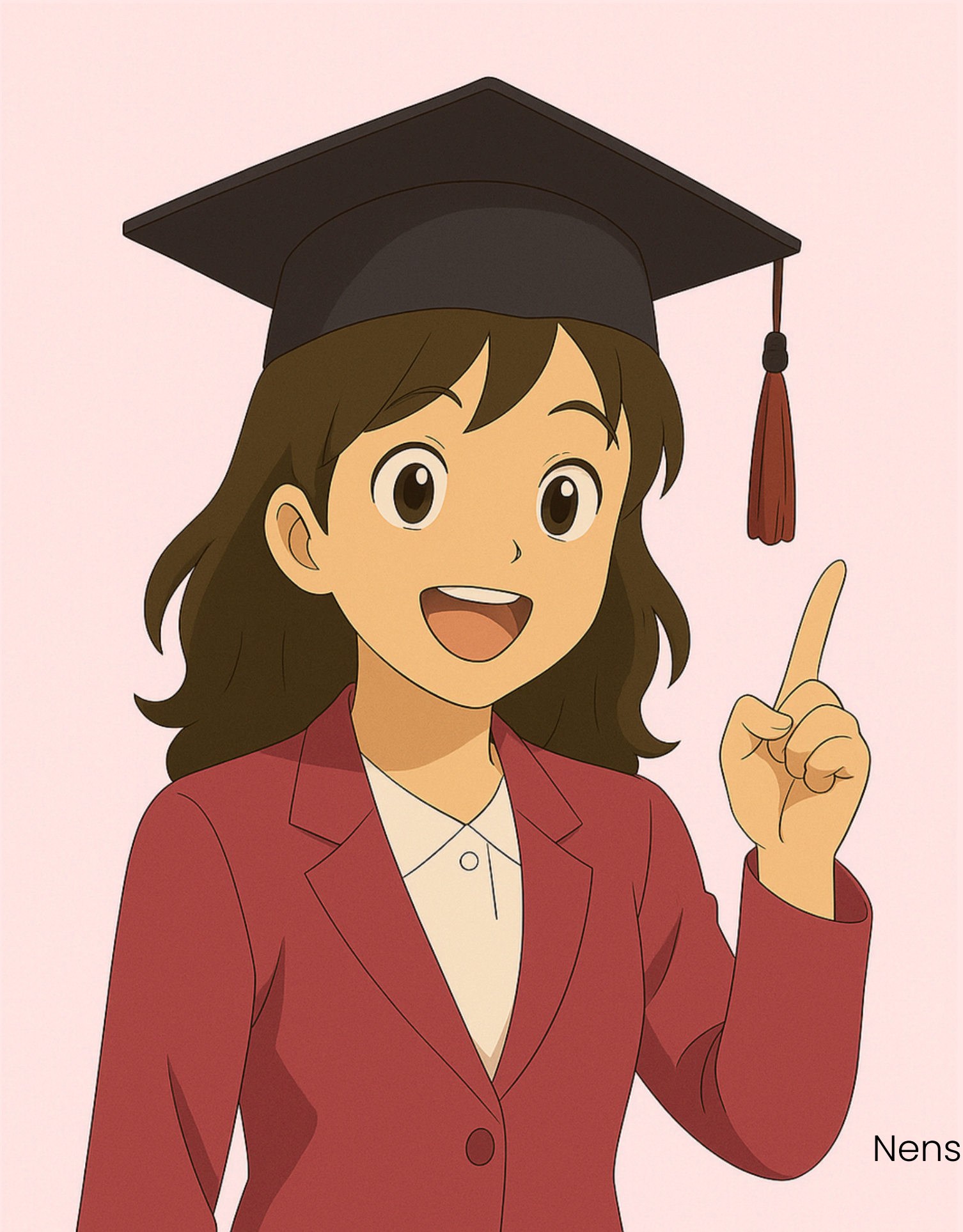
TOP 50

DevOps Interview

Q&A Asked in MNCs



Nensi Ravaliya

Top 50 DevOps Interview Questions & Answers - Nensi Ravaliya

Core DevOps Concepts

1. What is DevOps, and why is it important?

Answer:

DevOps is a set of practices, tools, and a cultural philosophy that aims to bridge the gap between software development and IT operations teams. It emphasizes automation, collaboration, integration, and monitoring throughout the entire software development lifecycle.

Key reasons DevOps is important:

- **Faster Time-to-Market**: Enables rapid and frequent code releases with automated CI/CD pipelines, reducing the time from development to production deployment
- **Improved Quality**: Early detection of bugs and issues through continuous integration and automated testing reduces production defects
- **Better Collaboration**: Breaks down silos between development and operations teams, fostering a culture of shared responsibility
- **Reduced Manual Errors**: Automation eliminates repetitive manual tasks, minimizing human error and ensuring consistency
- **Increased Reliability**: Continuous monitoring and rapid incident response (low MTTR) ensure high system availability
- **Cost Efficiency**: Optimized resource allocation and reduced downtime directly impact the bottom line
- **Competitive Advantage**: Ability to respond quickly to market changes and customer feedback gives organizations an edge over competitors

---

2. What are the key phases of the DevOps lifecycle?

Answer:

The DevOps lifecycle consists of continuous phases that work together seamlessly:

- **Plan**: Define requirements, create backlog, and set goals for the development cycle
- **Code/Develop**: Developers write code and commit it to version control systems like Git
- **Build**: Source code is compiled and built using build automation tools like Jenkins, Maven, or Gradle
- **Test**: Automated tests (unit, integration, functional, security) are executed using tools like Selenium, JUnit
- **Deploy**: Code is deployed to staging and production environments using tools like Docker, Kubernetes, Terraform
- **Operate**: Applications run in production with infrastructure management and configuration
- **Monitor**: Continuous monitoring of performance, availability, and security using tools like Prometheus, ELK Stack, Grafana
- **Feedback**: Data from monitoring informs improvements, and the cycle repeats

Each phase feeds into the next, creating a continuous loop of improvement and rapid delivery.

---

3. Explain the difference between Continuous Integration (CI), Continuous Delivery (CD), and Continuous Deployment (CD)

Answer:

Aspect	CI	Continuous Delivery	Continuous Deployment
Definition	Automatically building, testing, and	Code is always in a deployable state; manual approval	Every code change passing tests is automatically

Aspect	CI	Continuous Delivery	Continuous Deployment
	integrating code changes frequently	needed before production release	deployed to production
Frequency	Multiple times per day	Ready for release at any time	Multiple times per day to production
Manual Steps	Automated after commit	Manual approval before production	Fully automated, no manual approval
Risk Level	Low (early detection)	Low to medium	Medium to high (requires excellent automation)
Example Workflow	Developer commits ′ Build ′ Test ′ Artifact stored	Build + Test + Staging ′ Awaiting approval ′ Ready to deploy	Build + Test ′ Automatic production deployment

Real-world scenario: A web application team uses CI to run tests on every commit. They use Continuous Delivery to ensure builds are always production-ready, with a manual gate for approval. If they implement Continuous Deployment, approved code goes directly to production without manual intervention.

---

4. What are the benefits of DevOps for businesses?

Answer:

- **Faster Releases**: Reduce time-to-market from months to days or hours
- **Higher Quality**: Fewer production incidents and bugs due to automated testing
- **Improved Stability**: Lower Mean Time to Recovery (MTTR) and high system uptime (99.99%+)
- **Better Collaboration**: Shared responsibility increases team morale and productivity
- **Cost Reduction**: Automation reduces operational overhead and resource waste
- **Scalability**: Infrastructure as Code allows easy scaling to handle increased loads

- **Enhanced Security**: Security integrated into every step (DevSecOps) rather than bolted on at the end
- **Better Decision Making**: Real-time monitoring and analytics provide data-driven insights
- **Customer Satisfaction**: Rapid feature delivery and quick bug fixes improve user experience

---

5. What is Infrastructure as Code (IaC), and why is it important?

Answer:

Infrastructure as Code (IaC) is the practice of managing and provisioning computing infrastructure through machine-readable definition files rather than physical hardware configuration or interactive configuration tools.

Key characteristics:

- Infrastructure configurations are stored as code in version control systems (Git)
- Identical infrastructure can be deployed repeatedly and consistently
- Infrastructure changes follow the same code review and testing processes as application code

Important IaC tools:

- **Terraform**: Cloud-agnostic, supports AWS, Azure, GCP, and more
- **CloudFormation**: AWS-specific IaC tool
- **Ansible**: Agentless configuration management for infrastructure and application deployment
- **Pulumi**: Infrastructure as Code using programming languages like Python, Go, Java

Why IaC is important:

- **Reproducibility**: Deploy identical environments across development, staging, and production
- **Version Control**: Track all infrastructure changes with full audit trail

- **Disaster Recovery**: Quickly recreate entire infrastructure if needed
- **Scalability**: Easily provision multiple environments or scale existing ones
- **Cost Optimization**: Identify and eliminate unused resources
- **Compliance**: Enforce consistent security and compliance policies across all environments
- **Documentation**: Code itself serves as living documentation of infrastructure

Example:

```
resource "aws_s3_bucket" "my_bucket" {
  bucket = "my-unique-bucket"
  acl    = "private"

  versioning {
    enabled = true
  }
}
```

Docker and Containerization

6. What is Docker, and how does it work?

Answer:

Docker is a containerization platform that packages applications and all their dependencies (libraries, runtime, configuration files) into a lightweight, isolated container that can run consistently across any environment (development, staging, production).

How Docker works:

1. **Dockerfile**: A text file containing instructions to build a Docker image (like a blueprint)
2. **Docker Image**: An immutable snapshot that includes the application, OS, libraries, and dependencies

3. **Docker Container**: A running instance of a Docker image with its own isolated filesystem, network, and processes
4. **Docker Registry**: A repository (e.g., Docker Hub) storing Docker images for sharing and distribution

Key benefits:

- **Consistency**: "It works on my machine" problem is eliminated because the container environment is identical everywhere
- **Isolation**: Each container runs independently without affecting others
- **Efficiency**: Containers are lightweight (megabytes vs. gigabytes for VMs) and start in milliseconds
- **Portability**: Run the same container on any system with Docker installed
- **Scalability**: Easily scale by running multiple container instances

Example Dockerfile:

```
FROM python:3.9
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "app.py"]
```

---

7. What is the difference between Docker Image and Docker Container?

Answer:

Aspect	Docker Image	Docker Container
Type	Blueprint/template (immutable)	Running instance (mutable)
Creation	Built using Dockerfile	Created from an image
Persistence	Persists in registry	Temporary (deleted when stopped unless committed)

Aspect	Docker Image	Docker Container
Size	Typically kilobytes to megabytes	Same as image + changes made to filesystem
Isolation	No isolation (it's just a file)	Isolated filesystem, network, processes
Example	Like a class in OOP	Like an object/instance in OOP

Analogy: A Docker image is like a recipe, while a Docker container is like the actual dish prepared from that recipe.

