# Question1:

Provide examples of DevOps/DevSecOps tools you have worked with. Explain how you've utilized these tools in your previous project? Explain:
● A specific technical challenge you resolved using these tools
● How would you troubleshoot a pipeline failure where a security scan passes locally but fails in CI?

# Answer:

**I have experience working with a variety of DevOps and DevSecOps tools, including:**

- **CI/CD:** Jenkins, GitLab CI/CD, AWS CodePipeline.
- **Infrastructure as Code (IaC):** Terraform.
- **Configuration Management & Automation:** Ansible.
- **Containerization & Orchestration:** Docker, Kubernetes (EKS), Helm.
- **Security & Compliance:** SonarQube, Trivy, AWS WAF, IAM policies.
- **Monitoring & Logging:** Prometheus, Grafana, ELK Stack (Elasticsearch, Logstash, Kibana), AWS CloudWatch.
- **Version Control & Code Management:** Git, GitHub.

## ● A specific technical challenge you resolved using these tools

We had a **Node.js project** deployed on a **single EC2 server**. While the application functioned correctly, several operational challenges were identified:

- **Manual Deployment & Updates:** Every update required manual intervention, leading to inefficiencies and potential errors.
- **Lack of Security Scanning:** No automated security checks were in place, increasing the risk of vulnerabilities.
- **No Auto-Scaling:** The application could not handle high traffic efficiently, causing downtime.
- **Manually Configured Environments:** Different environments (**Production, Staging, Development**) were set up manually, making consistency and management difficult.

**Challenges**

1. **Frequent Downtime** due to high traffic with no auto-scaling policy.
2. **Time-Consuming Deployments** that required manual effort for every update.
3. **Lack of Standardization** across different environments.
4. **Security Risks** as there were no automated scans for vulnerabilities.

**Solution Implementation**

To resolve these issues, we implemented the following improvements:

**1. Infrastructure as Code (IaC) with Terraform**

- Used **Terraform** to automate and standardize infrastructure setup across all environments (**Prod, Stage, Dev**).
- Ensured consistency and repeatability in deployments.
- Managed networking, IAM roles, and security groups efficiently.

**2. Containerization & Orchestration with EKS**

- **Dockerized** the Node.js application to eliminate dependency issues.
- Deployed the application on **Amazon EKS (Elastic Kubernetes Service)** for better scalability and resilience.
- Implemented **Horizontal Pod Autoscaling (HPA)** to automatically adjust resources based on traffic load.

**3. CI/CD for Automated Deployment**

- Integrated **Jenkins/GitLab CI/CD pipelines** for automated deployment.
- Implemented **automated testing** before deployment to ensure code quality.
- Deployed changes to different environments seamlessly without manual intervention.

**4. Security Enhancements**

- Integrated **Trivy** for automated security scans to detect vulnerabilities in container images before deployment.
- Implemented **IAM role-based access control** to enhance security.
- Configured **AWS WAF & Security Groups** for additional protection against attacks.

## ● How would you troubleshoot a pipeline failure where a security scan passes locally but fails in CI?

For example, if I am using Trivy for a security scan, I will check the following steps:

Check Environment Differences:

- Ensure the Trivy version is the same locally and in CI (trivy --version).
- Verify if the base image differs (docker image inspect <image>).
- Check if the CI runner OS differs from local.

Analyze Pipeline Logs & Exit Codes:

- Review CI logs for detailed errors.
- Run manually in CI (trivy image <image>).

Validate Network & Credentials:

- Ensure Trivy database updates work (trivy --download-db-only).
- If using a private registry, check authentication (TRIVY_AUTH_URL).

Adjust Security Policies:

- Compare .trivyignore or trivy.yaml between local and CI.