# Terraform & Kubernetes — 100 Senior DevOps Interview Questions

**Covers both tools end-to-end | 5-Year Engineer Standard | Mental Models Included**

> ***How to use this guide:*** *Each question has three layers — (1) The direct answer, (2) The mental model (ELI5 or real-world analogy), (3) The production-level depth that separates seniors from juniors. Read all three. Skip none.*

# PART 1 — TERRAFORM (Questions 1–50)

## SECTION A: Core Concepts & Architecture

### Q1. What is Terraform and how does it differ from configuration management tools like Ansible or Puppet?

**Answer:**
Terraform is an **Infrastructure-as-Code (IaC) provisioning tool** — it creates, modifies, and destroys cloud resources (VMs, VPCs, DNS records, load balancers). Ansible/Puppet are **configuration management tools** — they configure software *inside* already-running machines.

**Mental Model — The Construction Analogy:**
- Terraform = The **architect + construction crew** that builds the building (provisions the cloud infra)
- Ansible = The **interior designer** that configures what's inside after the building exists

**Key Differences:**

| Dimension | Terraform | Ansible/Puppet |
|---|---|---|
| Primary job | Provision infra | Configure software |
| State model | Stateful (state file) | Stateless (idempotent runs) |
| Language | HCL (declarative) | YAML/DSL (procedural/declarative mix) |
| Approach | Desired state → plan → apply | Push/pull configuration |
| Cloud-native | Yes (providers for AWS, GCP, Azure) | No (uses SSH, WinRM) |

**Production nuance:** In real shops, Terraform provisions EC2 instances, then Ansible configures NGINX inside them. They're complementary, not competitors.

### Q2. Explain Terraform's execution model — the core workflow.

**Answer:**
Terraform follows a strict three-phase workflow:

```
terraform init → terraform plan → terraform apply
```

**Phase-by-phase breakdown:**

1. `terraform init`
   - Downloads provider plugins (AWS, GCP, etc.) into `.terraform/`
   - Initializes the backend (where state is stored)
   - Installs modules referenced in the code
   - **Only needs to run once per workspace, or when providers/modules change**

2. `terraform plan`
   - Reads your `.tf` files (desired state)
   - Reads the current state file (known state)
   - Calls provider APIs to refresh actual infra state
   - Computes a **diff** → what to create (+), update (~), destroy (-)
   - Outputs a plan but changes NOTHING

3. `terraform apply`
   - Executes the plan
   - Makes API calls to the cloud provider
   - Updates the state file after each resource is created/modified

**Mental Model — Git analogy:**
- `plan` = `git diff` (shows what will change)
- `apply` = `git push` (actually makes it happen)
- State file = the remote repository (source of truth)

**Production nuance:** Always save plans in CI: `terraform plan -out=tfplan` → `terraform apply tfplan`. This prevents plan-apply drift in pipelines.

---

### Q3. What is the Terraform state file? Why is it critical and what are the risks of losing it?

**Answer:**
The state file (`terraform.tfstate`) is a **JSON file that maps your HCL resources to real-world cloud resource IDs**. It's Terraform's memory of what it has built.

**What it stores:**
- Resource IDs (e.g., `i-0abc123` for an EC2 instance)
- Resource attributes (IP, ARN, tags)
- Dependency graph metadata
- Provider metadata

**Mental Model — The Inventory Ledger:**
Imagine you run a warehouse. The state file is your **inventory ledger**. Without it, you walk into your

warehouse and have no idea which box belongs to which customer order. You *can* see the boxes (cloud resources exist), but Terraform can't match them to your code.

**What happens if you lose state?**

- Terraform thinks all resources are new → tries to **recreate everything**
- This causes **duplicate resource creation** or **destroy + recreate** cycles
- For databases or stateful services: **catastrophic data loss**
- You can partially recover via `terraform import` (tedious, manual, error-prone)

**Risks of state corruption:**

- Two engineers running `apply` simultaneously → **state file race condition**
- Solution: **Remote state with state locking** (DynamoDB + S3 on AWS)

---

### Q4. What is remote state? How do you configure it on AWS?

**Answer:**

Remote state stores `terraform.tfstate` in a shared, centralized location instead of locally. This enables team collaboration, state locking, and audit trails.

**Why you MUST use remote state in production:**

- Local state = only you can run Terraform
- Remote state = team can collaborate without conflicts
- State locking prevents two `apply` runs from corrupting state simultaneously

**AWS Remote Backend Configuration:**

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

**How the locking mechanism works:**

1. Engineer A runs `terraform apply` → writes a lock record to DynamoDB (`LockID = bucket/key`)
2. Engineer B runs `terraform apply` simultaneously → DynamoDB `ConditionalCheck` fails → gets error "Error acquiring the state lock"
3. Engineer A finishes → lock released from DynamoDB
4. Engineer B can now proceed

**Production tip:** Enable S3 versioning on the state bucket. If state gets corrupted, you can roll back to a previous version.