---

8. What is Docker Compose, and when would you use it?

Answer:

Docker Compose is a tool for defining and running multi-container Docker applications using a YAML configuration file. It allows you to orchestrate multiple containers as a single application.

Use cases:

- Running microservices that need to communicate with each other
- Local development environments with multiple services (web app + database + cache)
- Testing environments with complete application stacks
- CI/CD pipelines that need multiple services running together

Example docker-compose.yml:

```
version: '3.8'services: web: build: . ports: - "5000:5000" environmen t: - DATABASE_URL=postgresql://db:5432/myapp db: image: postgres:13 environment: - POSTGRES_PASSWORD=secret volumes: - postgres_d ata:/var/lib/postgresql/datavolumes: postgres_data:
```

Common commands:
- `docker-compose up`: Start all services
- `docker-compose down`: Stop all services
- `docker-compose logs`: View service logs
- `docker-compose exec service_name bash`: Execute command in running container

---

9. Explain Docker Swarm and how it differs from Kubernetes

Answer:

Docker Swarm is Docker's native clustering and orchestration tool that turns a group of Docker hosts into a single virtual Docker host.

Feature	Docker Swarm	Kubernetes
Setup	Simple, built-in with Docker	Complex but more powerful
Learning Curve	Easy for beginners	Steep learning curve
Scalability	Good for small to medium deployments	Excellent for large-scale deployments
Features	Basic orchestration	Advanced features (service mesh, auto-scaling, custom resources)
Market Adoption	Less popular	Industry standard (used by Google, Amazon, Microsoft)
High Availability	Limited	Excellent
Load Balancing	Built-in round-robin	Advanced with multiple options

Kubernetes is generally preferred for production environments due to its superior features, community support, and ecosystem.

---

10. Explain Docker Architecture

Answer:

Docker uses a **client-server architecture** with the following components:

1. Docker Client
- Command-line tool (docker command)
- Sends commands to Docker daemon
- Can communicate with remote Docker daemon via API

2. Docker Daemon (Server)
- Runs in background on host machine
- Manages Docker objects (images, containers, networks, volumes)
- Listens to Docker API requests

3. Docker Objects
- **Images**: Read-only templates for creating containers
- **Containers**: Executable instances of images
- **Services**: Definition of how containers run in production (used with Swarm)
- **Networks**: Enable container communication
- **Volumes**: Data storage mechanism

4. Docker Registry
- Repository for storing Docker images
- **Docker Hub**: Official public registry
- **Private registries**: AWS ECR, Azure ACR, Docker Trusted Registry

Architecture Flow:

```
Docker Client
    ↓
Docker API
    ↓
Docker Daemon
    ↓
Images, Containers, Networks, Volumes
```

Kubernetes and Orchestration

11. What is Kubernetes, and why is it important for DevOps?

Answer:

Kubernetes (K8s) is an open-source container orchestration platform that automates the deployment, scaling, management, and networking of containerized applications across a cluster of machines.

Key responsibilities:

- **Deployment**: Automatically place containers on appropriate nodes
- **Scaling**: Horizontally scale applications based on resource demand
- **Self-healing**: Restart failed containers, replace, and reschedule them

- **Load Balancing**: Distribute traffic across containers
- **Rolling Updates**: Update applications without downtime
- **Resource Management**: Optimize resource allocation across the cluster
- **Service Discovery**: Enable containers to find and communicate with each other

Why important for DevOps:

- **High Availability**: Ensures applications remain available even if nodes fail
- **Cost Efficiency**: Optimal resource utilization reduces infrastructure costs
- **Automation**: Reduces manual operations, allowing teams to focus on innovation
- **Standardization**: Consistent platform across different environments
- **Community & Ecosystem**: Large ecosystem of tools and industry adoption

---

12. Explain the main Kubernetes components and their roles

Answer:

Master/Control Plane Components (manage cluster):

1. **API Server**: Exposes Kubernetes API; handles all REST requests
2. **Scheduler**: Assigns pods to nodes based on resource requirements and constraints
3. **Controller Manager**: Runs controller processes (Replication, Endpoints, Namespace, etc.)
4. **etcd**: Key-value store; maintains cluster state and configuration
5. **Cloud Controller Manager**: Integrates with cloud provider APIs (optional)

Node Components (run on each worker node):

1. **Kubelet**: Ensures containers run in pods; communicates with API server
2. **Container Runtime**: Pulls images and runs containers (Docker, containerd, etc.)

3. **kube-proxy**: Maintains network rules; handles service proxy and load balancing

Important Objects:

- **Pod**: Smallest deployable unit; contains one or more containers
- **Deployment**: Manages replicated applications with updates
- **Service**: Exposes pods to internal/external traffic
- **Namespace**: Virtual cluster for resource isolation
- **ConfigMap**: Store non-sensitive configuration data
- **Secret**: Store sensitive data (passwords, tokens)
- **Persistent Volume**: Provide storage to pods

---

13. What is a Kubernetes Pod, and how does it relate to containers?

Answer:

A Pod is the smallest deployable unit in Kubernetes. It's a wrapper around one or more containers (though typically one) that share:

- **Network namespace**: Containers in a pod share a single IP address
- **Storage resources**: Containers can share volumes
- **Configuration**: Environment variables, resource limits
- **Lifecycle**: Containers in a pod start and stop together

Pod anatomy:

```
Pod (IP: 10.0.0.5)
 Container 1 (port 8080)
 Container 2 (port 9000)
 Shared storage volumes
```

Why containers share a pod:

- **Tightly coupled**: Services that must communicate frequently

- **Shared storage**: Containers that need to access the same data
- **Example**: Main application container + logging sidecar container

Relationship to containers:

- One container per pod is the most common pattern
- Multiple containers in a pod = tightly coupled services (use rarely)
- Kubernetes doesn't run containers directly; it runs pods containing containers

---

14. Explain Kubernetes Services and their types

Answer:

A Service is an abstract way to expose applications running in pods. It provides stable networking and load balancing.

Types of Services:

1. **ClusterIP** (default)
   - Exposes service only within the cluster
   - Pods communicate with each other using this service
   - No external access

```
spec:  type: ClusterIP  selector:  app: myapp  ports:  - port: 80  target Port: 8080
```

2. **NodePort**
   - Exposes service on each node's IP at a specific port (30000-32767)
   - External clients can access via NodeIP:NodePort
   - Useful for development/testing

```
spec:  type: NodePort  ports:  - port: 80  targetPort: 8080  nodePort: 30007
```

3. **LoadBalancer**

- Exposes service externally using cloud provider's load balancer
- Each service gets its own external IP
- Used in production for external traffic

```
spec:  type: LoadBalancer  ports:    - port: 80      targetPort: 8080
```

4. **ExternalName**
    - Maps service to DNS name of external system
    - Creates DNS alias inside cluster
    - Used for integrating external services

---

15. What are Kubernetes liveness and readiness probes?

Answer:

Probes are health checks Kubernetes uses to determine pod health and readiness to accept traffic.

Liveness Probe:
- **Purpose**: Determine if a pod is alive and should be restarted
- **Failure behavior**: Pod is restarted if probe fails
- **Use case**: Detect when application is stuck in deadlock or infinite loop
- **Implementation types**: HTTP, TCP, Exec

Readiness Probe:
- **Purpose**: Determine if a pod is ready to accept traffic
- **Failure behavior**: Pod is removed from service endpoints if probe fails (not restarted)
- **Use case**: Prevent traffic routing to pods that are starting up or temporarily unavailable
- **Implementation types**: HTTP, TCP, Exec

Example:

```
apiVersion: v1kind: Podmetadata:  name: myappspec:  containers:  - name: my app    image: myapp:1.0    livenessProbe:      httpGet:        path: /health        por
```

```
t: 8080 initialDelaySeconds: 15 periodSeconds: 20 failureThreshold: 3 readinessProbe: httpGet: path: /ready port: 8080 initialDelaySeconds: 5 periodSeconds: 10
```

CI/CD Pipelines and Automation

16. What is a CI/CD pipeline, and what are its key stages?

Answer:

CI/CD Pipeline is an automated workflow that moves code from development through testing to production deployment.

Key stages:

1. **Source/Version Control**
   - Code is committed to Git repository
   - Triggers pipeline automatically (via webhooks)
2. **Build Stage**
   - Code is compiled
   - Dependencies are resolved
   - Artifacts (JAR, WAR, Docker images) are created
   - Duration: typically 5-15 minutes
3. **Test Stage**
   - **Unit tests**: Validate individual functions
   - **Integration tests**: Verify component interactions
   - **Security tests**: SonarQube, OWASP dependency checks
   - **Code quality**: Checkstyle, PMD analysis
   - **Test coverage**: Ensure adequate code coverage
4. **Build/Artifact Repository**
   - Artifacts stored in repositories (Nexus, Artifactory, ECR)

- Tagged with version and build number
- Used for deployment

5. **Deployment to Staging**
    - Application deployed to staging environment
    - Mimic production environment
    - Smoke tests and end-to-end tests run
6. **Approval/Manual Gate** (for Continuous Delivery)
    - Manual approval by stakeholders
    - Quality gates must pass
7. **Production Deployment**
    - Application deployed to production
    - May use canary, blue-green, or rolling deployment strategies
    - Health checks ensure successful deployment
8. **Monitoring & Logging**
    - Prometheus, Grafana monitor metrics
    - ELK Stack aggregates logs
    - Alerts notify team of issues

---

17. What is Jenkins, and how is it used in DevOps?

Answer:

Jenkins is an open-source automation server used to automate CI/CD pipelines. It coordinates software builds, testing, and deployment.

