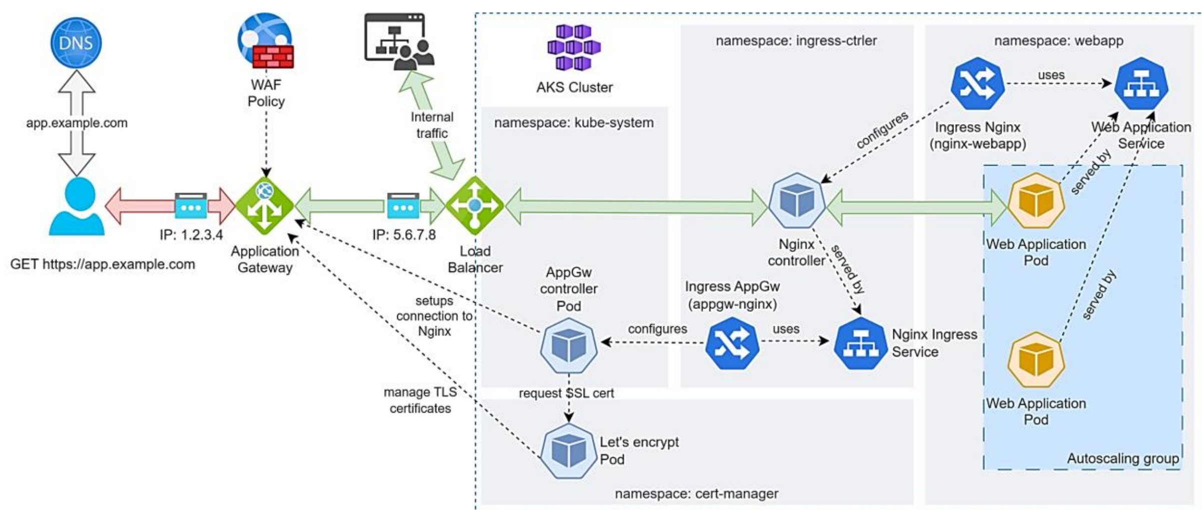

Figure 2. Diagram presenting Dual Ingress architectural pattern. Proposed design uses Application Gateway as an entry point for application and simultaneously Nginx Ingress controller. Diagram describes relations between all entities in presented pattern.

The main functional objective of the system is to run any web or RESTful system on the Kubernetes cluster. In the diagram from figure 2, the target application is installed in the namespace webapp. It consists of two *Pods* and a standard Kubernetes *Service* definition pointing to the application. The described scenario is the standard way how applications are deployed and served from Kubernetes cluster. An example of such a definition is presented in table 1.

The detailed routing to the above application, which will take place on a URL basis, is implemented on the *Nginx Controller*. For this purpose, the controller must be installed on a cluster. In the case of Azure AKS, the controller installation procedure described in (Microsoft learn: Create an ingress controller in AKS, 2023) can be used. In presented example, the Nginx controller is installed in the namespace ingress-ctrler as shown in the diagram in figure 2. The Nginx controller installation consists of a *Pod* on which the Nginx server is running and its own *Service* object. This *Service* directly integrates with the Azure *Load Balancer*. Thus, any network traffic passed to the *Load Balancer* input is effectively passed to the *Pod* with the *Nginx* server configured as reverse proxy. The second very important role of the aforementioned *Pod* is to observe the *Ingress* objects created on the cluster, which specify the ingress class on nginx in the spec.ingressClassName field of the manifest. An example of an *Ingress* definition linking the web application service to the *Nginx* controller is presented in table 2. The purpose of this manifest is to specify the URL path under which the web application is to be accessed externally.

Table 1.
Example Kubernetes manifests for application deployment. On left an deployment for example web application, on right Service definition

apiVersion: apps/v1 kind: Deployment metadata: name: webapp-hw namespace: webapp spec: replicas: 2 selector: matchLabels: app: webapp-hw template: metadata: labels: app: webapp-hw spec: containers: - name: webapp-hw image: mcr.microsoft.com/azuredocs/aks-helloworld:v1 ports: - containerPort: 80 env: - name: TITLE value: "Example WebService"	apiVersion: v1 kind: Service metadata: name: webapp-hw namespace: webapp spec: type: ClusterIP ports: - port: 80 selector: app: webapp-hw

Source: own work.

