Day to Day activity: AWS Cloud Engineer / AWS Devops Engineer

**1. Monitoring & Incident Management**

* Check alerts from monitoring tools (CloudWatch, Datadog, LogicMonitor, etc.)
* Respond to incidents (CPU spike, disk full, high memory, failed deployments, etc.)
* Log into servers or containers to check logs (using CloudWatch Logs, Kibana, etc.)
* Coordinate with developer and application team and escalate if required
* Update tickets in ServiceNow, Jira, or your ITSM tool

**2. CI/CD Pipeline Maintenance**

* Monitor Jenkins/Azure DevOps/GitHub Actions pipelines
* Fix failed builds or deployment issues
* Add caching, parallel jobs, etc.) modify pipeline
* Implement changes for new feature deployments (YAML updates, variable changes, secret management)

**3. Infrastructure Management (IaC)**

* Work on Terraform code for provisioning resources.
* Review and merge Pull Requests for infra changes
* Apply changes via CI or CLI (Terraform plan/apply, etc.)

**4. Environment & Release Management**

* Manage dev/staging/prod environments
* Perform version upgrades (java version upgrade, nginx, apache version upgrade) or rollbacks (ansible playbook for multiple servers application upgrade)

**5. Security & Access Control**

* Review IAM policies and security group rules
* Rotate credentials/secrets using AWS Secrets Manager
* Audit user and service permissions:
* Patch servers or containers (if still using EC2 or on-prem)

**6. Automation & Scripting**

* Write/modify Bash.
* Automate repetitive tasks (shell script )(e.g., backups, cleanup jobs)
* Update Ansible playbooks or automation templates

**7. Collaboration & Communication**

* Attend daily standups or sync meetings
* Discuss with dev teams about build/deployment needs
* Support QA/UAT teams for test environments

**8. Documentation & Knowledge Sharing**

* Document changes, fixes, playbooks
* Update Confluence/Notion/internal wikis
* Mentor juniors or review their code/scripts

**Question(1). Can you describe your experience with Infrastructure as Code (IaC) using Terraform?**   
Ans:  
In my 6 years of experience as a DevOps and AWS Cloud engineer, I have extensively utilized Terraform for Infrastructure as Code (IaC) implementations. My journey with Terraform began when I worked on provisioning AWS instances, where I learned to create and manage infrastructure using Terraform scripts. I have successfully provisioned various AWS resources such as EC2 instances, VPCs, subnets, and security groups, ensuring that the infrastructure is both scalable and secure.

One of my significant projects involved using Terraform to automate the deployment of a multi-tier application architecture on AWS.  
I created reusable modules that encapsulated the configuration for different environments, which allowed for consistent deployments across development, testing, and production environments.

This modular approach not only improved efficiency but also reduced the chances of human error during deployments.

Additionally, I have experience in managing Terraform state files, including locking and unlocking states using DynamoDB, which is crucial for team collaboration.

I also integrated Terraform with CI/CD pipelines using Jenkins, automating the deployment process and enabling continuous delivery of infrastructure changes.

This experience has equipped me with a solid understanding of best practices in IaC, including version control, state management, and modular design, which I believe are essential for any DevOps role.  
------

**Blue-Green Deployment** is a **deployment strategy** that reduces downtime and risk by running two identical environments — **"Blue"** and **"Green"** — and switching traffic between them during deployment.

**🔵🟢 How It Works**

1. **Two environments**:
   * **Blue**: The current live (production) environment.
   * **Green**: The new version of the app, prepared in parallel.
2. **Deploy to Green**:
   * You deploy the new version of your application to the **Green** environment.
   * It is **not receiving live traffic yet**, so you can test it safely.
3. **Switch traffic**:
   * If everything looks good, you **route traffic from Blue to Green**, often using a **load balancer** or **DNS switch**.
   * Now, **Green is live**.
4. **Fallback option**:
   * If something breaks, you can quickly **switch traffic back to Blue**.
   * This minimizes risk and downtime.

**Summary:**

| **Feature** | **Blue-Green Deployment** |
| --- | --- |
| **Downtime** | **Zero** |
| **Risk** | **Low (easy rollback)** |
| **Cost** | **Higher (two environments)** |
| **Rollback** | **Instant** |
| **Tool Support** | **AWS (ALB,).** |

Safer, **easier testing in production-like environment**.

* Better **continuous delivery** support.

**Example 1: Blue-Green Deployment in AWS using EC2 + ALB**

**Scenario:**  
You manage EC2 instances behind an **Application Load Balancer (ALB)**.

**Steps:**

1. Blue target group is currently active (e.g., EC2 instances running v1.0).
2. Deploy v2.0 to a new set of EC2 instances (Green) and register them with a second target group.
3. Perform health checks and test Green.
4. Update the ALB listener to route traffic from Blue TG to Green TG.
5. Monitor for issues.
6. If rollback needed, switch ALB listener back to Blue TG.