Key features:

- **Distributed builds**: Run jobs on multiple machines (master-slave architecture)
- **Pipelines**: Define complex workflows using Jenkins Pipeline DSL
- **Extensive plugins**: 1000+ plugins for integration with other tools

- **Scalable**: Easy to add agents for parallel execution
- **Web interface**: User-friendly GUI for job creation and management

Jenkins Pipeline types:

1. **Declarative Pipeline** (recommended):

```
pipeline {  agent any
  stages {    stage('Build') {      steps {        sh 'mvn clean package'      }    }
stage('Test') {      steps {        sh 'mvn test'      }    }    stage('Deploy') {      s
teps {        sh 'kubectl apply -f deployment.yaml'      }    }  }}
```

2. **Scripted Pipeline**:
    - More flexible but harder to read
    - Uses Groovy language directly

Master-Slave Architecture:
- **Master**: Schedules jobs, manages execution
- **Slave/Agent**: Executes jobs, can be distributed across multiple machines

Common use cases:
- Triggering builds on code commits
- Running automated tests
- Building Docker images
- Deploying applications
- Code quality analysis

---

18. Explain Jenkins Declarative and Scripted Pipelines

Answer:

Aspect	Declarative	Scripted
Syntax	YAML-like, structured	Groovy language, imperative
Readability	Easy to read and understand	Complex, harder to read
Learning Curve	Beginner-friendly	Requires Groovy knowledge
Flexibility	Limited but suitable for most cases	Highly flexible for complex scenarios

Aspect	Declarative	Scripted
Error Handling	Built-in error handling	Manual error handling
Community	Recommended by Jenkins team	Still supported but older

Declarative Pipeline Example:

```
pipeline {  agent any
  options {    buildDiscarder(logRotator(numToKeepStr: '10'))  }  parameters {
string(name: 'VERSION', defaultValue: '1.0', description: 'Build version')  }  sta
ges {    stage('Build') {      steps {        sh 'mvn clean package'      }    }    stage
('Test') {      steps {        sh 'mvn test'      }    }  }  post {    success {      echo 'Bu
ild successful!'    }    failure {      echo 'Build failed!'    }  }}
```

Scripted Pipeline Example:

```
node {  try {    stage('Build') {      sh 'mvn clean package'    }    stage('Test') {
sh 'mvn test'    }  } catch (Exception e) {    echo "Pipeline failed: ${e}"  }}
```

---

19. How do you handle secrets and sensitive data in CI/CD pipelines?

Answer:

Secrets must NEVER be hardcoded in code, Dockerfile, or configuration files. Security best practices:

1. Use Secrets Management Tools:

- **HashiCorp Vault**: Centralized secrets management with encryption
- **AWS Secrets Manager**: Native AWS secrets storage and rotation
- **Azure Key Vault**: Microsoft's secrets management service
- **Kubernetes Secrets**: For sensitive data in Kubernetes

2. Environment Variables:

```
// Jenkins: Use Jenkins Credentialspipeline {  stages {    stage('Deploy') {      s
teps {        script {          withCredentials([string(credentialsId: 'DB_PASSWOR
```

```
D', variable: 'DB_PASS')]) {            sh 'docker run -e DB_PASSWORD=$DB_PAS
S myapp:1.0'          }       }     }    }  }}
```

3. Best Practices:
- Rotate secrets regularly (every 30-90 days)
- Use short-lived tokens when possible
- Implement access controls (least privilege principle)
- Audit all secret access
- Use encrypted channels (HTTPS, TLS) for transmission
- Never log sensitive data
- Implement secret scanning in code (detect hardcoded secrets)

4. Secret Scanning Tools:
- **GitGuardian**: Scans Git repositories for leaked secrets
- **TruffleHog**: Searches Git history for credentials
- **OWASP Dependency Check**: Identifies vulnerable dependencies

---

20. Explain Blue-Green and Canary deployments

Answer:

Blue-Green Deployment:
- Maintain two identical production environments: Blue (current) and Green (new)
- Deploy new version to Green environment
- Once validated, switch all traffic from Blue to Green
- Rollback is instant by switching back to Blue
- **Advantages**: Zero downtime, instant rollback, easy to validate
- **Disadvantages**: Requires 2x resources, database synchronization complexity

```
Deployment 1 (Blue)  Traffic
Deployment 2 (Green) ’ Validated

Switch:
Deployment 1 (Blue)
Deployment 2 (Green)  Traffic
```

Canary Deployment:
- Deploy new version to a small subset of users (5-10%)
- Monitor metrics and logs for issues
- Gradually increase traffic to new version (25%, 50%, 100%)
- Rollback only affects that small subset if issues occur
- **Advantages**: Low risk, catches issues with real traffic, gradual rollout
- **Disadvantages**: Slower deployment, complex monitoring

```
Old Version: 100% ′ 95% ′ 80% ′ 50% ′ 0%
New Version:   0% ′  5% ′ 20% ′ 50% ′ 100%
```

Rolling Deployment:
- Replace instances one at a time
- New version gradually replaces old version
- No downtime if service is replicated
- **Kubernetes default strategy**

Infrastructure as Code and Configuration Management

21. What is Terraform, and how does it work?

Answer:

Terraform is an infrastructure as code tool by HashiCorp for provisioning and managing infrastructure across multiple cloud providers.

Key features:

- **Declarative**: Define desired state, not steps to achieve it
- **Cloud-agnostic**: Works with AWS, Azure, GCP, and 300+ providers
- **State management**: Tracks infrastructure state (terraform.tfstate)
- **Plan and apply**: Preview changes before applying

Terraform workflow:

1. **Write**: Define infrastructure in HCL (HashiCorp Configuration Language)
2. **Plan**: Preview what Terraform will create/modify

3. **Apply**: Execute the plan and provision infrastructure

Example Terraform configuration:

```
# Configure AWS provider
provider "aws" {
  region = "us-east-1"
}

# Create S3 bucket
resource "aws_s3_bucket" "my_bucket" {
  bucket = "my-unique-bucket-name"

  tags = {
    Name        = "My Bucket"
    Environment = "Production"
  }
}

# Create EC2 instance
resource "aws_instance" "web_server" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  tags = {
    Name = "WebServer"
  }
}

# Output public IP
output "instance_ip" {
  value = aws_instance.web_server.public_ip
}
```

Terraform state:
- `terraform.tfstate`: JSON file tracking current infrastructure state

- Used for plan calculations and change detection
- Should be stored remotely (S3, Azure Storage) for team collaboration
- Sensitive data stored in state; secure it properly

Common commands:
- `terraform init`: Initialize Terraform directory
- `terraform plan`: Preview changes
- `terraform apply`: Apply changes
- `terraform destroy`: Destroy infrastructure

---

22. Explain Terraform Modules and remote state management

Answer:

Terraform Modules:
- Reusable collections of Terraform configurations
- Encapsulate resources and best practices
- Enable code reuse and standardization
- Similar to functions in programming

Module structure:

```
my-module/
 main.tf        # Core resources
 variables.tf    # Input variables
 outputs.tf      # Output values
 terraform.tfvars # Variable values
```

Using modules:

```
module "vpc" {
  source = "./modules/vpc"

  cidr_block = "10.0.0.0/16"
  name       = "production-vpc"
}

module "app_servers" {
```

```
  source = "terraform-aws-modules/ec2-instance/aws"
  version = "~> 2.0"

  instance_count = 3
  instance_type  = "t2.micro"

  tags = {
    Name = "AppServer"
  }
}
```

Remote State Management:

Why remote state?
- Enables team collaboration
- Secure storage with encryption
- Easier to manage and backup
- Prevents state file loss

Remote state backends:

1. **AWS S3 + DynamoDB**:

```
terraform {
  backend "s3" {
    bucket         = "terraform-state"
    key            = "prod/terraform.tfstate"
    region         = "us-east-1"
    encrypt        = true
    dynamodb_table = "terraform-locks"
  }
}
```

1. **Azure Storage**:

```
terraform {
  backend "azurerm" {
```

```
    resource_group_name  = "rg-terraform"
    storage_account_name = "satstg"
    container_name       = "tfstate"
    key                  = "prod.tfstate"
  }
}
```

1. **Terraform Cloud**:

```
terraform {
  cloud {
    organization = "my-org"
    workspaces {
      name = "production"
    }
  }
}
```

State locking: Prevents concurrent modifications using DynamoDB or blob locks

---

23. What is Ansible, and how is it different from Terraform?

Answer:

Ansible:
- Configuration management and application deployment tool
- **Agentless**: Uses SSH; no agent installation needed
- **Imperative**: Describes steps (how to do it)
- **YAML-based**: Playbooks written in YAML
- **Focuses on**: Server configuration, software installation, application management

Terraform:
- Infrastructure provisioning tool
- Requires provider authentication (API keys)
- **Declarative**: Describes desired end state (what, not how)
- **HCL-based**: Infrastructure defined in HCL
- **Focuses on**: Creating and managing infrastructure resources

Aspect	Ansible	Terraform
Agent	Agentless (SSH)	Requires provider API access
Paradigm	Imperative (steps)	Declarative (state)
State	No state file	Maintains state file
Idempotency	Depends on module	Built-in
Best for	Configuration, deployment	Infrastructure provisioning
Cloud support	Multiple clouds	Multiple clouds

Ansible Playbook Example:

```
---- hosts: webservers  become: yes  tasks:    - name: Install Nginx      apt:        name: nginx        state: present    - name: Start Nginx      service:        name: nginx        state: started        enabled: yes    - name: Deploy application      copy:        src: app/        dest: /var/www/html/
```

Common practice: Use **Terraform for infrastructure provisioning** and **Ansible for configuration management**

---

24. Explain version control branching strategies

Answer:

Branching Strategy defines how teams organize work and manage code releases.

1. Git Flow:
- Multiple long-lived branches
- **master**: Production-ready code
- **develop**: Integration branch for features
- **feature**: New features (branched from develop)
- **release**: Release candidates (bugfixes only)
- **hotfix**: Production emergency fixes

```
master
â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€€
  ‘                              ‘
  â”€ hotfix/1.0.1 â”€â”€â”˜
```

```
        release/1.1 â"€â"€â"€â"€â"€â"
                        "
  develop
  â"€â"€â"€â"€â"€â"€â"€â"€â"€â"€â"€â"€â"€â"€â"€â"€â"´â"€â"€â"€â"€â"€â
    '       '
    â""â"€feature/login â"€â"˜
```

2. GitHub Flow (Simpler):
- **main/master**: Always production-ready
- **feature branches**: For each feature
- Pull requests for code review
- Merge to master after approval
- Deploy immediately after merge

3. Trunk-Based Development:
- Short-lived branches
- Frequent merges to main
- Feature flags for incomplete features
- Suitable for CI/CD maturity

Best practices:
- Descriptive branch names: `feature/user-auth`, `bugfix/login-issue`
- Protect main/master branch (require PR reviews)
- Delete branches after merging
- Keep branches short-lived (1-3 days)

Monitoring and Logging

25. What is Prometheus, and how does it work?

Answer:

Prometheus is an open-source monitoring and alerting toolkit designed for reliability and scalability in microservices architectures.

