**POSSIBLE TECHNICAL QUESTIONS:**

**General DevOps**

1. **What is Infrastructure as Code (IaC), and which tools have you used to implement it?**

**Infrastructure as Code (IaC)** is the practice of managing and provisioning computing infrastructure (like networks, servers, databases, and other resources) through machine-readable definition files, rather than through manual processes. It allows infrastructure to be deployed, managed, and version-controlled using code and automation tools, leading to more consistent and repeatable deployments.

**Common IaC Tools (and examples of usage):**

1. **Terraform (by HashiCorp):**
   * Used for provisioning infrastructure across multiple cloud providers (e.g., AWS, Azure, GCP).
   * Example: Defining VPCs, subnets, and EC2 instances on AWS using .tf files.
2. **AWS CloudFormation:**
   * AWS-native IaC tool for modeling and setting up AWS resources using YAML or JSON templates.
   * Example: Deploying a complete serverless stack (Lambda, API Gateway, DynamoDB) with a single template.
3. **Ansible:**
   * While primarily a configuration management tool, Ansible can also provision infrastructure using modules.
   * Example: Creating cloud instances and configuring them in the same playbook.
4. **Explain the difference between CI and CD. How have you implemented them in your projects?**

**Continuous Integration (CI)**

**CI** refers to the practice of frequently integrating code changes into a shared repository. The goal is to detect errors early, reduce integration problems, and enable fast feedback on changes made to the codebase.

**Key Practices of CI:**

* **Frequent Code Commits**: Developers commit their code changes multiple times per day to the version control system (e.g., Git).
* **Automated Builds**: Once the code is committed, an automated build process is triggered. This build includes compiling the code, running unit tests, and generating artifacts.
* **Automated Testing**: Automated tests (unit tests, integration tests, etc.) are run as part of the build process to verify that the code behaves as expected.
* **Feedback Loops**: Developers are notified of failed tests or build issues immediately, so they can fix them before merging their changes.

**Example CI Tools:**

* **Jenkins**: A popular open-source automation server that can build, test, and deploy software.

**Continuous Delivery (CD)**

**CD** refers to the practice of automatically deploying every change that passes automated tests to a staging or production environment. While **Continuous Delivery** focuses on ensuring that the software can be released at any time, **Continuous Deployment** is a more advanced practice where every change is automatically deployed to production without manual intervention (this is an extension of CD).

**Key Practices of CD:**

* **Automated Deployment**: After passing all tests, code changes are automatically deployed to a staging or production environment. This helps ensure that the software is always in a deployable state.
* **Versioning and Release Management**: Each change is versioned and released in a controlled and repeatable way.
* **Infrastructure as Code (IaC)**: CD often uses IaC tools like Terraform, CloudFormation, or Ansible to ensure environments are consistently deployed.

**Example CD Tools:**

* **Jenkins**: Jenkins can be extended to implement CD pipelines for automated deployment.

**Key Differences Between CI and CD:**

* **CI** focuses on automatically building and testing code frequently to catch issues early, ensuring code is always in a working state.
* **CD** extends CI by automating the deployment process, enabling the ability to release software frequently, either to staging or production.

1. **How do you handle secrets and credentials in a CI/CD pipeline?**

**Use Role-Based Access Control (RBAC)**

When using cloud-based CI/CD services (e.g., AWS, Azure, Google Cloud), **Role-Based Access Control (RBAC)** should be configured to limit access to secrets based on the principle of least privilege. This ensures that only specific services, roles, or users can access sensitive credentials.

For example:

* **AWS IAM** roles can be used in **AWS CodePipeline** or **GitHub Actions** to control which actions or environments can access specific secrets in **AWS Secrets Manager**.

**Encrypt Secrets**

It’s important to ensure that all secrets are encrypted both in transit and at rest. Most cloud providers and secrets management tools automatically handle encryption for you.

* **AWS Secrets Manager**: Secrets are encrypted at rest using AWS KMS (Key Management Service).

1. **Describe blue/green deployment vs. canary deployment strategies.**
2. **What tools do you use for monitoring and logging? Why?**

**Monitoring Tools**

**Monitoring** involves tracking the performance, health, and availability of your applications and infrastructure. It typically includes metrics such as uptime, response times, error rates, and system resource usage (e.g., CPU, memory, disk).

**1. Prometheus & Grafana**

* **Prometheus** is an open-source monitoring and alerting toolkit designed for reliability and scalability. It stores time-series data (e.g., CPU usage, memory usage, request counts) and allows you to query that data.
* **Grafana** is a visualization tool that integrates seamlessly with Prometheus to create dashboards for monitoring and analysis.