---

## Q5. What is `terraform import` and when do you use it?

**Answer:**
`terraform import` brings an **existing real-world resource** under Terraform management by writing its state into the state file — without modifying the resource itself.

**When you use it:**
- Resources were created manually (click-ops) and you want to codify them
- State file was lost for specific resources
- Migrating from another IaC tool

**Example:**

```
# Existing EC2 instance with ID i-0abc123def456
terraform import aws_instance.web i-0abc123def456
```

**Critical workflow after import:**
1. Run `terraform import` → state is updated
2. Write the matching HCL code for the resource
3. Run `terraform plan` → verify zero diff (if code matches reality)
4. If plan shows changes, adjust your HCL until plan is clean

**Mental Model:**
`terraform import` is like **scanning an existing item into your inventory system**. The item was already in the warehouse; you're just giving it a barcode so the system knows it exists.

**Limitation:** `terraform import` does NOT generate HCL code for you (as of Terraform <1.5). Terraform 1.5+ introduced `import` blocks that can generate config.

---

## Q6. Explain `count` vs `for_each` — when do you use each?

**Answer:**
Both are meta-arguments for creating multiple instances of a resource, but they behave very differently.

`count` **— index-based:**

```
resource "aws_instance" "web" {
  count         = 3
  ami           = "ami-0abc123"
  instance_type = "t3.micro"
  tags = {
    Name = "web-${count.index}"
  }
}
# Creates: aws_instance.web[0], web[1], web[2]
```

`for_each` **— key-based (map or set):**

```
resource "aws_instance" "web" {
  for_each      = toset(["web-a", "web-b", "web-c"])
  ami           = "ami-0abc123"
  instance_type = "t3.micro"
  tags = {
    Name = each.key
  }
}
# Creates: aws_instance.web["web-a"], web["web-b"], web["web-c"]
```

**The Critical Difference — The Destruction Problem:**

With `count`, if you remove the middle element:

```
Before: ["web-a", "web-b", "web-c"]  → [0], [1], [2]
After remove web-b: ["web-a", "web-c"] → [0], [1]
```

Terraform sees index [2] is gone → **destroys web-c** and recreates it as index [1]. **This is dangerous for databases/stateful resources.**

With `for_each`, keys are stable. Removing "web-b" only affects "web-b". "web-a" and "web-c" are untouched.

**Rule of thumb:**
- `count` → simple numeric repetition where order doesn't matter (e.g., fixed N identical servers)
- `for_each` → named, distinct resources (different environments, different subnets) — **always prefer this in production**

---

## Q7. What is the `lifecycle` meta-argument and what are its key options?

**Answer:**
`lifecycle` controls how Terraform manages resource create/destroy/update behavior.

**Key options:**

```
resource "aws_instance" "web" {
  # ...
  lifecycle {
    create_before_destroy = true   # Create new before destroying old (zero-downtime)
    prevent_destroy       = true   # Block terraform destroy (protect prod DBs)
    ignore_changes        = [tags] # Don't drift-detect these attributes
    replace_triggered_by  = [aws_security_group.web] # Replace if SG changes
  }
}
```

`create_before_destroy`**:**
Default Terraform behavior: destroy old → create new. This causes downtime.

With `create_before_destroy = true`: create new → update references → destroy old. **Zero-downtime replacement.**

Use case: **EC2 instances behind ALB**, **Auto Scaling Groups**

**`prevent_destroy`:**
Acts as a safety net. Running `terraform destroy` or any plan that would destroy this resource throws an error. **Must be used on production databases (RDS, DynamoDB).**

**`ignore_changes`:**
If another system (auto-scaling, external automation) modifies a resource attribute, Terraform won't revert it on next apply.

```
ignore_changes = [desired_capacity]  # ASG capacity managed by autoscaler
```

---

### Q8. What is `depends_on` and when should you explicitly use it?

**Answer:**
Terraform automatically figures out resource dependencies by analyzing references. `depends_on` is for **implicit dependencies that Terraform can't detect through reference analysis**.

**Automatic dependency (Terraform detects this):**

```
resource "aws_instance" "web" {
  subnet_id = aws_subnet.main.id  # Terraform sees the reference → knows subnet must exist
first
}
```

**Explicit `depends_on` needed:**

```
resource "aws_instance" "app" {
  # ...
  depends_on = [aws_iam_role_policy_attachment.app_policy]
  # The EC2 instance doesn't reference the IAM policy directly,
  # but the instance's startup script USES IAM permissions.
  # If we create the instance before the policy is attached, startup fails.
}
```

**When to use `depends_on`:**
- Resource relies on side effects of another resource (not direct attribute reference)
- Module-level dependencies: `depends_on = [module.networking]`
- Ordering bootstrap scripts that rely on IAM, SSM Parameter Store values

**Warning:** Overusing `depends_on` forces serial execution and kills Terraform's parallel provisioning speed. Use it only when necessary.