Architecture:

```
Applications/Services (expose metrics)
        ↓
   Prometheus Server (scrapes)
        ↓
   Time-Series Database (stores)
        ↓
   Alertmanager (alerts)
   PromQL (queries)
```

Key components:

1. **Prometheus Server**:
   - Scrapes metrics from targets at regular intervals
   - Stores time-series data locally
   - Evaluates alert rules
   - Configuration: `prometheus.yml`
2. **Exporters**:
   - Agents that expose metrics in Prometheus format
   - Examples: node_exporter (system), mysql_exporter, blackbox_exporter
   - Targets expose metrics on `/metrics` endpoint
3. **Alertmanager**:
   - Handles alerts from Prometheus
   - Deduplicates, groups, routes alerts
   - Sends notifications (email, Slack, PagerDuty)
4. **Time-Series Database**:
   - Stores metrics as time-series data
   - Data format: `metric_name{label1="value1"} value timestamp`
   - Example: `http_requests_total{job="api",handler="/users"} 1027 1395066363000`

Prometheus configuration:

```
global:  scrape_interval: 15s  evaluation_interval: 15sscrape_configs:  - job_na
me: 'prometheus'    static_configs:      - targets: ['localhost:9090']  - job_nam
e: 'node'    static_configs:      - targets: ['localhost:9100']alerting:  alertmanage
rs:    - static_configs:        - targets: ['localhost:9093']
```

PromQL (Prometheus Query Language):

```
# Request rate over last 5 minutes
rate(http_requests_total[5m])

# Memory usage percentage
(1 - (node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes)) * 1
00

# Top 5 services by error rate
topk(5, rate(http_requests_total{status="500"}[5m]))
```

---

26. What is Grafana, and how does it integrate with Prometheus?

Answer:

Grafana is an open-source visualization and monitoring platform that creates interactive dashboards and alerts.

Key features:
- **Multi-source support**: Prometheus, Elasticsearch, Graphite, InfluxDB, etc.
- **Rich dashboards**: Create beautiful, customizable dashboards
- **Alerting**: Set thresholds and send notifications
- **User management**: Role-based access control
- **Plugins**: Extend functionality

Grafana + Prometheus integration:

1. **Add Prometheus as data source**:
   - Configuration ′ Data Sources ′ Add Prometheus
   - Set URL: `http://prometheus:9090`

2. **Create dashboards**:
    - Add panels with Prometheus queries
    - Use PromQL for querying
3. **Example dashboard**:

```
Panel 1: CPU Usage
Query: rate(node_cpu_seconds_total[5m]) * 100

Panel 2: Memory Usage
Query: (1 - (node_memory_MemAvailable_bytes / node_memory_MemTotal_by
tes)) * 100

Panel 3: Disk Space
Query: (1 - (node_filesystem_avail_bytes / node_filesystem_size_bytes)) * 100

Panel 4: Network Traffic
Query: rate(node_network_receive_bytes_total[5m])
```

Alert creation:

```
Alert Conditions:
- Name: High CPU Usage
- Condition: CPU > 80%
- Duration: 5 minutes
- Action: Send to Slack, email, PagerDuty
```

---

27. Explain the ELK Stack (Elasticsearch, Logstash, Kibana)

Answer:

ELK Stack is a popular log management and analysis solution for centralized logging.

Components:

1. **Elasticsearch**:

- Distributed search and analytics engine
- Stores logs as JSON documents
- Provides full-text search and aggregation capabilities
- Highly scalable and performant

2. **Logstash**:
    - Data processing and ingestion engine
    - Collects logs from various sources
    - Parses, filters, and enriches data
    - Ships logs to Elasticsearch or other destinations
3. **Kibana**:
    - Visualization and exploration platform
    - Creates interactive dashboards
    - Performs ad-hoc queries on logs
    - Real-time monitoring and alerting

Architecture:

```
Applications/Services (generate logs)
        ‟
Logstash (collect, parse, filter)
        ‟
Elasticsearch (store, index)
        ‟
Kibana (visualize, explore)
```

Example Logstash configuration:

```
input {
  file {
    path ⇒ "/var/log/app/*.log"
    start_position ⇒ "beginning"
```

```
  }
}

filter {
  grok {
    match ⇒ { "message" ⇒ "%{COMBINEDAPACHELOG}" }
  }
  date {
    match ⇒ [ "timestamp", "dd/MMM/yyyy:HH:mm:ss Z" ]
  }
}

output {
  elasticsearch {
    hosts ⇒ ["localhost:9200"]
    index ⇒ "logs-%{+YYYY.MM.dd}"
  }
}
```

Use cases:
- Centralized logging for microservices
- Troubleshooting production issues
- Security monitoring and SIEM
- Business analytics and insights

---

28. What are the key DevOps metrics and KPIs?

Answer:

Four Golden Signals (Google SRE best practices):

1. **Latency**:
    - Time to serve a request
    - **Good**: < 100ms for most operations
    - Measure end-to-end response time
2. **Traffic**:

- Volume of requests
- Queries per second (QPS), requests per minute (RPM)
- Indicates load on system

3. **Errors**:
   - Number and rate of failed requests
   - **Good**: < 0.1% error rate
   - 5xx errors, 4xx errors, timeout errors
4. **Saturation**:
   - How full the system is
   - CPU, memory, disk, network utilization
   - **Good**: Operating at 60-70% capacity

DORA Metrics (Deployment frequency):

1. **Deployment Frequency (DF)**:
   - How often code is deployed
   - **Elite performers**: Multiple deployments per day
   - **Low performers**: Once per month or less
2. **Lead Time for Changes (LTC)**:
   - Time from commit to production
   - **Elite**: < 1 hour
   - **Low**: > 1 month
3. **Mean Time to Recovery (MTTR)**:
   - Average time to restore service after incident
   - **Elite**: < 1 hour
   - **Low**: > 1 day
4. **Change Failure Rate (CFR)**:
   - Percentage of deployments causing issues

- **Elite**: 0-15%
- **Low**: > 46%

Other important metrics:
- **Uptime/Availability**: 99.9%, 99.95%, 99.99%
- **Test Coverage**: Code coverage percentage
- **Build Success Rate**: Percentage of successful builds
- **Time to Resolution**: Time to fix security vulnerabilities
- **Cost per deployment**: Infrastructure and operational costs

---

Cloud Platforms and DevOps

29. Explain AWS services commonly used in DevOps

Answer:

Compute:
- **EC2**: Virtual machines with full control
- **ECS**: Container orchestration (Docker)
- **EKS**: Managed Kubernetes service
- **Lambda**: Serverless functions

CI/CD:
- **CodePipeline**: Orchestrates CI/CD workflows
- **CodeBuild**: Compile, test, build code
- **CodeDeploy**: Deploy to various compute services
- **CodeCommit**: Git repository service

Infrastructure & Configuration:
- **CloudFormation**: Infrastructure as Code (AWS-specific)
- **Systems Manager**: Parameter Store for configuration
- **OpsWorks**: Configuration management (Chef, Puppet)

Monitoring & Logging:
- **CloudWatch**: Metrics, logs, and alarms
- **X-Ray**: Distributed tracing
- **CloudTrail**: API audit logging

Storage:
- **S3**: Object storage
- **EBS**: Block storage for EC2
- **EFS**: Elastic file system
- **Glacier**: Long-term archive storage

Database:
- **RDS**: Managed relational database
- **DynamoDB**: NoSQL database
- **ElastiCache**: In-memory caching

Networking:
- **VPC**: Virtual private cloud
- **ALB/NLB**: Load balancers
- **Route 53**: DNS service
- **CloudFront**: CDN

Example AWS DevOps workflow:

```
Code committed to CodeCommit
    ↓
CodePipeline triggered
    ↓
CodeBuild: Build and test
    ↓
CodeBuild: Build Docker image
    ↓
Push to ECR (EC2 Container Registry)
    ↓
CodeDeploy: Deploy to EKS/ECS/EC2
    ↓
CloudWatch: Monitor metrics and logs
    ↓
Auto Scaling: Scale based on demand
```

---

30. What is auto-scaling, and how do you configure it?

Answer:

Auto-scaling automatically adjusts the number of running instances based on demand to ensure optimal performance and cost efficiency.

Types of scaling:

1. **Horizontal Scaling (Scale Out/In)**:
    - Add or remove instances
    - Handles sudden traffic spikes
    - Better for stateless applications
    - Example: Add more web servers
2. **Vertical Scaling (Scale Up/Down)**:
    - Increase/decrease instance size
    - Requires downtime
    - Limited by max instance type
    - Example: t2.micro ′ t2.large

AWS Auto Scaling Example:

1. **Create Launch Template**:

```
Specify: AMI ID, instance type, security group, key pair
```

1. **Create Auto Scaling Group**:

```
Auto Scaling Group:  - Min capacity: 2  - Desired capacity: 3  - Max capacity: 10  - Launch template: specified above
```

1. **Create Scaling Policies**:

```
Policy 1 - Scale Up:  - Metric: Average CPU > 70%  - Duration: 2 minutes  - Action: Add 1 instancePolicy 2 - Scale Down:  - Metric: Average CPU < 30%  - Duration: 5 minutes  - Action: Remove 1 instance
```

Kubernetes Auto-scaling:

```
apiVersion: autoscaling/v2kind: HorizontalPodAutoscalermetadata:  name: ap
p-hpaspec:  scaleTargetRef:  apiVersion: apps/v1  kind: Deployment  name:
myapp  minReplicas: 2  maxReplicas: 10  metrics:  - type: Resource  resourc
e:  name: cpu  target:  type: Utilization  averageUtilization: 70  - ty
pe: Resource  resource:  name: memory  target:  type: Utilization
averageUtilization: 80
```

Advanced DevOps Concepts

31. Explain disaster recovery and backup strategies

Answer:

Disaster Recovery (DR) is a plan to restore systems and data after catastrophic failures.

