# Overview

This document describes the architecture and design of the infrastructure and application deployment for a web application hosted in a Kubernetes cluster. It includes:

* **Network configuration choices** in Terraform.
* Explanation of **Kubernetes deployment strategies** and resource configurations.
* Security measures at **AWS and Kubernetes levels**.
* How the solution ensures **scalability**, **availability**, and **maintainability**.

# Architecture Diagram

A diagram of a software project

Description automatically generated with medium confidence

* All pods are deployed in the private subnets only
* Only Loadbalancer are in the Public subnet (ALB has to be deployed separately, possibly just use Nginx)
* The ALB talks to the K8s service of the Web-apps only
* The Web-apps talk to the backend via DNS to port 8080 – policies can be used to enforce this
* The Web-App is autoscaled via an HPA which is loaded after the deployment of the applications
* The Nat gateway is required for outgoing internet connections from the Pods
* The API is called by the Kubectl tool to manage the cluster , the Kubectl needs to have the context to be able to address the cluster
* All operations in the Clusters are controlled by the API which recieves message from a lot of places

### **1.1 VPC Design**

The network configuration uses Terraform to create an AWS **Virtual Private Cloud (VPC)** with the following components:

* **CIDR block**: 10.0.0.0/16 for a large private address space.
* **Subnets**:
  + **Public Subnets** for internet-facing resources like load balancers (e.g., 10.0.1.0/24, 10.0.2.0/24).
  + **Private Subnets** for backend services and Kubernetes worker nodes (e.g., 10.0.3.0/24, 10.0.4.0/24).
* **NAT Gateway**:
  + A single NAT Gateway is used for private subnets to allow outbound internet access (e.g., for downloading updates or accessing APIs) while maintaining security. However for high availability a NAT Gateway should be deployed on each Availability zone.

### **1.2** Justification of Choices

* **Separation of Public and Private Subnets**:
  + Public subnets are used only for internet-facing services, minimizing exposure of backend services to external threats.
  + Private subnets ensure Kubernetes worker nodes and backend services are not directly accessible from the internet.
* **NAT Gateway**:
  + Provides internet access for private subnets, enabling updates and dependencies for services.
* **Tagging**:
  + Subnets are tagged with kubernetes.io/role/elb for public subnets and kubernetes.io/role/internal-elb for private subnets. These tags ensure proper usage of subnets by Kubernetes when provisioning load balancers.

### **2.1** Deployment Strategies

* **Rolling Updates**:
  + The default strategy is used to ensure zero downtime during application updates.
  + Pods are updated incrementally, maintaining service availability throughout the process. If a deployment has 3 replicas, one replica is updated at a time until all replicas are updated.
* **Replica Management**:
  + Frontend: Configured with 3 replicas for redundancy.
  + Backend: Configured with 2 replicas to ensure fault tolerance.

### **Resource configurations**

* **Requests and Limits**:
  + Defines the minimum and maximum resource allocations for CPU and memory to prevent resource starvation or overcommitment.
* **Probes:**
  + **Liveness Probes**: Ensures the container is running and restarts it if necessary.
  + **Readiness Probes**: Checks if the application is ready to serve traffic before directing requests to it.

### **AWS Level Security**

* **IAM Roles and Policies**
  + **IAM roles are attached to Kubernetes service accounts using IRSA (IAM Roles for Service Accounts).** The EBS CSI Driver uses a specific IAM role with the AmazonEBSCSIDriverPolicy attached, ensuring least privilege.
* **Security Groups**
  + Frontend Security Group: Allows HTTP traffic (port 80) from the internet.
  + Backend Security Group: Restricts access to traffic originating from the frontend.
* **Private Subnets**
  + Worker nodes are deployed in private subnets, making them inaccessible from the internet.

### **Kubernetes Level Security**

* **Namespaces**
  + Applications are isolated into specific namespaces to segregate resources and enforce resource quotas.
* **RBAC (Role-Based Access Control)**
  + Ensures fine-grained permissions for users, applications, and processes. Example: Only specific roles can access or modify certain namespaces or resources.
* **Secrets**:
  + Application secrets (e.g., database credentials) are stored securely in Kubernetes Secrets.

There are many other different way to improve security such as via Istio, securing containers etc but these are beyond the scope of this task.

### **Non functional requirements & Good practices**

Some of the things have been implemented but some need to be implemented, so i mention them here for completeness

* **Scalability** –
  + **Pods** - you would need to use a Horizontal Pod Autoscaler as applied via Kubectl to allow the automatic scaling of pods based on CPU or memory utilización. An HPA has been defined for the front end and the back end for application when the system is up.
  + **Nodes** – you would need to use a cluster autoscaler to be able to Schedule more worker nodes based on pending pods that cant be scheduled due to insufficent resources
  + **Karpenter** - as the cluster autoscaler is an excellent choice because it provides faster scaling and greater flexibility than the traditional Kubernetes Cluster Autoscaler. Karpenter automatically provisions and deprovisions nodes to optimize application performance and infrastructure utilization. This would need to be implemented using terraform and AWS highly recommend using Karpenter for all Cluster autoscaling (<https://docs.aws.amazon.com/pdfs/eks/latest/best-practices/eks-bpg.pdf#karpenter>)
* **Availability** –
  + **Multi-AZ Deployment**: Subnets and worker nodes are distributed across two availability zones to ensure fault tolerance. For a HA system I would be tempted to use all Availability zones, however there are cost and latency concerns that need to be considered in relation to data transfers using techniques such as
    - Pod distribution - Affinity and anti- affinity to spread pods across nodes and Azs
    - Replica management – using deployments to ensure a pod is available in each AZ
    - HPA – to handle spikey demand
    - Liveness probes – to ensure Pods are healthy
  + **Load balancing** - Provides health checks and distributes traffic evenly across healthy Pods.
  + **Cluster Autoscaling** – using tools such as Karpenter to dynamically provision nodes to handle unscheduled pods otherwise the standard autoscaler is deployed where each Nodegroup is assigned its own Autoscaling group and needs to be assigned specific autoscaling permissions to scale it up and down. In addition the cluster autoscaler now needs to be deployed via a Helm chart.
  + **Using Kubernetes Ingress controller** – for highly available routing and SSL termination
  + **File system storage or Object storage - to ensure data is available from each AZ**
* **Maintainability –** 
  + **Infrastructure as Code** - Terraform ensures repeatable, version-controlled infrastructure deployment. It is far simpler to use Terraform providers tan have lots of different YAML files and configurations which are difficult t oread and understand
  + **Helm charts** – Parameterized Helm charts make application deployment consistent and easy to update. There are a lot of ready made applications that can be deployed for reuse in any system architecture
  + **Monitoring and Logging** - AWS CloudWatch and Kubernetes monitoring tools (e.g., Prometheus, Grafana, elastic search ) ensure visibility into application health and can be easily deployed via Helm charts
  + **Disaster recovery – use of Cluster snapshots using tools like Velero to back up and restore cluster state and**