---

## Q9. Explain Terraform modules — structure, usage, and best practices.

**Answer:**

A module is a **reusable, encapsulated package of Terraform configuration** — a collection of `.tf` files in a directory treated as a unit.

**Mental Model — Functions in Programming:**

A module is like a function. You pass inputs (variables), it does work, returns outputs. You can call the same "function" multiple times with different inputs.

**Module Structure:**

```
modules/
└── ec2_web/
    ├── main.tf         # Resources
    ├── variables.tf    # Input variables (function parameters)
    ├── outputs.tf      # Output values (return values)
    └── versions.tf     # Provider/Terraform version constraints
```

**Calling a module:**

```
module "web_server" {
  source        = "./modules/ec2_web"
  instance_type = "t3.medium"
  environment   = "production"
  vpc_id        = aws_vpc.main.id
}

# Access module outputs
output "web_server_ip" {
  value = module.web_server.public_ip
}
```

**Module sources:**

```
source = "./local/path"                    # Local
source = "github.com/org/repo//modules/vpc" # GitHub
source = "registry.terraform.io/hashicorp/vpc/aws"  # Terraform Registry
source = "git::https://example.com/tf-modules.git?ref=v1.2.0"  # Git with tag
```

**Best practices:**

- Version-pin external modules: `version = "~> 3.0"`
- Keep modules focused (single responsibility)
- Always expose necessary outputs — don't make callers dig into module internals
- Use `terraform-docs` to auto-generate module documentation

## Q10. What are Terraform workspaces and what are their limitations?

**Answer:**
Workspaces allow **multiple state files from the same configuration** — used to manage multiple environments (dev/staging/prod) from a single codebase.

```
terraform workspace new staging
terraform workspace select production
terraform workspace list
terraform workspace show  # current workspace name
```

**Using workspace in code:**

```
locals {
  instance_type = {
    default    = "t3.micro"
    staging    = "t3.small"
    production = "t3.large"
  }
}

resource "aws_instance" "web" {
  instance_type = local.instance_type[terraform.workspace]
}
```

**Mental Model — Git Branches:**
Workspaces are like git branches for your state. Same code, different states. Each workspace gets its own `terraform.tfstate` file in the backend.

**Limitations (this is where seniors differ from juniors):**
1. **Same backend, same codebase** — no isolation of provider credentials or code versions between envs
2. **Easy to apply in wrong workspace** — human error risk
3. **Not suitable for environments with significantly different configs** — if prod needs 10 resources that dev doesn't have, workspace isn't the right tool
4. **No RBAC at workspace level** — anyone with backend access can switch workspaces

**Production reality:** Most mature teams use **separate directories + separate state files + separate AWS accounts** per environment rather than workspaces. Workspaces work well for short-lived feature environments spun off from a base config.

# SECTION B: Providers, Variables & Expressions

## Q11. What is a Terraform provider and how does provider versioning work?

**Answer:**
A provider is a **plugin that lets Terraform talk to an external API** (AWS, GCP, Azure, Kubernetes, GitHub, Datadog, etc.). Each provider translates your HCL resource declarations into API calls.

**Provider declaration:**

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"   # >= 5.0.0 and < 6.0.0
    }
  }
}

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

**Version constraint operators:**
- `= 5.0.0` — exact version
- `~> 5.0` — pessimistic constraint: `>= 5.0, < 6.0`
- `~> 5.0.1` — `>= 5.0.1, < 5.1.0` (patch only)
- `>= 4.0, < 6.0` — range

**The `.terraform.lock.hcl` file:**
When you run `terraform init`, Terraform creates a lock file pinning exact provider versions. **Commit this to git.** It ensures every team member and CI pipeline uses the same provider version — like `package-lock.json` in Node.js.

**Multiple provider instances (aliases):**

```
provider "aws" {
  alias  = "us_east"
  region = "us-east-1"
}

provider "aws" {
  alias  = "eu_west"
  region = "eu-west-1"
}

resource "aws_instance" "eu_server" {
  provider = aws.eu_west
  # ...
}
```

## Q12. Explain local values, input variables, and output values — when to use each.

**Answer:**

**Input Variables ( `variable` ) — External inputs:**

```
variable "instance_type" {
  description = "EC2 instance type"
  type        = string
  default     = "t3.micro"
  validation {
    condition     = contains(["t3.micro", "t3.small", "t3.medium"], var.instance_type)
    error_message = "Must be a valid instance type."
  }
}
```

Use when: Values change between environments, or callers need to customize behavior. Passed via `-var`, `terraform.tfvars`, or environment variables ( `TF_VAR_instance_type` ).