The next step required to implement the Dual Ingress architecture is installing the *AppGw* controller (AGIC). The *AGIC* controller itself is installed in the Kubernetes cluster as a corresponding AKS extension. It is an internal Azure mechanism that installs according to the instructions available in (Microsoft learn: Creating an ingress controller wit new Application Gateway, 2023), either as an Add-On or a Helm package. The execution of this instruction provides the necessary *AGIC Pod* to the namespace of the kubernetes-system and creates a physical *Application Gateway* in Azure. The installed *AGIC Pod* has a similar role to the *Nginx* controller *Pod*. The task of this *Pod* is to observer the *Ingress* manifests for definitions that indicate the ingress class as azure/application-gateway in the kubernetes.io/ingress.class annotation.

To perform the integration of the two *Ingress* controllers, we need to link the two *Services* to each other in an appropriate manner. Unfortunately, it is not possible to indicate in the *Ingress* definition prepared for the *AGIC* to redirect to another *Ingress*, in our case to the *Nginx Ingress Controller*. The main limitation of *Ingress* definitions is that only objects of type *Service* can be exposed as the target object. This is the primary reason for the lack of solutions that present the possibility of using both mechanisms simultaneously.

Table 2.
Nginx based Ingress manifest exposing Service webapp-hw on URL app.example.com

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: nginx-webapp
 namespace: webapp
 annotations:
 nginx.ingress.kubernetes.io/ssl-redirect: "false"
 nginx.ingress.kubernetes.io/use-regex: "true"
 nginx.ingress.kubernetes.io/rewrite-target: /$1
spec:
 ingressClassName: nginx
 rules:
 - host: app.example.com
 http:
 paths:
 - path: /(.*)
 pathType: Prefix
 backend:
 service:
 name: webapp-hw
 port:
 number: 80
```

Source: own work.

Table 3.
Kubernetes Application Gateway based Ingress manifest exposing Nginx controller Service on URL app.example.com. Ingress definition also contains TLS section for encryption and certificate provisioning by Let's Encrypt

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: appgw-nginx
 namespace: ingress-ctrler
 annotations:
 kubernetes.io/ingress.class: azure/application-gateway
 cert-manager.io/cluster-issuer: letsencrypt-appgw-http
spec:
 tls:
 - hosts:
 - app.example.com
 secretName: appgw-cert-secrets
 rules:
 - host: app.example.com
 http:
 paths:
 - path: /*
 pathType: Prefix
 backend:
 service:
 name: nginx-ingress-controller-svc
 port:
 number: 80
```

Source: own work.

However, there is a way around this problem and integrate *AGIC* and *Nginx* into a two-tier ingress architecture. To do this, we can take advantage of the fact that the *Nginx Controller* also has its own Kubernetes *Service* object. It is defined in the ingress-ctrler namespace as a mechanism to bind the *Azure Load Balancer* to *Nginix Pod*. In the solution discussed here, I use this definition directly to bind the *AGIC Ingress* to the *Service* of Nginx controller. Such a configuration is shown in the diagram in figure 2. An example implementation of *Ingress* pointing to *Nginx* is shown in table 3. It is important that the *Ingress* definition for the *AGIC* is defined in the ingress-ctrler namespace. This is necessary because the *Ingress* itself can only point to the target *Service* from the same namespace. This does not cause major complications, as *Ingress* controllers are implemented in such a way that they observe the corresponding manifests in any namespace. Using such a solution makes it possible to link the two mechanisms into a chain of two *Ingress Controllers* running one after the other. The *AGIC* acts as a direct external interface where network traffic goes at first to the application. This traffic is then redirected to the internal *Load Balancer* via the *AGIC Ingress*, which points to the *Nginx Controller* as the next step in the network traffic path. The *Nginx* server redirects the traffic in the second step directly to the *Pod* of the target application that handles the user's request. Thus, we have a two-tier architecture for handling network traffic, connecting all the mechanisms available in both *Ingress* controllers.