**Why I Use Prometheus & Grafana:**

* **Prometheus**:
  + It’s designed for cloud-native environments and integrates well with containerized applications (especially Kubernetes).
  + Provides powerful query language (PromQL) for custom metrics and alerting.
  + Easy to integrate with many microservices, apps, and infrastructure components.
* **Grafana**:
  + Powerful, flexible, and user-friendly dashboards for visualizing Prometheus data.
  + Supports multiple data sources (e.g., Prometheus, Elasticsearch, MySQL, etc.), so it’s not limited to Prometheus.
  + Offers real-time monitoring and alerting through integration with Prometheus.

**Example Usage**: I use Prometheus to collect metrics from Kubernetes clusters, applications, and databases. Grafana visualizes this data to provide real-time insights into system performance.

**Azure-Specific Questions**

1. What are Azure Resource Groups, and how do they relate to managing resources?
2. Explain the role of Azure DevOps Services. What are the differences between Pipelines, Repos, and Artifacts?
3. How would you configure and use Azure Key Vault?
4. What is an Azure Managed Identity, and how is it used in secure application deployments?
5. How do you use Azure Monitor and Log Analytics to troubleshoot issues?

**AWS-Specific Questions**

1. **Compare EC2 Auto Scaling Groups with AWS Elastic Beanstalk.**

 Use **EC2 Auto Scaling Groups** when you need **fine-grained control** over your infrastructure and deployment.

 Use **AWS Elastic Beanstalk** when you prefer **ease of deployment** and are okay with AWS managing most of the underlying infrastructure.

1. **What is the purpose of AWS CloudFormation? How do you handle stack updates?**

**AWS CloudFormation** is an **Infrastructure as Code (IaC)** service that allows you to **provision and manage AWS resources** in a predictable and repeatable way using **template files** written in **YAML or JSON**.

**Handling Stack Updates in CloudFormation**

When you need to **change your infrastructure**, you update the **CloudFormation stack** by modifying the template or parameters.

**Steps to Handle Stack Updates:**

1. **Modify the Template**
   * Edit the YAML or JSON template to reflect the desired changes (e.g., add a new EC2 instance, change instance type).
2. **Deploy the Update**
   * Use the AWS Console, CLI (aws cloudformation update-stack), or SDK to initiate the update.
3. **CloudFormation Performs a Change Set (optional but recommended)**
   * A **Change Set** previews the proposed changes before applying them.
   * Useful to **avoid destructive or unintended changes**.
4. **Update Stack Execution**
   * CloudFormation updates only the resources that have changed.
   * Resources that are **not affected remain untouched**.
5. **How would you use AWS IAM to manage user and resource-level access?**

### **Steps to Manage User Access:**

| **Step** | **Description** |
| --- | --- |
| **1. Create IAM Users** | Create individual IAM users for people or applications. |
| **2. Use Groups** | Assign users to **IAM Groups** (e.g., Admins, Developers, Viewers) to manage permissions collectively. |
| **3. Attach Policies** | Attach **managed** or **custom policies** to users or groups to define allowed actions. |
| **4. Use MFA and Password Policies** | Enforce **Multi-Factor Authentication (MFA)** and strong password policies for security. |

**Best Practices**

* Never use the root account for daily operations.
* Follow **principle of least privilege** – give users only the permissions they need.

**Managing Resource-Level Access**

IAM policies allow **fine-grained access control**, not just to AWS services but also **specific resources** within services.

**Example:**

You can allow a user to:

* Access only a specific **S3 bucket**
* Start/stop only specific **EC2 instances**
* Query only a specific **DynamoDB table**

**Policy Structure:**

IAM policies are written in **JSON** and define:

* **Effect**: Allow or Deny
* **Action**: API calls the user is allowed to perform
* **Resource**: Specific AWS resources the action applies to
* **Condition** (optional): Additional restrictions

1. **What’s the difference between an ALB and a NLB in AWS? When would you use each?**

**Use ALB (Application Load Balancer) When:**

* You need **Layer 7 features**, like:
  + **Path-based routing** (e.g., /api to one service, /images to another)
  + **Host-based routing** (e.g., api.example.com to one target, app.example.com to another)
  + **Header or query string-based routing**
* You need to run **containerized apps (e.g., ECS/Fargate)** with multiple services per host.
* You're serving **web applications** or **microservices** that rely on **HTTP or HTTPS.**
* You need to integrate with **AWS WAF** for request filtering and protection.
* You're using **Lambda functions** as backend targets.

**Use NLB (Network Load Balancer) When:**

* You need **ultra-low latency** and **high throughput**.
* You need to load balance **TCP, TLS, or UDP** traffic (e.g., databases, game servers, VPNs).
* Your application requires **static IP addresses** or **Elastic IPs** for firewall whitelisting.
* You are dealing with **millions of requests per second**.
* You need to preserve the **source IP address** (ALB doesn't forward it by default).

