Kubernetes Networking, Routing, Ingress, and HTTPS on Oracle Kubernetes Engine (OKE)

Talentlink

Challeneges & Solutions

Alizada,Erfan E.

2025

**Introduction**

Deploying applications on Kubernetes in **Oracle Cloud (OKE)** requires solving multiple networking challenges. By default, workloads use **ClusterIP** (internal-only) or **LoadBalancer** (public, one per service). Without careful configuration, nodes may be exposed to the internet, API access may break, and scaling with domains and TLS becomes painful.

In this guide, we’ll walk through the challenges I faced and the solutions I applied. We’ll cover:

1. **VCN and gateways**: securing API, node, and load balancer subnets
2. **Ingress vs ClusterIP/LoadBalancer**: why Ingress is better
3. **Ingress controller setup**: replacing multiple LoadBalancers with one entrypoint
4. **Domain & DNS setup**: mapping a real domain to our Ingress LB
5. **TLS with Let’s Encrypt**: securing traffic with auto-renewing certificates

**1. VCN and Gateways: Securing API, Node, and Load Balancer Subnets**

When I first examined the networking of my OKE cluster, I noticed that all three subnets the **API endpoint subnet**, the **worker node subnet**, and the **service load balancer subnet** were using the same public route table. This route table had one rule: send all traffic (0.0.0.0/0) straight to the **Internet Gateway (IGW)**.

That seemed simple, but it was also insecure. It meant my **worker nodes were directly exposed to the internet**. At one point, I tried moving the API endpoint behind a NAT Gateway to hide it, but the moment I did, my **kubectl** access from VS Code stopped working. That was because the Kubernetes API must always remain publicly reachable.

To solve this, I created a **NAT Gateway** and then set up two new route tables. The API endpoint subnet stayed with IGW so I could continue to reach it with kubectl. The worker node subnet switched to NAT, giving nodes secure outbound internet access without exposing them. The service load balancer subnet remained on IGW because load balancers must accept public traffic.

After creating and assigning the route tables in the OCI console, I validated everything. First, I checked the API endpoint was still reachable:

A black background with white text

AI-generated content may be incorrect.

The nodes showed as Ready.

Then I tested outbound connectivity from a pod running inside the cluster:



The ***HTTP/2 200*** response confirmed the nodes could reach the internet through NAT. With that, my VCN and gateways were properly secured.

**2. Ingress vs ClusterIP/LoadBalancer: Why Ingress Is Better**

Once networking was stable, I turned to the question of how to expose my applications. At first, I relied on ClusterIP services and LoadBalancer services. ClusterIP is Kubernetes’ default, and it only makes services accessible inside the cluster. This is fine for internal communication, but external users cannot access it.

To expose workloads externally, I used LoadBalancer services. These created individual OCI load balancers, each with its own IP. It worked, but quickly became inefficient. Each new service would add costs, requires new DNS entries, and lacked centralized TLS or path-based routing.

At this point, I asked myself the following guiding questions:

* *How can this be done in a more efficient way according to Enterprise architecture?*

**The answer** to this question on google **was Ingress using NGINX Config** which brought up the following question:

* **“*How do Ingress and an NGINX Ingress controller provide a scalable way to expose applications in enterprise Kubernetes architecture?”***

I used this question to explore how I could enhance my design and make it more efficient according to enterprise architecture principles.

In my project setup (frontend + backend in VS Code), I included both an **NGINX configuration** and an **Ingress resource**. Here is how they work together:

* The **Ingress resource** defines the external access rules in Kubernetes. For example, requests to / are routed to the frontend service, while requests to /api are routed to the backend service.
* The **Ingress controller (NGINX)** watches these rules and configures itself accordingly. When traffic comes in through the single OCI load balancer, NGINX applies the Ingress rules and forwards the request to the right Kubernetes service.
* Inside the cluster, NGINX communicates with the backend and frontend pods via their respective **ClusterIP services**. These ClusterIP services act as stable internal endpoints that load balance traffic across pods.
* My custom **NGINX configuration file** can also define additional behavior (like caching, compression, static file serving, or custom headers), giving me more control beyond just routing.

This means that external users only ever hit one public IP (the OCI load balancer), and all the complexity of routing, TLS termination, and traffic distribution is handled centrally by the NGINX Ingress controller. This approach aligned well with enterprise architecture goals: it reduced cost, simplified DNS management, and created a scalable and maintainable solution.

**3. Ingress Controller Setup: Replacing Multiple LoadBalancers with One Entrypoint**

To implement this, I installed the **NGINX Ingress controller** using Helm:

A screen shot of a computer

AI-generated content may be incorrect.

The installation created a service of type LoadBalancer in OCI. I waited for it to receive an external IP:

A black background with white text

AI-generated content may be incorrect.

That IP became the single public entrypoint for all my apps.

Inside the cluster, I kept my frontend and backend as **ClusterIP services** (See pic1),

After applying these services, I wrote an Ingress manifest to route external traffic(See pic2).  
  
