**Professional Software Architecture (DevOps and Cloud-Native Architecture)**

Talentlink

Focus: Non-Functional Requirements

Alizada,Erfan E.

2025

**Introduction**

In the Talentlink project, my goal was not only to deliver a functioning software system but also to build it according to modern industry standards showing my capabilities of creating an enterprise piece of software. I wanted to make professional choices about architecture, deployment, and operations that reflect how enterprise applications are built and maintained. This essay describes my current setup in detail, explains the reasoning behind my technical decisions.

**The Setup: From Code to Cloud**

At its core, Talentlink consists of two main applications: Currently a **Python backend** and a **Flutter web frontend**. Each is developed separately but deployed together to provide a full-stack solution.

From the very beginning, I wanted to avoid the trap of “it works on my machine.” To achieve that, I used **Docker** to containerize both services. The backend is packaged into a Python Docker image, while the frontend is compiled into static web files and then served by **Nginx** in its own Docker image. This ensures that both services run the same way in development, testing, and production.

**Repository Structure**

To keep the project professional and maintainable, I divided the repository into clear modules:

A screenshot of a computer

AI-generated content may be incorrect.

* **backend/:** Python app, requirements, and Dockerfile.
* **frontend/**: Flutter web build, Nginx config, and Dockerfile.
* **k8s/**: Kubernetes manifests for deployments, services, and ingress.
* **.github/workflows/**: CI/CD pipelines for automation.  
  This separation of concerns makes it easier to scale, test, and maintain different parts of the system independently.

**Automation: CI/CD Pipelines**

The most important part of my setup is the **CI/CD pipeline**. I implemented it using **GitHub Actions**, which automatically builds, tests, and deploys my code.

**Continuous Integration (CI)**

Whenever I push code or create a pull request, the CI workflow is triggered. The steps are:

1. **Checkout** the repository.
2. **Log in to GitHub Container Registry (GHCR).**
3. **Build Docker images** for both backend and frontend.
4. **Push the images to GHCR.**

A screenshot of a black screen

AI-generated content may be incorrect.

At this point, I have versioned and reproducible Docker images for every build. This means my code is always tested in an environment that mirrors production.  
  
**Continuous Deployment (CD)**

The deployment workflow is triggered only when changes are merged into the main branch. This is a deliberate choice. I want to ensure that only production-ready code is deployed.

The CD workflow does the following:

1. **Install and configure OCI CLI** (Oracle Cloud Infrastructure command-line tool).
2. **Configure kubeconfig** to connect to my Kubernetes cluster in Oracle Cloud.
3. **Deploy Kubernetes manifests** from the k8s/ folder.
4. **Restart deployments** to pull the latest Docker images.
5. **Verify pods and services** by checking that the new versions are running and accessible.  
   A screenshot of a computer screen

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

   AI-generated content may be incorrect.

The result is that whenever I push to main, Oracle Kubernetes automatically pulls the new Docker images from GHCR and updates the running services. My system is always live and up to date, with minimal manual intervention.

**The Runtime Environment**

Once deployed, the application runs entirely in **Oracle Cloud Kubernetes (OKE)**:

* The **backend** is exposed as a Kubernetes Deployment and Service.
* The **frontend** runs as an Nginx container serving Flutter’s compiled web files.
* An **Ingress** resource handles routing from the outside world into the cluster.
* **Cluster issuers and configs** manage certificates and security for HTTPS connections.

This environment is cloud-native, scalable, and production-ready. Kubernetes ensures that if one pod crashes, it will automatically restart. If more traffic comes in, the system can be scaled horizontally.

**Why This Setup Matters**

At first glance, my setup may look like a standard cloud-native project. But for me, it represents a deliberate journey of applying professional practices, critical thinking, and personal leadership. I made choices based not just on functionality, but on **non-functional requirements**: scalability, security, reproducibility, and maintainability.

For example:

* I chose **Docker** because it guarantees consistency across environments.
* I used **GHCR** for storing images because it integrates directly with my repository and provides secure access control.
* I deployed on **Oracle Kubernetes** because it supports automatic scaling and aligns with enterprise practices.
* I limited CD triggers to main to separate development from production, showing professional discipline.

**Communication:**

* **Dev**: GitHub Actions (CI/CD): developers push code.
* **CI/CD**: GHCR: Docker images are built and pushed to the container registry.
* **GHCR**: Oracle Kubernetes Cluster: the cluster pulls updated images during deployment.
* **Oracle Kubernetes**: Backend + Frontend: the cluster runs both services.
* **Backend:** Frontend: backend provides data consumed by the frontend.
* **Frontend**: Users: the frontend (Flutter Web + Nginx) serves the application to end-users.

**Conclusion**

The Talentlink setup represents more than just a functioning deployment, it demonstrates a structured, professional approach to software development. By combining Docker, GitHub Actions, GHCR, and Oracle Kubernetes, I created a workflow that is automated, reliable, and aligned with enterprise practices.

Every part of the setup is intentional: Docker ensures consistency, GHCR provides secure image storage, Kubernetes offers scalability and resilience, and CI/CD automation guarantees smooth delivery. Together, these choices form a cohesive architecture that is not only functional today but also adaptable for future needs Such as database services that are going to come.

**Relation to Learning Outcomes**

**Learning Outcome 1 – Professional Standard**

I approached the project as if it were an enterprise product. By using Docker, Kubernetes, and CI/CD pipelines, I delivered a professional setup that is reproducible, maintainable, and secure. My choices are future-oriented, with clear documentation and separation of staging/production. This shows accountability, sustainability, and a structured methodology.

**Learning Outcome 2 – Personal Leadership**

I took initiative in designing the architecture and pipelines. I set personal goals (e.g., learning Kubernetes and OCI), sought feedback, and improved my setup iteratively. Restricting deployments to main was my own decision to protect production quality. This shows leadership in guiding my own development and responsibility for my work.

**Learning Outcome 3 – Scalable Architectures**

By deploying on Kubernetes, my architecture is inherently scalable. Services can be replicated, balanced, and updated without downtime. I designed for future growth — databases, monitoring, or additional services can easily be added. Security and GDPR considerations are addressed by managing secrets and using HTTPS.

**Learning Outcome 4 – Development and Operations (DevOps)**

I applied DevOps principles by automating both builds and deployments. My system can be deployed repeatedly, with zero manual steps besides pushing code to main. GitHub Actions integrates with OCI Kubernetes, ensuring continuous integration and continuous delivery are part of the workflow.

**Learning Outcome 5 – Cloud Native**

The project fully embraces cloud-native practices: containerized services, Kubernetes orchestration, and registry-based deployments. It can scale up or down depending on demand. Cloud services like OCI Kubernetes reduce operational overhead and increase reliability.

**Learning Outcome 6 – Security by Design**

Security is integrated into every step. Docker images are stored in a private registry. Kubernetes secrets protect sensitive data. HTTPS ingress ensures encrypted traffic. Limiting CD to main prevents accidental deployments of unfinished or insecure code.

**Learning Outcome 7 – Distributed Data**

Although Talentlink currently does not handle massive data volumes, the architecture is ready to do so. Kubernetes can integrate distributed databases or cloud storage in the future. The design ensures GDPR compliance by managing data securely, with room for ethical and legal considerations in scaling data handling.