**Local Values ( `local` ) — Internal computed values:**

```
locals {
  common_tags = {
    Environment = var.environment
    ManagedBy   = "Terraform"
    Owner       = "platform-team"
  }
  full_name = "${var.project}-${var.environment}"
}
```

Use when: You compute a value used multiple times (DRY principle). Not exposed outside the module.

**Output Values ( `output` ) — Return values:**

```
output "alb_dns_name" {
  description = "DNS name of the Application Load Balancer"
  value       = aws_lb.main.dns_name
  sensitive   = false
}
```

Use when: Other modules or root config need to read values from this module.

**Mental Model:**

- `variable` = function parameter (external input)
- `local` = local variable inside a function (internal computation)
- `output` = function return value (exported result)

## Q13. What are dynamic blocks in Terraform and when do you need them?

**Answer:**

Dynamic blocks let you generate **repeated nested blocks** within a resource based on a collection — because you can't use `for_each` on nested blocks, only on top-level resources.

**Problem without dynamic blocks:**

```
# Hard-coded — inflexible
resource "aws_security_group" "web" {
  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  ingress {
    from_port   = 443
    to_port     = 443
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

**With dynamic blocks:**

```
variable "ingress_rules" {
  default = [
    { port = 80,   protocol = "tcp", cidr = "0.0.0.0/0" },
    { port = 443,  protocol = "tcp", cidr = "0.0.0.0/0" },
    { port = 8080, protocol = "tcp", cidr = "10.0.0.0/8" },
  ]
}

resource "aws_security_group" "web" {
  dynamic "ingress" {
    for_each = var.ingress_rules
    content {
      from_port   = ingress.value.port
      to_port     = ingress.value.port
      protocol    = ingress.value.protocol
      cidr_blocks = [ingress.value.cidr]
    }
  }
}
```

**Production use cases:** Security group rules, ECS task definition container definitions, ALB listener rules, IAM policy statements.

**Caution:** Nested dynamic blocks are valid but become hard to read. If you're nesting 3 levels deep, consider restructuring your data model.

## Q14. What are data sources in Terraform?

**Answer:**
Data sources let Terraform **read information from external sources** (existing cloud resources, APIs, files) **without managing them**. They're read-only references.

```
# Read an existing VPC (not managed by this Terraform config)
data "aws_vpc" "existing" {
  filter {
    name   = "tag:Name"
    values = ["production-vpc"]
  }
}

# Use it in a resource
resource "aws_subnet" "new" {
  vpc_id     = data.aws_vpc.existing.id
  cidr_block = "10.0.10.0/24"
}
```

**Common data sources:**

```
data "aws_ami" "amazon_linux" {          # Latest AMI
  most_recent = true
  owners      = ["amazon"]
  filter { name = "name"; values = ["amzn2-ami-hvm-*-x86_64-gp2"] }
}

data "aws_caller_identity" "current" {}  # Current AWS account ID
data "aws_region" "current" {}           # Current region
data "aws_availability_zones" "available" { state = "available" }
data "terraform_remote_state" "network" { # Read another stack's outputs
  backend = "s3"
  config = { bucket = "tfstate", key = "network/terraform.tfstate", region = "us-east-1" }
}
```

**Mental Model:**
Data sources are like SQL `SELECT` queries — you're reading existing data, not inserting or modifying it.

---

## Q15. How does Terraform handle sensitive values?

**Answer:**
Sensitive values are masked in plan/apply output but are **still stored in plain text in the state file** — this is the critical nuance most candidates miss.

**Marking as sensitive:**

```
variable "db_password" {
  type      = string
  sensitive = true
}

output "connection_string" {
  value     = "postgres://user:${var.db_password}@${aws_db_instance.main.endpoint}"
  sensitive = true
}
```

**The state file problem:**

```
# State file stores everything in plain text!
cat terraform.tfstate | jq '.resources[].instances[].attributes.password'
# → "my-super-secret-password"
```

**Production solution — Never put secrets in variables:**