Key metrics:

1. **RTO (Recovery Time Objective)**:
    - Maximum acceptable downtime
    - Example: 1 hour
    - Depends on business criticality
2. **RPO (Recovery Point Objective)**:
    - Maximum acceptable data loss
    - Example: 15 minutes
    - Determines backup frequency

DR Strategies:

1. **Backup and Restore** (Cheapest, slowest):
    - RTO: Hours to days
    - RPO: Hours to days

- Suitable for: Non-critical systems

2. **Pilot Light** (Standby):
    - Minimal infrastructure running in DR site
    - RTO: Minutes to hours
    - RPO: Minutes
    - Cost-effective middle ground
3. **Warm Standby** (Running but scaled down):
    - Reduced-capacity infrastructure in DR site
    - RTO: Seconds to minutes
    - RPO: Seconds
    - Good balance of cost and speed
4. **Hot Standby** (Active-Active):
    - Fully operational duplicate infrastructure
    - RTO: Seconds
    - RPO: Seconds
    - Most expensive but highest availability

Backup strategies:

1. **Full Backup**:
    - Copy all data
    - Takes longest, uses most space
    - Frequency: Weekly
2. **Incremental Backup**:
    - Only changed data since last backup
    - Fast, uses least space
    - Frequency: Daily
3. **Differential Backup**:

- Changed data since last full backup
- Balance between full and incremental
- Frequency: Daily

AWS DR Implementation:

```
Primary Region (Active)
    " (Continuous replication)
Secondary Region (Standby)

On failure: Route 53 fails over to secondary region
```

---

32. What is DevSecOps, and how is it implemented?

Answer:

DevSecOps integrates security practices throughout the entire DevOps pipeline rather than adding security at the end.

Principles:
- **Shift left**: Move security testing earlier in development
- **Automation**: Automate security checks
- **Collaboration**: Security team works with dev and ops teams
- **Continuous**: Security is ongoing, not a phase

Implementation strategies:

1. **Secure Code Development**:
    - Code review process
    - Static Application Security Testing (SAST): SonarQube, Checkmarx
    - Dependency scanning: OWASP Dependency Check, Snyk
2. **Container Security**:
    - Image scanning: Trivy, Aqua
    - Registry security: Private registries, access controls
    - Runtime security: Falco monitors container behavior

3. **Infrastructure Security**:
    - IaC scanning: Terraform plan review, Checkov
    - Secrets management: HashiCorp Vault, AWS Secrets Manager
    - Network security: Security groups, NACLs, WAF
4. **CI/CD Pipeline Security**:
    - Secure Jenkins: RBAC, disable script approval
    - Secrets in pipeline: Use credential stores
    - Artifact signing: Verify image integrity
5. **Compliance & Monitoring**:
    - Policy as Code: Terraform policy checks
    - Compliance scanning: CIS benchmarks
    - Security monitoring: SIEM tools, threat detection

Example DevSecOps Pipeline:

```
Code Commit
    ↓
SAST (SonarQube)
    ↓
Dependency Check (Snyk)
    ↓
Build (Docker build)
    ↓
Container Scan (Trivy)
    ↓
IaC Check (Checkov)
    ↓
Deploy (with security policies)
    ↓
Runtime Security (Falco)
```

```
"
Compliance Check (CIS Benchmark)
```

33. Explain incident management and post-mortems in DevOps

Answer:

Incident Management is the process of handling production outages and critical issues.

Incident severity levels:

1. **SEV 1** (Critical):
    - Complete service outage
    - Affects all users
    - Response: Immediate, CEO notified
2. **SEV 2** (High):
    - Major functionality broken
    - Affects subset of users
    - Response: < 30 minutes
3. **SEV 3** (Medium):
    - Minor functionality issue
    - Workaround available
    - Response: < 2 hours
4. **SEV 4** (Low):
    - Cosmetic issue
    - No user impact
    - Response: Next working day

Incident response process:

1. **Detection**: Monitoring alerts or user reports

2. **Classification**: Determine severity
3. **Initial Response**: Engage on-call team
4. **Triage**: Identify scope and impact
5. **Mitigation**: Apply temporary fix or workaround
6. **Resolution**: Permanently fix root cause
7. **Communication**: Update stakeholders continuously

Post-Mortem Meeting:

Purpose: Learn from incidents to prevent recurrence

Best practices:
- Blameless culture: Focus on systems, not people
- Held within 24-48 hours of incident
- Include: Incident lead, engineers, product, leadership
- Document: Timeline, root cause, action items

Post-Mortem structure:

1. What happened? (Timeline of events)
2. Why did it happen? (Root cause analysis)
3. What did we learn? (Key insights)
4. What will we do? (Action items with owners)
5. Follow-up: Track action items to completion

---

34. What is GitOps, and why is it important?

Answer:

GitOps is a methodology where Git serves as the single source of truth for infrastructure and application deployment.

Core principles:

1. **Everything in Git**: Infrastructure, configurations, policies defined in version-controlled files
2. **Single Source of Truth**: Git is the authority for desired state

3. **Pull-based Deployment**: System automatically syncs to match Git state
4. **Declarative Configuration**: Specify desired state, not steps

GitOps workflow:

```
Developer creates PR
    ↓
Code review and approval
    ↓
Merge to main branch
    ↓
GitOps operator (ArgoCD, Flux) detects change
    ↓
Automatically sync to cluster
    ↓
Application deployed
```

Popular GitOps tools:

1. **ArgoCD**:
    - Kubernetes-native
    - Automatic sync to Git state
    - Web UI for visibility
2. **Flux**:
    - CNCF project
    - GitOps toolkit
    - Helm and Kustomize support

Benefits:
- **Version control**: Complete audit trail of all changes
- **Rollback**: Easy rollback to previous Git commit
- **Compliance**: Approved changes only
- **Disaster recovery**: Infrastructure easily recreated from Git
- **Collaboration**: Pull requests for infrastructure changes

Example ArgoCD Application:

```
apiVersion: argoproj.io/v1alpha1kind: Applicationmetadata:  name: myapp  nam
espace: argocdspec:  project: default  source:    repoURL: https://github.com/
myorg/myapp    targetRevision: main    path: k8s/  destination:    server: http
s://kubernetes.default.svc    namespace: default  syncPolicy:    automated:
prune: true      selfHeal: true
```

---

35. Explain observability vs. monitoring

Answer:

Monitoring:
- Collecting and alerting on predefined metrics
- You decide what to measure in advance
- “Tell me if X is happening”
- Examples: CPU > 80%, Memory > 90%, Response time > 2s

Observability:
- Ability to understand system state from external outputs
- Answers arbitrary questions about system behavior
- “Tell me why X happened”
- Includes metrics, logs, traces

Observability consists of three pillars:

1. **Metrics**:
    - Numerical measurements over time
    - Tools: Prometheus, Datadog, New Relic
    - Examples: CPU usage, request rate, latency
2. **Logs**:
    - Detailed records of events
    - Tools: ELK Stack, Splunk, Google Cloud Logging
    - Examples: Application errors, API calls, system events
3. **Traces**:

- End-to-end flow through distributed system
- Tools: Jaeger, Datadog, Zipkin
- Examples: Request path through microservices, timing of each service

Observability vs. Monitoring:

Aspect	Monitoring	Observability
Approach	Know-what-to-monitor	Ask any question
Metrics	Predefined	Cardinality unlimited
Response	Alert on condition	Investigate any issue
Proactive	Limited	Excellent
Reactive	Good	Excellent
Cost	Lower	Higher (more data)

Building observable systems:
- Structured logging with context
- Distributed tracing for request flow
- Rich metrics with high cardinality labels
- Correlation IDs across services

---

36. What is a service mesh, and why use it?

Answer:

Service Mesh is an infrastructure layer that manages service-to-service communication in microservices architectures.

Architecture:
- **Control Plane**: Manages configuration and policies
- **Data Plane**: Network proxies (sidecars) that route traffic
- Sidecar proxy runs alongside each service

Popular service meshes:
- **Istio**: Full-featured, most popular
- **Linkerd**: Lightweight, focused
- **Consul**: From HashiCorp

Key capabilities:

1. **Traffic Management**:
    - Load balancing between services
    - Retry logic and timeouts
    - Circuit breakers
2. **Security**:
    - Automatic mutual TLS (mTLS) between services
    - Fine-grained access policies
    - Certificate management
3. **Observability**:
    - Automatic metrics collection
    - Request tracing
    - Service dependency visualization
4. **Resilience**:
    - Rate limiting
    - Bulkheads to isolate failures
    - Canary deployments

Istio example configuration:

```
apiVersion: networking.istio.io/v1beta1kind: VirtualServicemetadata:  name: m
yappspec:  hosts:  - myapp  http:  - match:    - uri:       prefix: "/api"    route:
- destination:       host: myapp       port:         number: 8080       subset: v1
weight: 80    - destination:       host: myapp       port:         number: 8080
subset: v2      weight: 20---apiVersion: networking.istio.io/v1beta1kind: Destin
ationRulemetadata:  name: myappspec:  host: myapp  trafficPolicy:    outlierDe
tection:      consecutiveErrors: 5      interval: 30s      baseEjectionTime: 30s  su
bsets:  - name: v1    labels:      version: v1  - name: v2    labels:      version: v2
```

37. Explain rolling, blue-green, and canary deployments with Kubernetes

Answer:

Rolling Deployment (Kubernetes default):
- Replace pods one at a time
- Old version gradually replaced with new
- No downtime with proper health checks
- Slower deployment but reduces risk
- If issue detected, old pods can still serve traffic

```
apiVersion: apps/v1kind: Deploymentmetadata:  name: myappspec:  replicas:  3  strategy:  type: RollingUpdate  rollingUpdate:  maxSurge: 1  # One e xtra pod during update  maxUnavailable: 1  # One pod can be unavailable  s elector:  matchLabels:  app: myapp  template:  metadata:  labels:  a pp: myapp  spec:  containers:  - name: myapp  image: myapp:v2  readinessProbe:  httpGet:  path: /ready  port: 8080  initia lDelaySeconds: 10  periodSeconds: 5
```

Blue-Green Deployment:
- Maintain two full environments
- Deploy new version to “green”
- Once validated, switch all traffic to green
- Blue environment becomes standby
- Instant rollback possible

Implementation with Kubernetes:

```
# Green deployment (new version)apiVersion: apps/v1kind: Deploymentmetad ata:  name: myapp-greenspec:  replicas: 3  selector:  matchLabels:  app: m yapp  version: green  template:  metadata:  labels:  app: myapp  v ersion: green  spec:  containers:  - name: myapp  image: myapp:v2--- -# Service initially points to blueapiVersion: v1kind: Servicemetadata:  name:  myappspec:  selector:  app: myapp  version: blue  # Initially pointing to blue ports:  - port: 80  targetPort: 8080# Update selector to "green" to switch traf fic
```

Canary Deployment:
- Gradually roll out new version
- Small percentage of traffic to new version
- Monitor metrics and error rates
- Expand to larger percentage if healthy
- Automatic rollback if issues detected

```
apiVersion: argoproj.io/v1alpha1kind: Rolloutmetadata:  name: myappspec:  replicas: 5  strategy:  canary:  steps:  - weight: 10  pause: {duration: 30s}  - weight: 50  pause: {duration: 30s}  - weight: 100  selector:  matchLabels:  app: myapp  template:  metadata:  labels:  app: myapp  spec:  containers:  - name: myapp  image: myapp:v2
```

---

38. What is immutable infrastructure, and why is it important?

Answer:

Immutable Infrastructure means infrastructure components (servers, containers) are never modified after deployment. Instead of updating, they're replaced entirely.