1. How do you use AWS CloudWatch for metric collection and alerting?

**Containerization and Orchestration**

1. How do you manage container orchestration in Azure and AWS (e.g., AKS vs. EKS)?
2. **What are some key considerations when deploying microservices using Kubernetes?**

**Kubernetes Resource Management**

| **Resource** | **Key Considerations** |
| --- | --- |
| **Pods & Deployments** | Use Deployments or StatefulSets to manage replicas and lifecycle. |
| **Namespaces** | Segment environments or teams logically (e.g., dev, prod, team-a). |
| **Resource Limits** | Define requests and limits for CPU and memory to ensure fair scheduling and prevent resource starvation. |
| **Horizontal Pod Autoscaling (HPA)** | Automatically scale services based on metrics like CPU or custom application metrics. |

**Multi-Tenancy and Environment Management**

| **Consideration** | **Best Practice** |
| --- | --- |
| **Environment Isolation** | Use separate clusters or namespaces per environment. |
| **Quota Management** | Apply ResourceQuotas and LimitRanges to prevent overuse. |
| **Access Control** | Use Kubernetes RBAC for secure and granular access control. |

1. Describe how you would set up CI/CD for a containerized application.
2. Explain the role of Helm charts or Terraform modules in managing infrastructure.

**Helm** is focused on **Kubernetes-native** applications.

**Terraform** is focused on **cloud infrastructure provisioning**.

**Practical/Scenario-Based Questions**

1. A deployment is failing in production — walk me through your troubleshooting steps.

**Step 1: Confirm and Define the Problem**

**Goal:** Understand the scope and impact of the failure.

* ❓ **What is failing?** (App crash, rollout stuck, 5xx errors, latency spikes?)
* 🧑‍🤝‍🧑 **Who is affected?** (All users? A subset? A specific feature?)
* 📅 **When did it start?** (Check deployment timestamp, change logs)
* 🧾 **What changed?** (New code, config, infra, secrets?)

**🧪 Step 2: Check the Deployment/CI/CD Pipeline**

**Goal:** Ensure the deployment pipeline completed successfully and pushed the expected version.

* ✅ Check your **CI/CD logs** (e.g., GitHub Actions, Jenkins, GitLab CI)
* 🔀 Confirm correct **image/tag** was deployed
* 🔄 Look for errors in kubectl rollout (e.g., crash loop, probe failures)

bash

Copy code

kubectl rollout status deployment my-app -n production

kubectl describe deployment my-app -n production

**🧼 Step 3: Check Pod and Container Health**

**Goal:** See if pods are running, restarting, or crashing.

* 👀 View pod statuses:

bash

Copy code

kubectl get pods -n production -l app=my-app

* 🔎 Inspect problematic pods:

bash

Copy code

kubectl describe pod <pod-name> -n production

* 📄 Check container logs:

bash

Copy code

kubectl logs <pod-name> -c <container-name> -n production

* 🔁 Look for:
  + Crash loops
  + Readiness/liveness probe failures
  + Init container issues
  + Out-of-memory or resource errors

**🌐 Step 4: Validate Service Connectivity and Networking**

**Goal:** Ensure your app is accessible via services, ingress, or external endpoints.

* 🧪 Test service DNS resolution and port accessibility

bash

Copy code

kubectl exec -it <pod> -- curl http://<service-name>:<port>

* 🔍 Check Ingress logs, NLB/ALB health, and error messages (if exposed to the public)
* 🔥 Investigate any recent **NetworkPolicy** or **Security Group** changes

**🧰 Step 5: Review Logs and Metrics**

**Goal:** Use observability tools to detect anomalies and get clues.

* 📘 **Application logs** via CloudWatch, EFK, or Loki
* 📈 **Metrics** (CPU, memory, response time) via Prometheus/Grafana
* 🔦 **Tracing** with OpenTelemetry, Jaeger, or X-Ray
* ⏱️ Use timestamp filters to correlate logs/metrics with the deployment time

**🔐 Step 6: Validate Configurations and Secrets**

**Goal:** Make sure environment-specific config or secrets didn't break the deployment.

* 🔐 Check if **Secrets**, **ConfigMaps**, or **environment variables** changed
* 🔍 Validate required keys, tokens, endpoints, DB connections
* 🧪 Try mounting/test accessing secrets manually if needed

**♻️ Step 7: Roll Back if Needed**

**Goal:** Quickly restore the last known working state.

* 🔙 Use GitOps or Helm to revert:

bash

Copy code

helm rollback my-app <previous\_revision> -n production

* 🔁 Or rollback Deployment:

bash

Copy code

kubectl rollout undo deployment my-app -n production

**✅ Step 8: Document and Prevent Recurrence**

**Goal:** Capture the root cause and put safeguards in place.

* 📝 Record the root cause in an incident postmortem
* 🧪 Add tests or staging checks to catch it earlier next time
* 🔒 Apply resource limits, better probes, or tighter validation logic
* 📉 Set up alerting for the failure signals you missed

1. How would you migrate a legacy monolith app to the cloud using DevOps principles?
2. Your CI/CD pipeline takes 30+ minutes. What optimizations would you consider?
3. A service is facing high latency intermittently. What tools and methods would you use to diagnose and fix it?