There are many advantages of this solution, the most important of which I outline below.

4.1. Web Application Firewall as SaaS

The first advantage is the ability to use the *Web Application Firewall* (WAF), which is provided as a SaaS service by Azure. This is a very sophisticated system that enhances the security of a web-based system, with the aim of detecting and responding to anomalies in network traffic. It is important to note that the system administrator does not need to be a high-level expert in this area, as *WAF* uses the OWASP ModeSecurity Core Rule Set providing recommendations available for use. The separation of the *WAF* functionality as a SaaS service available outside the cluster also allows for easier separation of duties for the DevOps and Security teams. The Security team defines the necessary security definitions directly on *AppGw* and *WAF*, without the need to interfere with the Web application itself or the Kubernetes cluster. On the other hand, the DevOps team uses the *Nginx* service to implement the functional rules. In this situation, access to the security rules defined on *AppGw+WAF* may not be available to the DevOps team, as they will deploy their functional rules on Nginx instead. This increases the level of security by narrowing access to defined rules only by Security team.

4.2. SSL offload

Another important asset of the proposed architecture is SSL offloading. SSL offloading is the process of removing the SSL-based encryption from incoming traffic that a web server receives to relieve it from decryption of data. The entire encryption effort has been moved off the cluster to *AppGw*. Network traffic has been secured to the first device, while in many situations internal network connections do not need to be encrypted. This relieves the end devices of additional power requirements, thereby increasing the throughput of the solution and reducing costs.

4.3. Automated TLS certificate renewal

Using HTTPS connection encryption requires obtaining a certificate with which the connection will be encrypted. Since in the solution presented here, all encryption handling has been moved to *AppGw*, we can use the *cert-manager* that works with this service. The *cert-manager* provided by Let's Encrypt can integrate with Azure *AppGw* by installing the corresponding controller on the Kubernetes cluster. For this purpose, the Let's Encrypt *Pod* was installed in the namespace of cert-manager according to the documentation available in (Microsoft learn: Use TLS…, 2023). The *cert-manager* mechanism observes the *Ingress* definitions created for *AppGw* and, when a definition is detected that indicates the need to enable TLS, performs the appropriate steps to automatically acquire a trusted certificate prepared by Let's Encrypt certificate authority. An example of such a link is shown in table 3. The corresponding annotation of cert-manager.io/cluster-issuer and the spec.tls section informs the *cert-manager* mechanism to acquire a certificate and handle encrypted HTTPS traffic. This is done by temporarily manipulating the routing on *AppGw*, exposing temporary URLs

pointing to Let's Encrypt *Pod*. A third-party certification system is then requested in the next step to issue a certificate, and confirmation of domain authority is achieved through a corresponding feedback message provided on the temporary URL. Once the certificate is correctly obtained, this certificate is stored in the *Secret* on Kubernetes cluster and automatically installed on *AppGw* by the *cert-manager*. In addition, Let's Encrypt cert-manager itself takes care of the appropriate rollover of expired certificates automatically.

4.4. Internal entry for maintenance team

Basing communication on two-party access to applications provides the possibility of maintenance operation. If it is necessary to temporarily disable end-user access for administrative work, the easiest way to achieve this is to temporarily redirect traffic on the *AppGw* to the maintenance work page or appropriate HTTP *503 Service Unavailable* response. At the same time, access to the application is still possible from internal corporate traffic directly using the *Load Balancer* interface integrated with the *Nginx Controller*.

4.5. Single Sign-on with Nginx

Nginx proxied applications can use *Single Sign-On* (SSO) to secure access to them. Several solutions providing SSO authorization and authentication can be integrated for this purpose, such as Auth0, Keycloak, OneLogin or Microsoft Active Directory FS. This makes it possible to centralize access handling for individual elements of the overall system.

