# Kubernetes Cluster:

## **What is the most problematic issue we might have in the Cluster?**

* **Cluster performance.** For example, slow pod startup times, when a large number of pods are starting up at the same time, it can put a strain on the cluster and lead to slow startup times. Timeouts and increasing latencies for responses.
* **Resource Constraints**. Managing resources efficiently to accommodate the varying pod counts (1K to 15K) can be challenging. Oversubscription of resources or resource starvation during peak loads can lead to instability. High CPU and memory usage: k8s pods can consume a lot of CPU and memory, especially if they are running large or complex workloads. With 10K pods running, we need to make sure that the cluster has enough resources to handle the load. Quotas limits will prevent scale increase flexibility.
* Network Congestion: High pod counts can lead to network congestion, affecting inter-service communication and external traffic routing. With a large number of Kubernetes pods running, there is a risk of network congestion, which can lead to slow performance and even outages
* Security Risks: Ensuring secure handling of credentials, secrets, and sensitive data is critical. Unauthorized access to secrets or misconfigured permissions can result in data breaches.

It is important to ensure that the cluster has enough resources (CPU, memory, storage, and networking) to handle the workload. Additionally, we need to make sure that the cluster is configured efficiently and that we are using the best practices for scaling and managing large k8s clusters.

## **How can we plan to resolve/mitigate it?**

* **Monitoring:**
  + Use a Kubernetes-aware monitoring tools (Prometheus, VictoriaMetrics, NetData) to identify performance bottlenecks and troubleshoot issues.
* **Resource Management:**
  + Use Horizontal Pod Autoscaling (HPA) and Vertical Pod Autoscaling (VPA) to dynamically adjust resources based on load.
  + Implement resource quotas and limits to prevent overuse of resources.
* **Network Optimization:**
  + Use a Service Mesh (e.g., Istio or Linkerd) to handle inter-service communication efficiently.
  + Implement Network Policies to control traffic and reduce network congestion.
  + Consider a Content Delivery Network (CDN) for serving static content to offload traffic from the cluster.
* **Security Measures:**
  + Utilize Kubernetes Secrets and ConfigMaps for storing sensitive information.
  + Implement Role-Based Access Control (RBAC) to restrict access to resources and secrets.
  + Store credentials securely in a Secret Manager like HashiCorp Vault or AWS Secrets Manager.
  + Regularly audit and rotate credentials to minimize security risks.
* **High Availability:**
  + Deploy multiple k8s control plane nodes across availability zones or regions to ensure HA.
  + Use horizontal pod autoscaling (HPA): HPA can automatically scale the number of pods running for a given deployment based on CPU or memory usage.
  + Utilize a managed Kubernetes service like Google Kubernetes Engine (GKE), Amazon EKS, or Azure AKS, which often provide built-in high availability features.
* **Managed Services:**
  + Consider using managed k8s services like GKE, EKS, or AKS, which offer automated scaling, patching, and monitoring.
  + Leverage cloud-native load balancers and auto-scaling groups for worker nodes.

## Cluster/Network Schema

To create a Kubernetes cluster with the specified features, we can follow these steps:

* Create separate deployments or StatefulSets for each service.
* Use Horizontal Pod Autoscaling to scale pods based on CPU/memory utilization or custom metrics.
* Implement Cluster Autoscaling to automatically add more nodes when needed.
* Implement Cluster Autoscaling to remove unnecessary nodes during low traffic periods.
* Use standard Kubernetes networking with IPv4.
* Deploy across multiple availability zones/regions if using a cloud provider.
* Use Kubernetes features like Pod anti-affinity and Node affinity to distribute workloads.
* Use ReplicaSets and StatefulSets for redundancy.
* Use Kubernetes Secrets for storing sensitive credentials.
* Implement RBAC (Role-Based Access Control) to restrict access to credentials.

Possible cluster and network schema for a k8s cluster with 10,000 pods:

### Cluster:

* 3 control plane nodes (highly available)
* 100 worker nodes (auto-scaled)
* Each worker node has 64GB of memory and 32 CPUs

### Network:

* Each worker node has two network interfaces:
  + Eth0: For connecting to the external network
  + Eth1: For connecting to the k8s cluster network
* The Kubernetes cluster network uses a CIDR block of 10.0.0.0/16
* A kube-proxy service is running on each worker node to route traffic between pods

### Managed Services

* Google Kubernetes Engine (GKE)
* Amazon Elastic Kubernetes Service (EKS)
* Azure Kubernetes Service (AKS)

### Storage:

* A distributed database, such as Cassandra or CockroachDB, is used to store the application's data.
* Kubernetes PersistentVolumes (PVs) and PersistentVolumeClaims (PVCs) are used to store any persistent data that the application needs.