```
# Fetch from AWS Secrets Manager at apply time
data "aws_secretsmanager_secret_version" "db_password" {
  secret_id = "prod/rds/master-password"
}

resource "aws_db_instance" "main" {
  password = data.aws_secretsmanager_secret_version.db_password.secret_string
}
```

**Best practices:**

- Use `sensitive = true` to prevent console/log exposure
- Store secrets in AWS Secrets Manager, HashiCorp Vault, or SSM Parameter Store (SecureString)
- Restrict access to the S3 state bucket (IAM policies + encryption)
- Enable S3 server-side encryption on the state bucket

---

## SECTION C: State Management & Troubleshooting

---

### Q16. What is `terraform taint` / `terraform apply -replace` and when do you use it?

**Answer:**

These commands **force recreation of a specific resource** on the next apply, even if Terraform wouldn't otherwise replace it.

**Old way (deprecated in TF 1.0):**

```
terraform taint aws_instance.web
terraform apply
```

**New way:**

```
terraform apply -replace="aws_instance.web"
```

**When to use:**
- EC2 instance has a corrupted OS / failed config management run
- RDS parameter group changes require replace (can't update in-place)
- SSL certificate renewed and you need to force re-attachment
- Kubernetes node is unhealthy and needs reprovisioning

**What happens:**
1. Resource is marked for destruction
2. On next apply: old resource destroyed, new one created
3. `create_before_destroy` lifecycle rule is respected if set

---

## Q17. Explain `terraform state` subcommands and their use cases.

**Answer:**
The `terraform state` family of commands lets you **directly manipulate the state file** — a power tool that should be used carefully.

```
terraform state list                          # List all resources in state
terraform state show aws_instance.web         # Show all attributes of a resource
terraform state mv aws_instance.old aws_instance.new  # Rename/move resource in state
terraform state rm aws_instance.web           # Remove resource from state (stop managing it)
terraform state pull                          # Download remote state locally
terraform state push                          # Upload local state to remote
```

**`state mv` use cases:**
- Renamed a resource in HCL without wanting to destroy + recreate it
- Moving a resource into or out of a module
- Restructuring configuration without infrastructure changes

**`state rm` use cases:**
- Resource was manually deleted from cloud; remove it from state to stop Terraform from trying to manage it
- You want to "abandon" a resource (let it exist without Terraform tracking it)

**Production war story scenario:**
"We renamed our module from `module.app` to `module.application`. Without `state mv`, Terraform would destroy all resources in `module.app` and create new ones in `module.application`. We used `terraform state mv` to surgically rename each resource reference."

---

## Q18. What is `terraform refresh` and when is it dangerous?

**Answer:**

`terraform refresh` **syncs the state file with real-world cloud resources** by querying provider APIs and updating state — without making any changes to actual infrastructure.

**Modern approach:** As of Terraform 0.15.4+, `terraform plan` and `terraform apply` automatically include a refresh step. Explicit `terraform refresh` is rarely needed.

**When it was used:**
- After someone manually modified a resource outside Terraform
- To detect drift between state and reality

**Why it's dangerous:**

```
Scenario: Engineer manually changes an EC2 security group.
terraform refresh → state now reflects the manual change.
Next terraform apply → Terraform thinks manual change is "desired state"
→ May or may not revert it depending on what's in HCL.
```

The real danger: refresh can update state in ways that make drift invisible. Better approach is `terraform plan -refresh-only` which shows you what drift exists without applying changes.

---

## Q19. How do you handle Terraform configuration drift?

**Answer:**

Drift = real infrastructure state differs from Terraform state or HCL code.

**Detection:**

```
terraform plan -refresh-only    # Show what changed in real infra vs state
terraform plan                  # Show what will change to match HCL
```

**Drift scenarios and responses:**

| Scenario | Action |
|---|---|
| Someone added a tag manually | `ignore_changes = [tags]` if intentional, or `apply` to revert |
| Instance type was manually upgraded | Update HCL to match, then run `apply` to confirm no diff |
| Resource was manually deleted | Terraform will show "must create" on next plan — apply to recreate |
| New resource added outside TF | Use `terraform import` to bring it under management |

**Production setup — automated drift detection:**

```
# In CI/CD (runs nightly)
terraform plan -detailed-exitcode
# Exit code 0 = no changes
# Exit code 1 = error
# Exit code 2 = changes detected → alert on-call team
```

### Q20. What is Terraform Graph and how do you use it?

**Answer:**
Terraform builds a **Directed Acyclic Graph (DAG)** of all resources and their dependencies. This graph determines the order of creation and enables parallel provisioning of independent resources.

