YouTube Video URL:

https://youtu.be/XXf8-Snf1jA

Q1. Imagine you are working as an SRE and you have been given a task of designing an architecture for a micro-services based application. How would you design it to handle sudden traffic spike in a cloud environment?

Lets assume that we are deploying this application to an AWS EKS Cluster. To handle sudden traffic spikes in a microservices-based application on AWS EKS, I would design a **scalable, resilient, and self-healing architecture** using best practices like **autoscaling**, **load balancing**, and **observability tools**. The goal is to make sure the system can grow quickly when there's high demand and shrink back when traffic drops, all without human intervention.“

To handle sudden traffic spike, we can utilize the following approach:

1. **Horizontal Pod Autoscaling (HPA):** HPA automatically increases or decreases the number of **pods** based on CPU, memory, or custom metrics like request count.

2. **Cluster Autoscaler for Node Scaling:** A few points:-

• What if the number of pods increase but there's not enough room in the cluster to run them?
• **Cluster Autoscaler** adds more **EC2 nodes** to your managed node group.
• It also scales down unused nodes when traffic reduces.

3. Load Balancing with AWS ALB (Application Load Balancer): -
AWS ALB + Kubernetes Ingress = perfect combo.
ALB distributes traffic evenly across healthy pods and avoids overloaded ones.

4. Observability (Logs, Metrics, Alerts) - You can’t fix what you can’t see - observability is your eyes and ears.
• Use **CloudWatch**, **Prometheus**, and **Grafana** to track performance.
• Monitor pod metrics (CPU, memory), request latency, and errors.
• Set up **alerts** for spikes, failures, or when autoscaling doesn't trigger.

Q2. How would you aggregate logs from multiple services for centralized log analysis so that it can help you in troubleshooting issues?

We use a **centralized logging system** to collect logs from **many services** so that we can:
Troubleshoot issues faster, Spot patterns or errors, Monitor system health in real-time

1. Log Generation: Every app or microservice running on your system writes logs. Example
User logged in, Payment failed - card declined, Database connection timeout.

2. Log Shippers or Agents: We configure these software agents beside the services that collect logs and then can send to a centralized location. Example: Promtail, FluentD, Logstache etc.
They can **read logs from files or containers**, format them neatly, and send them out.

3. Centralized Storage and Processing: All logs are sent to a centralized logging system:
Elasticsearch (for searching and indexing logs)
Amazon CloudWatch Logs (if using AWS)
Splunk, **Datadog**, or **Grafana Loki** (popular in SRE/DevOps world)

This is where logs get **stored, searchable, and structured.** You can find logs using filters like:
By service name
By error level (INFO, ERROR, DEBUG)
By time window
Free Grafana Tool: https://play.grafana.org/drilldown

4. **Visualization and Alerting:** You don't want to read raw logs all day. So we connect our storage to a **dashboarding tool** like KIbana (for Elasticsearch), **Grafana** (with Loki or CloudWatch), Splunk Dashboards. These tools allow you to:
Search logs with Google-like queries
Create visual dashboards
Set up **alerts**: "Alert me if we get 50 errors in 5 minutes!"

Q3. Imagine you're managing a production Kubernetes cluster and one of your critical services suddenly becomes unresponsive. What steps would you take to diagnose and resolve the issue?

1. **Stay calm and acknowledge the impact:** **Unresponsive** means something is broken or very slow and users are likely affected.
   So I quickly check:

- **Is it affecting customers?** (via **monitoring (Email Alerts, Dashboards)** or **user reports)**
- **Is it related to any recent changes in the application.**
- **How severe is it?** (Is it partial or full outage?)

2. Verify the Service Status with with kubectl or a K8s UI tool like Lens : kubectl get pods -n <namespace>
This tells me:

- Are the pods running?
- Are they stuck in CrashLoopBackOff, ImagePullBackOff, Errors or in Pending etc?