4.6. Caching, compression, and scaling

Nginx server can also cache and compress a content to increase user experience. If caching is not possible or sufficient to handle the traffic, we can easily scale applications by increasing the number of *Pods* that handle requests. *Nginx* provides a load balancer mechanism to handle scalable network traffic

5. Conclusions

This paper presents an architectural pattern using a Dual Ingress consisting of Azure *Application Gateway* and *Nginx Ingress Controller* as a reverse proxy. The described solution provides several functionalities that combine the capabilities of both mechanisms into a single cohesive system. The use of Kubernetes cluster provides the solution with high scalability alongside with additional SaaS services enhancing the capabilities of the system. The discussed solution has been successfully implemented in a commercial solution providing empirical confirmation of the designed advantages. The dual-tier Ingress system successfully handles variable network traffic using the austoscaler and *Nginx Load Balancer*. The presented

architecture pattern provides a cost-optimized solution, adapting to the requirements of the current network traffic. It allows for flexibility in choosing where to implement certain mechanisms, considering the cost-effectiveness of implementation between *AppGw* and *Nginx*. The entire solution was deployed in the Microsoft Azure cloud, using *Application Gateway Ingress* as the internet facing interface. The presented solution, combined with the tools provided by Crossplane (Crossplane Concepts, 2023), made it possible to achieve full automation of the environment provisioning and application deployment process on Kubernetes cluster. Network traffic is secured by appropriate encryption of connections using automatic acquisition of TLS certificates. At the same time, the presented architecture provides internal access to system components during maintenance windows, which requires temporary disconnection of services from public access.

It is worth mentioning that the proposed architectural pattern of the Dual Ingress on Kubernetes cluster is not limited to the Azure cloud only. The presented example was discussed in the context of cloud from Microsoft, however, the general concept is also feasible to implement in other Kubernetes service providers, like AWS or Google. The required changes will only relate to the practical application, which requires the installation of corresponding controllers available from the respective service provider.

References

1. Burns, B., Beda, J., Hightower, K., Evenson, L. (2022). *Kubernetes: Up and Running*. O'Reilly Media.
2. *CNCF Annual survey 2021* (2021). Retrieved from: https://www.cncf.io/wp-content/uploads/2022/02/CNCF-AR_FINAL-edits-15.2.21.pdf, 5 May 2023.
3. *Crossplain Concepts* (2023). Retrieved from: https://docs.crossplane.io/v1.12/concepts/, 15 April 2023.
4. *Ingress with NGINX controller on Google Kubernetes Engine* (2023). Retrieved from: https://cloud.google.com/community/tutorials/nginx-ingress-gke, 6 May 2023.
5. *Microsoft learn: Create an ingress controller in Azure Kubernetes Service (AKS)* (2023). Available online. Retrieved from: https://learn.microsoft.com/en-us/azure/aks/ingress-basic?tabs=azure-cli, 5 May 2023.
6. *Microsoft learn: Creating an ingress controller with a new Application Gateway* (2023). Retrieved from: https://learn.microsoft.com/en-us/azure/application-gateway/ingress-controller-install-new, 15 May 2023.
7. *Microsoft learn: Use TLS with an ingress controller on Azure Kubernetes Service (AKS)* (2023). Retrieved from: https://learn.microsoft.com/en-us/azure/aks/ingress-tls?tabs=azure-cli, 15 May 2023.

8. *Microsoft learn: What is Azure Application Gateway?* (2023). Retrieved from: https://learn.microsoft.com/en-us/azure/application-gateway/overview, 6 May 2023.
9. *Nginx docs - Nginx Ingress Controller* (2023). Retrieved from: https://docs.nginx.com/nginx-ingress-controller/, 5 May 2023.
10. Palmer, M. (2023). *Kubernetes Ingress with Nginx Example – Kubernetes Book*. Retrieved from: https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-ingress-guide-nginx-example.html, 5 May 2023.
11. *Provide external access to Kubernetes services in Amazon EKS | AWS re:Post* (2023). Retrieved from: https://repost.aws/knowledge-center/eks-access-kubernetes-services, 5 May 2023.
12. *Top 7 Benefits of Kubernetes* (2022). IBM Cloud Education. Retrieved from: https://www.ibm.com/cloud/blog/top-7-benefits-of-kubernetes, 5 May 2023.