```
terraform graph | dot -Tsvg > graph.svg   # Visualize with Graphviz
```

**How parallel provisioning works:**

```
VPC ──→ Subnet A ──→ EC2-1 ─┐
     └→ Subnet B ──→ EC2-2 ─┴──→ ALB
```

Terraform creates VPC first, then Subnet A and B **in parallel** (no dependency between them), then EC2-1 and EC2-2 **in parallel**, then ALB.

**Mental Model:**
Terraform is like a kitchen with multiple chefs. Independent tasks (chopping onions, boiling water) happen simultaneously. Dependent tasks (sauce goes on after pasta is cooked) wait for prerequisites.

**Controlling parallelism:**

```
terraform apply -parallelism=20   # Default is 10; increase for large infra
```

## SECTION D: CI/CD, Security & Advanced Patterns

### Q21. How do you structure Terraform for a multi-environment, multi-account AWS setup?

**Answer:**
This is a common senior-level design question. There are two main patterns:

**Pattern 1 — Directory-based (most common in production):**

```
infra/
├── modules/
│   ├── vpc/
│   ├── ec2/
│   └── rds/
├── environments/
│   ├── dev/
│   │   ├── main.tf        # Calls modules
│   │   ├── variables.tf
│   │   └── backend.tf     # dev state bucket
│   ├── staging/
│   │   └── ...            # staging state bucket
│   └── prod/
│       └── ...            # prod state bucket
```

Each environment is a **separate Terraform root module** with its own state. To apply:

```
cd environments/prod && terraform apply
```

**Pattern 2 — Terragrunt (wrapper tool):**
Terragrunt eliminates DRY violation in backend configs. Define backend once, inherit in all envs.

**Multi-account isolation:**

```
provider "aws" {
  assume_role {
    role_arn = "arn:aws:iam::${var.account_id}:role/TerraformExecutionRole"
  }
}
```

Each environment assumes a role in its respective AWS account. **This is the gold standard for security isolation.**

---

## Q22. How do you handle Terraform in CI/CD pipelines?

**Answer:**

**Standard pipeline:**

```
# GitHub Actions example
stages:
  - validate    # terraform validate + fmt check
  - plan        # terraform plan -out=tfplan
  - apply       # terraform apply tfplan (manual approval gate for prod)
```

**Best practices:**

1. **Never store credentials in pipeline** — use OIDC (GitHub Actions → AWS)

`yaml - uses: aws-actions/configure-aws-credentials@v4 with: role-to-assume: arn:aws:iam::123456789:role/GitHubActionsRole aws-region: us-east-1`

2. **Plan in PR, apply on merge:**
   - PR opened → run `plan`, post output as PR comment
   - PR merged to main → run `apply`
3. **Lock the state before plan:**
   `bash terraform plan -lock=true -out=tfplan`
4. **Use plan files:** Never run `apply` without `-out=tfplan` in CI — prevents plan-apply drift
5. **Atlantis** — open-source tool for GitOps-based Terraform. `terraform plan` runs on PR, `terraform apply` runs via PR comment.

---

## Q23. What is the Terraform `moved` block?

**Answer:**

Introduced in Terraform 1.1, the `moved` block lets you **rename or relocate resources in HCL without destroying and recreating them** — like `terraform state mv` but as code (reviewable, repeatable).

```
# You renamed aws_instance.old to aws_instance.web in your code
moved {
  from = aws_instance.old
  to   = aws_instance.web
}

# Moving a resource into a module
moved {
  from = aws_security_group.allow_http
  to   = module.security.aws_security_group.allow_http
}
```

**Why it's better than `terraform state mv`:**
- The moved block is **version-controlled** in HCL
- Works in team environments without each member running manual CLI commands
- Shows up in `terraform plan` output so the team can review the rename

**After the rename is stable:** Remove the `moved` block in a subsequent PR (once everyone has applied).

---

## Q24. Explain Terraform backend types and when to choose each.

**Answer:**

| Backend | When to Use |
| --- | --- |
| `local` | Development/learning only. Never in teams. |
| `s3` (+ DynamoDB) | AWS shops. Most common production choice. |
| `gcs` | GCP shops. |
| `azurerm` | Azure shops. |
| `terraform cloud` | HashiCorp's managed service. Includes locking, remote execution, RBAC. |
| `consul` | HashiCorp stack shops (rare in pure cloud environments). |
| `http` | Custom backend (GitLab-managed state uses this). |

**S3 Backend production checklist:**

```
backend "s3" {
  bucket         = "company-tfstate-prod"
  key            = "platform/api/terraform.tfstate"
  region         = "us-east-1"
  encrypt        = true                 # Server-side encryption
  dynamodb_table = "terraform-locks"    # State locking
  # Optional but recommended:
  kms_key_id     = "arn:aws:kms:..."    # CMK encryption
}
```

**S3 bucket requirements:**
- Versioning enabled (rollback)
- Block public access (critical)
- Server-side encryption (AES-256 or KMS)
- Access logging enabled
- MFA delete enabled (prevents accidental deletion)

---

## Q25. What is the difference between `terraform destroy` and removing a resource from HCL?

**Answer:**
Both result in infrastructure deletion, but they work differently:

`terraform destroy`**:**
- Destroys **all resources** managed in the current state
- A nuclear option — destroys the entire environment
- Rarely used in production (except for ephemeral environments)

**Removing a resource block from HCL:**

- Only the removed resource is destroyed on next `apply`
- Targeted and controlled
- The correct way to decommission specific resources

**To remove a resource without destroying it:**