Principles:
- No SSH into servers to make changes
- No manual configuration modifications
- No in-place patches or updates
- Everything is codified and versioned

Traditional approach (mutable):

```
Server created
    "
SSH into server
    "
Install packages manually
    "
Configure manually
    "
```

Over time: Different configurations, unknown state
↓
Difficult to reproduce or troubleshoot

Immutable approach:

Infrastructure as Code (Terraform, CloudFormation)
↓
Container Image (Docker)
↓
Deploy image
↓
Need change? Create new image
↓
Deploy new image, kill old one
↓
Always known, reproducible state

Benefits:
- **Consistency**: Every deployment is identical
- **Reproducibility**: Easy to create identical environments
- **Reliability**: Eliminates configuration drift
- **Disaster recovery**: Quickly recreate from image
- **Testing**: Can fully test before deployment
- **Security**: Fewer attack vectors (no SSH access)
- **Simplicity**: Easier troubleshooting (replace vs. debug)

Implementation:

1. **Golden Image**:
    - Pre-built image with all dependencies
    - Version controlled
    - Used as base for deployments
2. **Container approach**:

```
FROM ubuntu:20.04
RUN apt-get update && apt-get install -y \    nginx \    curl \    git
COPY nginx.conf /etc/nginx/nginx.conf
CMD ["nginx", "-g", "daemon off;"]
```

3. **Deployment**:
    - Deploy image (never modify running container)
    - Configuration via environment variables or ConfigMaps
    - To update: Build new image, deploy new container, remove old

---

39. Explain high availability and redundancy architecture

Answer:

High Availability (HA) is the ability of a system to remain operational even when components fail.

Redundancy strategies:

1. **Horizontal Redundancy** (Multiple servers):

```
Load Balancer
 Server 1
 Server 2
 Server 3

If Server 1 fails, traffic routes to Servers 2 & 3
```

2. **Vertical Redundancy** (Backup components):
    - Primary database + replica database
    - Primary power supply + backup power supply
    - Primary network connection + backup connection
3. **Geographic Redundancy** (Multi-region):

```
Region A (Primary)
  “ (Replication)
Region B (Secondary)

Route 53 fails over if Region A is down
```

HA architecture example (AWS):

```
Internet
  “
Route 53 (DNS failover)
  “
  â”œâ”€ ALB (us-east-1a)
  â”‚   EC2 Instance 1
  â”‚   EC2 Instance 2
  â”‚   EC2 Instance 3
  â”‚
  â”””â”€ ALB (us-east-1b)
      EC2 Instance 4
      EC2 Instance 5
      EC2 Instance 6

Database: RDS Multi-AZ
  Primary (us-east-1a)
    “ (synchronous replication)
  Standby (us-east-1b)

Cache: ElastiCache Multi-AZ
  Primary ’ Replica
```

Kubernetes HA setup:

```
apiVersion: apps/v1kind: Deploymentmetadata:  name: myapp-haspec:  replic
as: 3  # Multiple replicas across nodes  selector:  matchLabels:  app: myap
p  template:  metadata:  labels:  app: myapp  spec:  affinity:  pod
```

```
AntiAffinity: preferredDuringSchedulingIgnoredDuringExecution: - weight: 100 podAffinityTerm: labelSelector: matchExpressions: - key: app operator: In values: - myapp topologyKey: kubernetes.io/hostname containers: - name: myapp image: myapp:latest livenessProbe: httpGet: path: /health port: 8080 initialDelaySeconds: 15 periodSeconds: 20
```

HA metrics:
- **Uptime**: Percentage of time service is available
- **MTBF**: Mean Time Between Failures
- **MTTR**: Mean Time To Recovery
- **MTTA**: Mean Time To Acknowledge

---

40. What is cost optimization in DevOps?

Answer:

Cost Optimization involves using resources efficiently to reduce cloud spending without compromising performance.

Strategies:

1. **Right-sizing**:
   - Analyze actual usage patterns
   - Use smaller instance types if possible
   - AWS Compute Optimizer provides recommendations
   - Saves 20-30% of compute costs
2. **Reserved Instances & Savings Plans**:
   - AWS Reserved Instances: 40-70% discount for 1-3 year commitment
   - Savings Plans: Flexible pricing for compute/database
   - Identify predictable, long-term workloads
3. **Spot Instances**:
   - 70-90% discount for spare capacity

- Use for fault-tolerant, batch jobs
- Can be interrupted (risk/reward)

4. **Auto-scaling**:
    - Scale down during low traffic
    - Scale up during peaks
    - Avoid paying for unused capacity
5. **Resource Cleanup**:
    - Delete unused instances, volumes, snapshots
    - Unattach unused Elastic IPs
    - Remove unused security groups
    - Can save 30-40% through cleanup
6. **Data Transfer Optimization**:
    - Use VPC endpoints instead of NAT Gateway
    - CloudFront CDN for static content
    - Data transfer costs often overlooked
7. **Database Optimization**:
    - Use managed services (RDS, DynamoDB) vs. self-managed
    - Multi-AZ only when needed
    - Read replicas for scaling reads
    - Archive old data to Glacier
8. **Monitoring & Chargeback**:
    - AWS Cost Explorer for visibility
    - Tagging for cost allocation
    - Departmental chargeback
    - Identify cost anomalies early

Cost optimization workflow:

Monitor Spend
  ”
Identify expensive resources
  ”
Analyze usage patterns
  ”
Apply optimization strategies
  ”
Verify cost reduction
  ”
Document learnings

Tools:
- AWS Cost Explorer
- AWS Trusted Advisor
- AWS Compute Optimizer
- CloudHealth, Flexera (third-party)

---

41. Explain the concept of scalability vs. reliability

Answer:

Scalability:
- Ability of system to handle increased load
- Adding more resources (horizontal) or larger resources (vertical)
- Question: “Can we handle 10x more traffic?”

Types of scalability:

1. **Horizontal Scaling** (scale-out):
   - Add more machines/servers
   - Better for modern distributed systems
   - Example: Add web servers behind load balancer
2. **Vertical Scaling** (scale-up):
   - Increase capacity of existing machine

- Easier to implement but has limits
- Example: t2.micro ′ t2.large

Reliability:
- Ability of system to function correctly over time
- Probability that system performs as designed
- Question: “Can we trust this system?”

Reliability measures:
- **MTBF** (Mean Time Between Failures): How long until next failure
- **MTTR** (Mean Time To Recovery): How long to fix when it breaks
- **Availability**: (MTBF / (MTBF + MTTR)) — 100%

Relationship:

Scalability | Reliability

A system can scale but be unreliable
A system can be reliable but not scalable

Both are necessary for production systems

Example:

Scalable but unreliable:
- Can handle 10x traffic
- But breaks under load (bad error handling, memory leaks)

Reliable but not scalable:
- Works great for current traffic
- Breaks when traffic doubles

Scalable AND reliable:
- Handles 10x traffic smoothly
- Graceful degradation when overwhelmed
- Health checks and recovery mechanisms

Best practices for both:
- Load testing to understand limits
- Horizontal scaling architecture
- Stateless design for easier scaling
- Health checks and auto-recovery
- Circuit breakers and rate limiting
- Monitoring and alerting

---

42. What is chaos engineering, and why is it important?

Answer:

Chaos Engineering is the practice of intentionally injecting failures into systems to test their resilience and discover weaknesses.

Philosophy: "Break things on purpose to find problems before users do"

Key principles:
1. **Define baseline**: Establish normal system behavior
2. **Form hypothesis**: Predict what will happen if X fails
3. **Inject chaos**: Intentionally cause failure
4. **Monitor**: Observe system response
5. **Recover**: Stop experiment and verify recovery
6. **Learn**: Document findings and improve

Common chaos experiments:

1. **Kill random pods** (Kubernetes):

   ```
   # Randomly terminate podskubectl delete pod -n production <random-po
   d> --grace-period=0
   ```

2. **Inject latency**:
   - Add 500ms delay to all database queries
   - Verify system handles slow responses
3. **Increase error rate**:
   - Make 10% of API calls fail

- Check error handling and fallbacks

4. **CPU stress**:
    - Max out CPU on instance
    - See if auto-scaling triggers, check if graceful
5. **Network partition**:
    - Simulate network split
    - Test distributed system consensus
6. **Disk full**:
    - Fill up disk space
    - Check if system fails gracefully

Chaos engineering tools:
- **Chaos Monkey**: Netflix tool for random failure injection
- **Gremlin**: Commercial chaos engineering platform
- **Litmus**: Kubernetes chaos experiments
- **kube-chaos**: Kubernetes chaos toolkit
- **Locust**: Load testing and chaos injection

Benefits:
- **Identify weaknesses**: Find issues before production failure
- **Improve resilience**: System better handles failures
- **Confidence**: Team knows system can handle real failures
- **Documentation**: Discover actual vs. expected behavior
- **Reduced MTTR**: Practice recovery procedures

Example Litmus experiment:

```
apiVersion: litmuschaos.io/v1alpha1kind: ChaosEnginemetadata:  name: pod-delete-chaosspec:  appinfo:  appns: default  applabel: app=myapp  engineState: "active"  chaosServiceAccount: litmus-admin  experiments:  - name: pod-delete spec:  components:  env:  - name: TOTAL_CHAOS_DURATION  value: "15"  - name: CHAOS_INTERVAL  value: "5"  - name: FORCE  value: "false"
```

43. Explain logging strategies and best practices

Answer:

Logging is the practice of recording events and system behavior for troubleshooting and monitoring.