3. Check Pod Logs with kubectl or a K8s UI tool like Lens : kubectl logs <pod-name> -n <namespace>
I will look for:

- Any **errors or exceptions** (like database timeouts, missing files, failed dependencies, memory leaks)
- Sudden shutdown messages
- Clues about what's wrong

4. **Describe the Pod: kubectl describe pod <pod-name> -n <namespace>**
I will check for:
Events at the bottom, like:

- Failed to pull image
- Liveness/readiness probes failing
- Node resource issues

5. Use Log Aggregation Visualization tools like Grafana: You will use log search queries to find out error logs:
Search with **Service Name**, **choose time window**, choose **keywords** to filter logs.

6. Check the Service and Networking with kubectl or a K8s UI tool like Lens : Maybe the pod is fine, but users can't reach it. So I will check the **Kubernetes Service**: **kubectl get svc -n <namespace>**
And describe it: **kubectl describe svc <service-name> -n <namespace>**
This will confirm that:
• The service has endpoints (points to healthy pods)
• The ports are correct
• The networking isn't broken

7. Manual reach ability test: like hitting the endpoint directly from inside the cluster:
kubectl exec -it <pod> -- curl http://<service-name>:<port>
This helps me validate:
• Is the service **internally reachable**?
• Is it returning a **proper response**, or timing out?

8. Fixing the issue:
• Restart a failed pod: **kubectl delete pod <pod-name>**
• **If it is related to a recent change, we will try and rollback to the last successful version to restore the issue**
• **Scale the replicas** up or down if it's a load issue.
• **Change other configuration parameters** depending on your investigation.

9. Root Cause Analysis:
• **Why did it break?** (code, infra, config, bad deployment?)
• **What can we do to prevent it in the future?**
• And I write a **Postmortem -** a document explaining:
• What happened?
• What we did?
• What we learned?
• Action items to fix and automate?

Q4. During a deployment, your CI/CD pipeline fails unexpectedly. What would be your approach to troubleshoot and fix the problem?

1. The first thing that we might want to do is **stop any further deployments** till the pipeline is fixed and inform your team about it.

2. To identify the failed step and related job. Look for **failure logs**. Example: **GitLab failed job logs**.
https://gitlab.com/

- **Syntax errors in code or configuration**
- **Missing environment variables**
- **Authentication or permission issues**
- **Failing test cases**
- **Container image pull or build failures**

3. Try to rerun the failed job.

4. Check the last change that was merged. Did it have any **config file changes** (yaml, configmaps, json, shell scripts etc) or was it a code change.

5. Try to **revert the last commit** and see if it works.

6. Check other external dependencies as well:
- Is your container registry up and running
- If your cloud platform like AWS isn't facing any outages
- Any other dependent service or a server (VM) is up and accessible. Example: Jenkins server

7. Root Cause Analysis:
Once you identify the real cause, fix it properly. Examples:
- If the issue was due to a hardcoded secret, move it to a secure vault.
- If a script failed due to a missing file, update your pipeline to validate dependencies first.

Q5. How would you identify areas of unnecessary cloud spending, and what strategies would you use to optimize costs without sacrificing performance?

1. The first thing is to **identify** where your **money** is being **spent**. To do that you can utilize popular Cost Explorer tools: AWS Cost Explorer, GCP Billing Reports, Microsoft Cost Management.

These tools help break down your cloud bill in terms of:
Most expensive services; Region-wise or zone-wise costs; which teams/projects are driving the cost

2. Follow a **general routine checkup (manual or automated)** to identify unused or idle resources:
- **Idle VMs (AWS EC2 Instances)**
- **Under-utilized VM capacity (AWS EC2 Instances) – Right-Sizing resources**
- **Unused/unattached EBS Volumes and snapshots in AWS**
- **S3 Buckets, LBs, Databases etc**

3. A robust **tagging** strategy: **Critical** for cost optimization. Example:
Application Tier: Web/Backend/Database; Project Owner: Dev, Ops; Environment: Dev/Test/Prod etc

4. **Shutting** down resources in **off hours** (example: AWS EC2 Instances).

5. For **random development/testing**, use **EC2 Spot Instances.**

6. If you are using **AutoScaling Groups**, use it smartly to add/remove instances based on **traffic**.

7. For long-term apps, use **reserved instances** or **Savings plans** with EC2.

8. Implement **Budgeting** Alerts for teams or projects so that they are mindful of each resource that they create in the account.