A screen shot of a computer

AI-generated content may be incorrect. A screen shot of a computer

AI-generated content may be incorrect.

Pic 1 Pic 2

With this configuration and Ingress we created a solution that a single IP could can serve multiple apps securely, while internal services remains ClusterIP-only.

**4. Domain & DNS Setup: Mapping a Real Domain to My Ingress LB**

A public IP worked, but I wanted a proper domain name. I purchased **talentlink-erfan.nl** and created an **A record** at my DNS provider that pointed directly to the Ingress controller’s external IP.

To confirm the setup, I ran:



It resolved to the correct OCI load balancer IP. With this step, I no longer had to share an IP address I could give users a clean URL.

**5. TLS with Let’s Encrypt: Securing Traffic with Auto-Renewing Certificates**

After setting up Ingress and my domain, the final piece was securing everything with HTTPS. Without HTTPS, browsers show warnings, and data travels in plain text. To solve this, I decided to use **cert-manager** together with **Let’s Encrypt**.

**What Helm Is and Why I Used It**

Helm is a **package manager for Kubernetes**. Instead of writing dozens of YAML files by hand, Helm lets me install entire applications into my cluster with one command. It fetches Kubernetes resources packaged in “charts,” applies them, and manages updates.

I used Helm to install **cert-manager**, which is the Kubernetes component that requests and manages certificates from Certificate Authorities (CAs) like Let’s Encrypt. Here’s the command I ran:

A computer screen shot of white text

AI-generated content may be incorrect.

This did several things for me:

1. Added the Jetstack Helm repo, which contains the official cert-manager charts.
2. Installed cert-manager into the cert-manager namespace.
3. Created the required CRDs (Custom Resource Definitions) that allow Kubernetes to understand new resource types like Certificate, ClusterIssuer, and Issuer.

Without Helm, I would have had to manually apply many YAML files. With it, I got cert-manager running with a single command.

**What Cert-Manager Is and Why I Needed It**

Cert-manager is a **Kubernetes add-on** that automates the creation, renewal, and use of TLS certificates inside the cluster. It integrates with issuers like Let’s Encrypt to automatically request certificates when an Ingress needs them. It also keeps track of expiration dates and renews certificates before they expire, without me doing anything.

Without cert-manager, I would have to manually generate certificates, store them as Kubernetes secrets, and replace them every 90 days. With cert-manager, this entire process is automated.

**What Let’s Encrypt Is and Why I Chose It**

Let’s Encrypt is a **Certificate Authority (CA)** that provides free TLS/SSL certificates trusted by all modern browsers. Normally, getting a certificate involves paying a CA, validating domain ownership, and installing it. Let’s Encrypt automates this through the **ACME protocol**, where challenges (like HTTP requests to /.well-known/acme-challenge/) prove I own the domain.

I chose Let’s Encrypt because it’s free, widely trusted, and works seamlessly with cert-manager in Kubernetes.

**The ClusterIssuer and Why It Is Needed**

Once cert-manager was installed, I had to tell it **which CA to use and how to request certificates**. That’s where the **ClusterIssuer** comes in. A ClusterIssuer is a Kubernetes resource that defines the configuration for cert-manager to connect to a CA.  
A screenshot of a computer

AI-generated content may be incorrect.

This YAML tells cert-manager:

* Use my email (Example@gmail.com) when registering with Let’s Encrypt. This is important because Let’s Encrypt uses it to send expiry notices.
* Use the production Let’s Encrypt server (https://acme-v02.api.letsencrypt.org/directory) to issue real certificates.
* Store the ACME account private key in a Kubernetes secret called letsencrypt-prod-account-key.
* Solve domain validation using an **HTTP-01 challenge**, where Let’s Encrypt will make an HTTP request to my domain, and cert-manager will serve the correct response through the NGINX Ingress controller.

Without this ClusterIssuer, cert-manager wouldn’t know how to talk to Let’s Encrypt.

**How HTTPS Was Secured End-to-End**

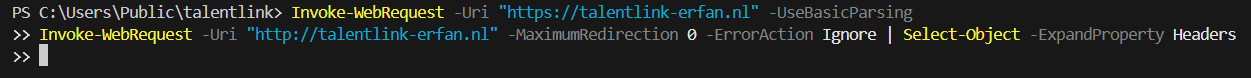
Here’s how the process worked in practice:

1. I applied the ClusterIssuer with:



1. My Ingress referenced the letsencrypt-prod issuer and requested a certificate for talentlink-erfan.nl.
2. Cert-manager automatically created a challenge. It told NGINX to serve a special file at <http://talentlink-erfan.nl/.well-known/acme-challenge/>....
3. Let’s Encrypt connected to my Ingress LoadBalancer, followed the domain, hit the challenge URL, and confirmed I owned the domain.
4. Once validated, Let’s Encrypt issued a TLS certificate valid for 90 days. Cert-manager stored it in a secret called myapp-tls.
5. The Ingress automatically began using this secret for TLS termination. From that point, any request to https://talentlink-erfan.nl was encrypted.