### Networking:

* A load balancer is used to distribute traffic across multiple pods.

### Monitoring:

* A monitoring tool, such as Prometheus or Grafana, is used to collect metrics from the cluster and visualize them in dashboards.

### Additional requirements:

* Autoscaling: A Kubernetes autoscaler is used to automatically scale the number of worker nodes in the cluster up or down based on the current load.
* Horizontal pod autoscaling: Horizontal pod autoscaling is used to automatically scale the number of replicas of each pod up or down based on the current load.
* Resource limits: Resource limits are set on each pod to ensure that no single pod can consume too many resources and impact the performance of other pods.
* Priority classes: Priority classes are used to prioritize certain pods over others. This can be useful for ensuring that critical pods always have enough resources, even during peaks.
* Pod eviction policy: A pod eviction policy is used to ensure that pods that are not critical to the application are evicted first during peaks.

These services provide features such as auto-scaling, load balancing, and monitoring, which can help us to ensure that our Kubernetes cluster is performing optimally.

|  |
| --- |
| terraform {  required\_providers {  kubernetes = {  source = "hashicorp/kubernetes"  version = "~> 4.0"  } }}  provider "kubernetes" {  host = var.kubernetes\_host  config\_context = var.kubernetes\_config\_context  }  resource "kubernetes\_cluster" "default" {  name = "default"  kubernetes\_version = "v1.23.5"  control\_plane\_endpoint = kubernetes\_cluster.default.control\_plane\_endpoint  node\_taints = [  {  key = "dedicated"  value = "master"  effect = "NoSchedule"  }  ]  node\_labels = {  dedicated = "master"  }  node\_count = 3  autoscaler = {  min\_nodes = 10  max\_nodes = 50  }  }  resource "kubernetes\_cluster\_autoscaler" "default" {  metadata {  name = "default"  }  min\_nodes = 10  max\_nodes = 50  }  resource "kubernetes\_node\_group" "worker" {  name = "worker"  node\_count = 100  taints = [  {  key = "dedicated"  value = "worker"  effect = "NoSchedule"  }  ]  labels = {  dedicated = "worker"  }  node\_selector = {  dedicated = "worker"  }  kubelet\_config = {  eviction\_hard = {  memory.available = "1Gi"  }  }  }  resource "kubernetes\_service\_account" "default" {  name = "default"  }  resource "kubernetes\_priority\_class" "default" {  name = "default"  value = 1000  }  resource "kubernetes\_priority\_class" "high" {  name = "high"  value = 10000  }  resource "kubernetes\_pod\_disruption\_budget" "default" {  metadata {  name = "default"  }  min\_available = 1  selector {  match\_labels = {  app = "myapp"  }  }} |

## API Service Architecture:

1. **Service Architecture:**

* API Gateway (e.g., AWS API Gateway, Kong, or Ambassador) for handling external requests.
* Microservices deployed in containers within the k8s cluster.
* Use an Ingress Controller to route traffic to the appropriate services.

1. **Security and Segregation:**

* Use OAuth 2.0 or JWT (JSON Web Tokens) for authentication and authorization.
* Implement API key or token-based authentication for clients.
* Enforce rate limiting and throttling at the API Gateway layer.
* Isolate client data within separate k8s namespaces or RBAC policies.   
  The API Gateway can isolate clients from each other by running them on separate servers or by using separate databases.
* Employ Network Policies in k8s to restrict communication between different client namespaces.
* Use a Web Application Firewall (WAF) to mitigate DDoS attacks at the API Gateway level.
* Dynamic Request Count:
* Implement Horizontal Pod Autoscaling for the microservices.
* Use Kubernetes Custom Metrics for more precise scaling.
* Base DNS:
* Configure DNS records to route traffic to the API Gateway.
* Resilience and DDoS Mitigation:
* Implement autoscaling to handle traffic spikes.
  + Use a WAF to detect and block malicious traffic.
  + Set up DDoS protection services offered by cloud provider.
* Handling Breaking Changes:
  + Implement versioning in the API (e.g., /v1/resource).
  + Use API Gateway policies to route different versions to different services.
  + Delivery breaking changes to affected clients and provide migration plans.

1. **Deployment Strategy:**

* Deploy the API service using Kubernetes on a managed Kubernetes service like Amazon EKS or Google GKE.
* Use Kubernetes for container orchestration, scaling, and management, as it provides granular control and scalability.
* Fargate, ECS, and Lambda are not recommended for this scenario due to their limitations in handling complex requirements like dynamic scaling, network policies, and API Gateway functionality.
* Managed Kubernetes services are well-suited for containerized microservices and provide the necessary features for high availability, scalability, and security.