Logging levels:

1. **DEBUG**: Detailed information for troubleshooting
2. **INFO**: General informational messages
3. **WARN**: Warning messages, something unexpected
4. **ERROR**: Error conditions that need attention
5. **FATAL**: System-breaking errors

Structured logging:

Traditional logging (hard to parse):

```
2024-01-15 10:30:45 User login failed
```

Structured logging (easy to parse and query):

```
{  "timestamp": "2024-01-15T10:30:45.123Z",  "level": "ERROR",  "service": "a
uth-service",  "user_id": "12345",  "event": "login_failed",  "reason": "invalid_p
assword",  "client_ip": "192.168.1.1",  "duration_ms": 245}
```

Logging best practices:

1. **Use structured logging format** (JSON recommended)
2. **Include context**: Request IDs, user IDs, trace IDs
3. **Log at appropriate levels**: Don’t log everything at INFO
4. **Avoid sensitive data**: Never log passwords, API keys, tokens
5. **Use correlation IDs**: Track requests across services
6. **Centralize logs**: Use ELK, Splunk, or cloud provider
7. **Set retention policies**: Balance storage cost vs. compliance

8. **Monitor log volume**: High log volume indicates issues

Example structured logging (Python):

```
import logging
import json
class JSONFormatter(logging.Formatter):
    def format(self, record):
        log_data = {
            "timestamp": record.created,
            "level": record.levelname,
            "service": "user-service",
            "message": record.getMessage(),
            "module": record.module,
        }
        return json.dumps(log_data)
logger = logging.getLogger()
handler = logging.StreamHandler()
handler.setFormatter(JSONFormatter())
logger.addHandler(handler)
# Usagelogger.info("User login successful", extra={"user_id": 123})
logger.error("Database connection failed", extra={"error": "timeout"})
```

Log aggregation with ELK:

```
Application logs
    ”
Logstash (collect, parse, filter)
    ”
Elasticsearch (index, store)
    ”
Kibana (visualize, query)
```

---

44. What is the difference between horizontal and vertical pod autoscaling in Kubernetes?

Answer:

HPA (Horizontal Pod Autoscaler):
- Scales the number of pod replicas
- Adds or removes pods based on metrics
- Good for handling traffic spikes
- Requires multiple replicas for effectiveness

```
apiVersion: autoscaling/v2kind: HorizontalPodAutoscalermetadata:  name: ap
p-hpaspec:  scaleTargetRef:  apiVersion: apps/v1  kind: Deployment  name:
myapp  minReplicas: 2  maxReplicas: 10  metrics:  - type: Resource  resourc
e:  name: cpu  target:  type: Utilization  averageUtilization: 70  - ty
pe: Resource  resource:  name: memory  target:  type: Utilization
averageUtilization: 80
```

VPA (Vertical Pod Autoscaler):
- Scales resource requests and limits of individual pods
- Changes CPU/memory allocation
- Causes pod restarts when adjusting
- Better for right-sizing applications

```
apiVersion: autoscaling.k8s.io/v1kind: VerticalPodAutoscalermetadata:  name:
app-vpaspec:  targetRef:  apiVersion: "apps/v1"  kind: Deployment  name:
myapp  updatePolicy:  updateMode: "Auto"  # or "Off", "Initial", "Recreate"  r
esourcePolicy:  containerPolicies:  - containerName: "*"  minAllowed:
cpu: 100m  memory: 128Mi  maxAllowed:  cpu: 2  memory: 2Gi
```

Aspect	HPA	VPA
What scales	Number of pods	Resource per pod
When to use	Traffic spikes, horizontal scaling	Right-sizing, resource optimization
Restart pods	No	Yes
Recommendation	Add more pod replicas	Increase/decrease CPU/memory
Good for	Stateless applications	Long-running processes
Combination	Can use both together	Can use both together

45. Explain configuration management in DevOps

Answer:

Configuration Management is the process of managing and tracking infrastructure and application configurations.

Challenges without CM:
- Configuration drift (servers become inconsistent)
- Difficult to reproduce environments
- Manual changes hard to track
- Hard to roll back changes
- Scalability problems

Configuration management approaches:

1. **Agentless** (Ansible):
    - Uses SSH for remote execution
    - No agent installation required
    - Simpler setup but less control
    - Idempotent playbooks
2. **Agent-based** (Chef, Puppet):
    - Install agent on every server
    - More control and flexibility
    - Better for complex environments
    - Regular reporting to server

Key concepts:

1. **Idempotency**:
    - Running same configuration multiple times produces same result
    - Important for reliability
2. **Convergence**:
    - System gradually reaches desired state

- Not immediate (like IaC)

3. **Configuration Drift**:
   - Actual state differs from desired state
   - Detected and corrected by CM tools

CM best practices:

1. **Version control** all configurations
2. **Declarative** approach preferred
3. **Test** configurations before deployment
4. **Document** all changes
5. **Automate** as much as possible
6. **Audit** configuration changes
7. **Remediate** drift automatically

Tools comparison:

Tool	Type	Paradigm	Best for
Ansible	Agentless	Imperative	Simple deployments
Chef	Agent	Imperative	Complex environments
Puppet	Agent	Declarative	Large scale
SaltStack	Agent	Declarative	Event-driven

Scenario-Based Questions

46. How would you troubleshoot a slow application in production?

Answer:

Systematic troubleshooting approach:

Step 1: Identify symptoms
- Check dashboards (Grafana)
- Verify metrics: CPU, memory, disk, network

- Check error logs in Kibana/ELK
- Get user reports: Which features are slow?

Step 2: Check application metrics

```
Prometheus queries:
- HTTP request latency: rate(http_request_duration_seconds_sum[5m]) / rate
(http_request_duration_seconds_count[5m])
- Request rate: rate(http_requests_total[5m])
- Error rate: rate(http_requests_total{status=~"5.."}[5m])
- Database query time: database_query_duration_seconds
```

Step 3: Check infrastructure

- CPU usage: `top`, `htop`
- Memory usage: `free -m`
- Disk I/O: `iostat`, `vmstat`
- Network: `netstat`, `ss`, check for connection limits
- Database connection pool exhaustion

Step 4: Check application logs

```
Common patterns:
- Timeout errors: Maybe upstream service is slow
- Database errors: Connection pool issues
- OOM errors: Memory leak or misconfiguration
- GC pauses: Java garbage collection taking too long
```

Step 5: Distributed tracing

- Use Jaeger/Zipkin to trace request
- Identify which service is slow
- See latency breakdown per service call

Step 6: Database investigation

```
-- Check slow queriesSELECT query, time, lock_time FROM slow_query_log;
-- Check active queriesSHOW PROCESSLIST;
```

```
-- Check query planEXPLAIN SELECT ...;
-- Check indexesSHOW INDEX FROM table_name;
```

Step 7: Profile if necessary
- Java: JProfiler, YourKit
- Python: cProfile, line_profiler
- Node.js: Node Inspector
- Focus on hot spots

Step 8: Check recent changes
- Deploy a new version recently?
- Database schema changes?
- Configuration changes?
- Increased load?

Resolution examples:
- Missing database index
- Connection pool too small
- Memory leak in application
- Slow upstream service call
- Misconfigured caching

---

47. How would you recover from a database corruption?

Answer:

Immediate actions (during incident):

1. **Stop writes** to prevent further corruption

```
-- MySQLFLUSH TABLES WITH READ LOCK;
-- PostgreSQLALTER TABLE table_name SET (autovacuum_enabled = false);
```

1. **Take backup** immediately (frozen state)

```
mysqldump -u root -p --all-databases > backup.sql
```

1. **Assess impact**:

- Which tables affected?
- How much data corrupted?
- How long without service?

Recovery options:

Option 1: Restore from backup (fastest if recent)

```
# Restore from backupmysql -u root -p < backup.sql
# Verify data integrityCHECKSUM TABLE table_name;
```

Option 2: Use point-in-time recovery (PITR)

```
# MySQL with binary logs# Find position where corruption occurredSHOW M
ASTER STATUS;# Restore to before corruptionmysqlbinlog /var/log/mysql/my
sql-bin.000003 \  --stop-position=123456 | \  mysql -u root -p
```

Option 3: Restore from replica (fastest)

```
# If have replica with good data# Promote replica to master# Reconfigure othe
r replicas to point to new masterSTOP SLAVE;RESET MASTER;
```

Option 4: Table-level recovery

```
-- If only specific table corruptedREPAIR TABLE table_name;
-- Or drop and recreateDROP TABLE table_name;
RENAME TABLE table_name_backup TO table_name;
```

Post-recovery actions:

1. **Verify data integrity**:

```
CHECK TABLE table_name;
CHECKSUM TABLE table_name;
```

1. **Run application tests** to ensure data consistency

2. **Monitor** for similar issues
3. **Investigate root cause**:
    - Hardware failure?
    - Software bug?
    - Human error?
    - Security breach?
4. **Implement preventive measures**:
    - Regular backups
    - Replica consistency checks
    - Better monitoring
    - Better access controls

Prevention strategy:

```
Regular backups (daily/weekly)
  "
Test restore procedures regularly
  "
Use replication for redundancy
  "
Monitor data integrity
  "
Document recovery procedures
  "
Train team on recovery
```

---

48. How would you implement a zero-downtime deployment?

Answer:

Techniques:

1. Blue-Green Deployment:

1. Deploy new version to green environment
2. Run smoke tests on green
3. Switch load balancer to green
4. Keep blue as fallback
5. Monitor for issues
6. Decommission blue after success

2. Rolling Deployment (Kubernetes):

apiVersion: apps/v1kind: Deploymentmetadata: name: myappspec: replicas: 1 0 strategy: type: RollingUpdate rollingUpdate: maxSurge: 2 # 2 extr a pods during update maxUnavailable: 0 # Never make pods unavailable s elector: matchLabels: app: myapp template: metadata: labels: a pp: myapp spec: containers: - name: myapp image: myapp:v2 t erminationGracePeriodSeconds: 30 readinessProbe: httpGet: pat h: /ready port: 8080 initialDelaySeconds: 10 periodSeconds: 5 livenessProbe: httpGet: path: /health port: 8080 initialDelay Seconds: 15 periodSeconds: 20

3. Canary Deployment:

1. Deploy to 5% of traffic
2. Monitor error rate and latency
3. Gradually increase to 25%, 50%, 100%
4. Auto-rollback if metrics degrade