To test it, I ran:



The first command returned 200 OK with my site’s HTML, and the second showed 308 Permanent Redirect, proving that all HTTP traffic was automatically redirected to HTTPS.

**Why This Matters**

Before, my site was only reachable via plain HTTP, which was insecure and flagged by browsers. Now, thanks to **Helm, cert-manager, and Let’s Encrypt**, my site automatically:

* Proves domain ownership
* Issues a trusted TLS certificate
* Terminates TLS at the Ingress
* Redirects all traffic to HTTPS
* Auto-renews certificates before expiry

This means I never need to worry about manually renewing certificates or accidentally serving insecure traffic. My cluster now enforces modern HTTPS security by default.

**Final Outcome and Conclusion**

By going through these steps, I transformed my OKE cluster from a simple but insecure setup into a **production-ready platform** that is both secure and scalable. At the networking layer, I corrected the default OKE configuration by splitting my subnets across the proper gateways: the API endpoint remained reachable through the Internet Gateway, worker nodes were safely hidden behind a NAT Gateway, and the service load balancer subnet continued using the Internet Gateway for public traffic. This ensured the cluster was both secure and functional.

On top of that, I replaced the inefficient approach of exposing each service with its own LoadBalancer by installing an NGINX Ingress controller. With Ingress, I was able to centralize routing, reduce costs, and gain flexibility to map multiple services under one entrypoint. This became even more powerful once I registered my own domain, **talentlink-erfan.nl**, and pointed it to the Ingress LoadBalancer.

Finally, I added **TLS with Let’s Encrypt** using cert-manager. Thanks to Helm, the setup was automated, and the ClusterIssuer ensured that certificates were requested, validated, and renewed without my intervention. My Ingress now terminates TLS and enforces HTTPS, meaning that users accessing my domain are always securely connected.

In the end, I moved from a cluster that was exposed, fragmented, and running over plain HTTP, to one that is **secure, automated, and professionally routed**. With these improvements, my OKE environment is ready to handle real-world traffic with confidence, while keeping operations streamlined and maintenance low.

# References

cert-manager. (n.d.). *cert-manager*. From cert-manager: https://cert-manager.io/docs/

Documentation, K. (n.d.). *Ingress*. From Kubernetes: https://kubernetes.io/docs/concepts/services-networking/ingress/

Documentation, K. (n.d.). *Service*. From Kubernetes: https://kubernetes.io/docs/concepts/services-networking/service/

Documentation, O. C. (n.d.). *Overview of Kubernetes Engine (OKE)*. From Oracle: https://docs.oracle.com/en-us/iaas/Content/ContEng/Concepts/contengoverview.htm

Encrypt, L. (n.d.). *How It Works*. From Let's Encrypt: https://letsencrypt.org/how-it-works/

Helm. (n.d.). *Helm Docs*. From Helm: https://helm.sh/docs/

kubernetes.gihub.io. (n.d.). *Ingress-Nginx Controller*. From kubernetes.gihub.io: https://kubernetes.github.io/ingress-nginx/how-it-works/

Oracle. (n.d.). *Oracle Cloud Infrastructure Documentation* . From Oracle: https://docs.oracle.com/en-us/iaas/Content/Network/Concepts/overview.htm

**Learning outcome:**

1. **Learning Outcome 1 – Professional Standard**
   * I took responsibility for solving a complex ICT issue (cluster networking, ingress, TLS).
   * I applied critical thinking (splitting IGW/NAT roles, moving from multiple LoadBalancers to Ingress).
   * I delivered a professional product (secure, production-ready Kubernetes cluster).
   * I solution is transferable — someone can follow your document from scratch.

This fits perfectly as a **Professional Standard case study**.

1. **Learning Outcome 3 – Scalable Architectures**
   * I explicitly addressed **non-functional requirements**: security (TLS, NAT), scalability (Ingress replacing multiple LoadBalancers), and cost-effectiveness (reducing OCI LBs).
   * I chose architectural patterns (Ingress Controller, ClusterIssuer + cert-manager) that support quality requirements.

This is also strongly aligned with **Scalable Architectures**.

1. **Learning Outcome 4 – Development and Operations (DevOps)**
   * I used **Helm** to install cert-manager and ingress-nginx.
   * I set up a **production-ready environment** (domain + TLS).
   * I automated certificate renewals with cert-manager (zero manual intervention, reduces downtime).

This gives a clear link to **DevOps** principles.

1. **Learning Outcome 5 – Cloud Native**
   * I deployed to a **cloud platform (OCI)**.
   * I integrated **cloud services** (OCI Load Balancers, VCN with NAT/IGW, Let’s Encrypt).
   * I explained the added value of these services (security, scalability, cost optimization).

This ties my work to **Cloud Native development**.