4. Traffic shifting (Istio VirtualService):

apiVersion: networking.istio.io/v1beta1kind: VirtualServicemetadata: name: m yappspec: hosts: - myapp.example.com http: - match: - uri: prefix: "/" route: - destination: host: myapp-v1 weight: 90 - destination: h ost: myapp-v2 weight: 10

Prerequisites for zero-downtime:

1. **Stateless application**:
    - Don’t store session in application memory
    - Use external session store (Redis)
    - Enable easy horizontal scaling
2. **Health checks**:
    - Readiness probe: Pod ready for traffic?
    - Liveness probe: Pod alive and responsive?
3. **Graceful shutdown**:
    - Listen for SIGTERM
    - Stop accepting new requests
    - Wait for in-flight requests to complete
    - Exit cleanly

```
# Python exampleimport signal
import time
def signal_handler(sig, frame):
    print("Shutdown signal received")
    # Stop accepting new requests    app.stop_accepting_requests()
    # Wait for in-flight requests    time.sleep(30)
    # Exit    exit(0)
signal.signal(signal.SIGTERM, signal_handler)
```

1. **Database migrations**:
    - Always backward compatible
    - Expand schema first, contract later
    - Test migration rollback
2. **Load balancer health checks**:
    - Remove unhealthy instances immediately
    - Don’t send traffic to restarting pods

3. **Dependency management**:
    - Ensure upstream services handle temporary unavailability
    - Implement retry logic with backoff
    - Use circuit breakers

Monitoring during deployment:

```
Real-time metrics:
- Request latency (should not increase)
- Error rate (should not increase)
- Throughput (should remain stable)
- Pod restart count (should not increase)
```

---

49. How would you migrate a monolithic application to microservices?

Answer:

Migration strategy:

Phase 1: Assessment & Planning
1. Identify bounded contexts (Domain-Driven Design)
2. Map service boundaries
3. Identify integration points
4. Plan migration sequence
5. Identify risks

Phase 2: Setup infrastructure
1. Implement Kubernetes cluster
2. Setup CI/CD pipeline
3. Setup monitoring and logging
4. Setup secrets management
5. Setup service mesh (optional but recommended)

Phase 3: Strangler fig pattern (incremental migration)

```
Monolith
 User Service (migrated to microservice)
 Order Service (migrated to microservice)
 Product Service (still in monolith)
 API Gateway (routes to both old and new)
```

Step-by-step:

1. **Create API Gateway** (in front of monolith):

```
Clients ′ API Gateway ′ Monolith
```

1. **Extract first microservice** (usually least dependent):

```
Clients ′ API Gateway ′ User Service (microservice)
                      ′ Monolith (for other services)
```

1. **Extract next service**:

```
Clients ′ API Gateway ′ User Service (microservice)
                      ′ Order Service (microservice)
                      ′ Monolith (remaining)
```

1. **Continue until monolith is empty**

Implementation details:

1. **Service extraction**:

```
# New User Service deploymentapiVersion: apps/v1kind: Deploymentmetadat
a:  name: user-servicespec:  replicas: 3  selector:  matchLabels:  app: user
-service  template:  metadata:  labels:  app: user-service  spec:  co
ntainers:  - name: user-service  image: user-service:v1  ports:  -
containerPort: 8080  env:  - name: DB_HOST  valueFrom:  c
onfigMapKeyRef:  name: app-config  key: db-host
```

1. **API Gateway** (Kong, Ambassador):

```
apiVersion: configuration.konghq.com/v1kind: KongIngressmetadata:  name: api-gatewayspec:  route:  protocols:  - http  - https---apiVersion: networking.k8s.io/v1kind: Ingressmetadata:  name: api-gatewayspec:  rules:  - host: api.example.com  http:  paths:  - path: /users  backend:  service:  name: user-service  port:  number: 8080  - path: /orders  backend:  service:  name: order-service  port:  number: 8080  - path: /  backend:  service:  name: monolith  port:  number: 8080
```

1. **Data synchronization**:
    - Share database initially (dual writes)
    - Gradually migrate data
    - Use change data capture (CDC) for sync
2. **Service-to-service communication**:
    - Synchronous: gRPC, REST APIs
    - Asynchronous: Message queues (RabbitMQ, Kafka)
    - Service mesh for resilience

Challenges & solutions:

Challenge	Solution
Data consistency	Saga pattern, eventual consistency
Network latency	Caching, circuit breakers
Distributed debugging	Distributed tracing (Jaeger)
Service interdependencies	Service mesh, careful planning
Testing complexity	Contract testing, integration tests
Operational overhead	Automation, better monitoring

---

50. How would you design a fault-tolerant CI/CD pipeline?

Answer:

Architecture for resilience:

```
Developer commits
    "
Source Control (Git) with redundancy
    "
CI/CD Controller (Jenkins/GitLab CI) with high availability
    "
Build Agents (multiple, load-balanced)
    "
Test Infrastructure (parallelized, redundant)
    "
Artifact Repository (replicated, backed up)
    "
Deployment Orchestration (with rollback)
    "
Monitoring & Alerts
```

1. Version Control Redundancy:

```
GitHub (primary) + GitHub backup or GitLab mirror
All code in version control
Regular backups of Git repositories
Access control and audit logs
```

2. CI/CD Server High Availability:

```
apiVersion: apps/v1kind: Deploymentmetadata:  name: jenkinsspec:  replicas:
2  # At least 2 for HA  selector:  matchLabels:  app: jenkins  template:  me
tadata:  labels:  app: jenkins  spec:  affinity:  podAntiAffinity:  # S
pread across nodes  requiredDuringSchedulingIgnoredDuringExecution:
- labelSelector:  matchExpressions:  - key: app  operato
r: In  values:  - jenkins  topologyKey: kubernetes.io/host
name  containers:  - name: jenkins  image: jenkins:latest  volumeM
ounts:  - name: jenkins-home  mountPath: /var/jenkins_home  volu
mes:  - name: jenkins-home  persistentVolumeClaim:  claimName: j
```

```
enkins-pvc  # Persistent storage for shared config---apiVersion: v1kind: Persi
stentVolumeClaimmetadata:  name: jenkins-pvcspec:  accessModes:    - Read
WriteMany  # Multiple pods can write  resources:    requests:      storage: 100G
i  storageClassName: nfs  # Shared NFS storage
```

3. Build Agent Resilience:

```
pipeline {  agent {    kubernetes {      label 'build-agent'      yaml '''        apiVersi
on: v1        kind: Pod        spec:          affinity:            podAntiAffinity:              pre
ferredDuringSchedulingIgnoredDuringExecution:                - weight: 100
podAffinityTerm:                    labelSelector:                      matchExpressions:
- key: app                          operator: In                        values:                          - build-ag
ent                    topologyKey: kubernetes.io/hostname          containers:          - n
ame: docker            image: docker:latest            securityContext:              privile
ged: true          - name: kubectl            image: bitnami/kubectl:latest      '''    }  }
stages {    stage('Build') {      steps {        retry(3) {  # Retry failed builds
          sh 'docker build -t myapp:${BUILD_NUMBER} .'        }      }    }    stage('T
est') {      steps {        retry(2) {          sh 'pytest tests/'        }      }    }    stage('De
ploy') {      steps {        script {          try {            sh 'kubectl apply -f k8s/deploy
ment.yaml'            // Wait for rollout            sh 'kubectl rollout status deploymen
t/myapp --timeout=5m'          } catch (Exception e) {            // Rollback on failur
e            sh 'kubectl rollout undo deployment/myapp'            throw e
        }        }      }    }  }  post {    always {      // Always run cleanup      cleanWs
()    }    failure {      // Notify on failure      emailext (        to: 'team@example.co
m',        subject: "Build ${BUILD_NUMBER} failed",        body: "Check logs at
${BUILD_URL}"      )    }  }}
```

4. Artifact Repository Resilience:

```
Primary Repository (Nexus/Artifactory)
    " (replication)
Secondary Repository (backup)

All artifacts tagged with version and build metadata
```

Checksums for integrity verification
Retention policy for cleanup

5. Deployment Safety:

Canary deployment to staging
  ‟
Health checks and smoke tests
  ‟
If healthy: Canary deployment to prod (5%)
  ‟
Monitor metrics for 15 minutes
  ‟
If healthy: Progressive rollout (25%, 50%, 100%)
  ‟
If unhealthy: Auto-rollback
  ‟
Post-deployment validation

6. Monitoring & Alerting:

Monitor pipeline metrics:
- Build success rate
- Build duration
- Test coverage
- Deployment frequency
- Deployment success rate

Alert on:
- High failure rate
- Slow builds
- Deployment failures
- Test coverage drop

7. Disaster Recovery:

Regular backup of:
- Pipeline configurations (Jenkinsfile in Git)
- Build artifacts (separate artifact repo)
- Secrets and credentials (encrypted vault)
- Configuration as Code (Git)

Test recovery procedures:
- Quarterly DR drills
- Document recovery steps
- Practice with team

Fault tolerance checklist:
- âœ… No single point of failure
- âœ… Automatic failure detection and recovery
- âœ… Regular backups and tested recovery
- âœ… Monitoring and alerting
- âœ… Clear runbooks for common failures
- âœ… Team training on incident response

Conclusion

These 50 comprehensive DevOps interview questions cover the breadth and depth of modern DevOps practices. Success in DevOps interviews requires not just memorizing answers but understanding the underlying concepts, trade-offs, and practical applications.

Key Takeaways:

1. **Understand the fundamentals**: DevOps is about collaboration, automation, and continuous improvement
2. **Know your tools**: Docker, Kubernetes, Jenkins, Terraform are essential
3. **Grasp infrastructure concepts**: IaC, HA, DR, monitoring, logging
4. **Think in systems**: Understand how components interact
5. **Be practical**: Know real-world implementations and trade-offs

6. **Keep learning**: DevOps is rapidly evolving; stay updated

Interview Preparation Tips:

- Practice hands-on: Set up a home lab
- Write code: Don’t just know concepts, implement them
- Read documentation: Official docs are your best friend
- Follow industry blogs: Stay current with trends
- Discuss with peers: Talk through problems
- Be honest: Admit what you don’t know; explain how you’d learn it
- Tell stories: Use real examples from your experience

Good luck with your interviews!

Repost and Follow

Nensi Ravaliya

for more content

Want to build your career in cloud?

Subscribe to Yatri Cloud Channel