```
terraform state rm aws_instance.web
# Then delete the HCL block
# Result: resource still exists in AWS, Terraform no longer tracks it
```

**Production pattern for decommissioning:**

1. Set `prevent_destroy = true` → remove it
2. Remove from HCL in a PR
3. Run `terraform plan` → review what will be destroyed
4. Get peer review + approval
5. `terraform apply`

---

## Q26–Q30: Advanced Terraform Topics

### Q26. What is `templatefile()` and how does it differ from `file()`?

`file("path")` reads a file as a raw string. `templatefile("path", vars)` reads a file and **substitutes variables** using HCL template syntax `${var}`.

```
resource "aws_launch_template" "web" {
  user_data = base64encode(templatefile("${path.module}/userdata.sh.tpl", {
    db_endpoint = aws_db_instance.main.endpoint
    app_version = var.app_version
  }))
}
```

`userdata.sh.tpl`:

```
#!/bin/bash
echo "DB_HOST=${db_endpoint}" >> /etc/app.env
echo "VERSION=${app_version}" >> /etc/app.env
```

---

### Q27. What is `terraform validate` vs `terraform fmt`?

- `terraform fmt` — **formats** HCL code to canonical style (indentation, alignment). Like `prettier` for JS.
- `terraform validate` — **validates syntax and internal consistency** of configuration. Checks that variable references exist, resource types are valid. Does NOT check cloud API (no credentials needed).

In CI: run `fmt -check` (fails if code isn't formatted) + `validate` before `plan`.

### Q28. How do you test Terraform code?

- `terraform validate` — syntax checking
- `terraform plan` — integration test against real cloud
- **Terratest (Go)** — spin up real infra, run assertions, tear down
- **tflint** — linting for provider-specific rules (e.g., invalid instance types)
- **Checkov / tfsec** — security scanning of Terraform code (finds open security groups, unencrypted buckets)
- **OPA (Open Policy Agent)** — policy-as-code for compliance (enforce tagging, region restrictions)

### Q29. What is a Terraform provider's `alias` and when do you need it?

Provider aliases allow **multiple configurations of the same provider** in one codebase. Common for:
- Multi-region deployments
- Cross-account resource creation
- Different credentials for different resources

```
provider "aws" { alias = "primary"; region = "us-east-1" }
provider "aws" { alias = "replica"; region = "eu-west-1" }

resource "aws_s3_bucket" "backup" {
  provider = aws.replica
  bucket   = "my-backup-eu"
}
```

### Q30. What are Terraform provisioners and why should you avoid them?

Provisioners (remote-exec, local-exec, file) execute scripts on resources after creation. They exist as a **last resort** for things no provider supports.

**Why avoid them:**
- Not idempotent — if provisioner fails mid-run, resource is in unknown state
- Terraform marks resource as tainted on provisioner failure
- Better alternatives: EC2 User Data, AWS SSM, cloud-init, Ansible post-provisioning

```
# Acceptable use of local-exec (run locally after resource creates)
resource "aws_instance" "web" {
  # ...
  provisioner "local-exec" {
    command = "echo ${self.public_ip} >> inventory.txt"
  }
}
```

# PART 2 — KUBERNETES (Questions 51–100)

## SECTION A: Architecture & Control Plane

### Q51. Explain the Kubernetes architecture — control plane vs data plane.

**Answer:**

**Mental Model — An Airport:**
- **Control Plane** = Airport Control Tower (ATC). It doesn't fly planes; it orchestrates, schedules, and tracks everything.
- **Data Plane (Worker Nodes)** = The planes and runways. Actual work happens here.

**Control Plane Components:**

| Component | Role |
|---|---|
| **API Server** | Front door to K8s. All kubectl commands, internal communication go through it. RESTful

HTTP API. |
| **etcd** | Distributed key-value store. The ONLY persistent storage in the cluster. All cluster state lives here. |
| **Scheduler** | Assigns pods to nodes based on resource requirements, affinity rules, taints/tolerations. |
| **Controller Manager** | Runs control loops (reconciliation loops) — ensures desired state matches actual state. |
| **Cloud Controller Manager** | Interfaces with cloud APIs (AWS for ELBs, EBS volumes, node registration). |

**Data Plane Components:**
| Component | Role |
|---|---|
| **kubelet** | Agent on each node. Communicates with API server. Ensures pods are running as specified. |
| **kube-proxy** | Manages iptables/IPVS rules for Service networking on each node. |
| **Container Runtime** | Runs containers (containerd, CRI-O). Docker is no longer supported directly in K8s 1.24+. |

---

## Q52. What is etcd and what happens if it goes down?

**Answer:**
etcd is a **distributed, consistent key-value store** that stores ALL Kubernetes cluster state — pod specs, node info, secrets, configmaps, RBAC policies, service definitions — everything.

**Mental Model:**
etcd is the **brain's long-term memory** of the cluster. The API server is the brain's working memory. If you erase long-term memory (etcd), the brain (API server) can no longer remember what's supposed to exist.

**What happens when etcd goes down:**
1. **Existing workloads keep running** (pods already scheduled on nodes continue running — kubelet doesn't need etcd)
2. **No new pods can be scheduled** (scheduler needs to read/write state)
3. **kubectl commands fail** (API server can't read/write state)
4. **Autoscaling stops** (controllers can't reconcile state)
5. **Services keep working** (kube-proxy rules already in iptables — existing connections work)

**etcd HA setup:**
etcd uses the Raft consensus protocol. You need **odd numbers: 3 or 5 etcd members** (quorum = `(n/2)+1`).
- 3 members → tolerate 1 failure
- 5 members → tolerate 2 failures

**Production rule:** Back up etcd regularly:

```
ETCDCTL_API=3 etcdctl snapshot save backup.db \
  --endpoints=https://127.0.0.1:2379 \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key
```

### Q53. What is the Kubernetes API server and how does a `kubectl apply` actually work?

**Answer:**

The API server is the **single entry point for all cluster operations**. It validates, authenticates, authorizes, and persists every state change.

**What happens when you run `kubectl apply -f deployment.yaml`:**

```
kubectl apply -f deployment.yaml
    ↓
1. kubectl reads your YAML, serializes to JSON
2. kubectl sends HTTP PATCH/POST to API server
   POST /apis/apps/v1/namespaces/default/deployments

3. API Server: Authentication
   → Validates client cert / bearer token / OIDC token

4. API Server: Authorization (RBAC)
   → Can this user CREATE/UPDATE deployments in this namespace?

5. API Server: Admission Controllers
   → MutatingAdmissionWebhook (can modify the object — add labels, inject sidecars)
   → ValidatingAdmissionWebhook (can reject the object — policy enforcement)

6. API Server: Schema validation
   → Is this a valid Deployment spec?

7. API Server: Persist to etcd
   → Writes the Deployment object to etcd

8. Deployment Controller (in Controller Manager) sees the change
   → Computes desired pods vs actual pods
   → Creates ReplicaSet

9. ReplicaSet Controller sees new ReplicaSet
   → Creates Pod objects (just API objects, not running yet)

10. Scheduler sees unscheduled pods
    → Selects best node based on resources, affinity
    → Writes "node binding" to pod spec in etcd

11. kubelet on selected node sees pod assigned to it
    → Pulls image (via container runtime)
    → Creates container
    → Reports status back to API server
```

## Q54. What is the role of the Scheduler and how does it choose a node?

**Answer:**
The scheduler assigns **pending pods to nodes** using a two-phase process: **Filtering** → **Scoring**.

**Phase 1 — Filtering (eliminates ineligible nodes):**
- Does the node have enough CPU/Memory? (`requests`)
- Does the node match `nodeSelector`?
- Does the pod tolerate the node's taints?
- Are there `affinity/anti-affinity` hard rules?
- Is the node in the correct zone?
- Does the node have the required GPU?

**Phase 2 — Scoring (ranks eligible nodes):**
- Least requested resources (spread load)
- Image already pulled on node (faster startup)
- Node affinity soft rules (`preferredDuringScheduling`)
- Pod anti-affinity preference

**The node with the highest score wins.**

**Mental Model:**
Scheduler is a **hotel booking system**. Filtering removes hotels that don't have available rooms (no capacity). Scoring ranks remaining hotels by star rating, price, proximity (resource spread, affinity). The top-ranked hotel gets the booking.

# SECTION B: Pods, Deployments & Controllers

## Q55. What is a Pod? Why is it the smallest deployable unit (not a container)?

**Answer:**
A Pod is a **wrapper around one or more containers** that share:
- The same **network namespace** (same IP address, same localhost)
- The same **PID namespace** (can see each other's processes)
- The same **IPC namespace** (can use shared memory)
- The same **volumes** (shared storage)

**Why not just use containers directly?**
A Pod models the concept of a "logical host" — just as you might run multiple processes on one VM, a pod groups tightly-coupled processes that must communicate over localhost.

**The Sidecar Pattern — Why multiple containers per pod matters:**

```
Pod: my-web-app
├── Container 1: nginx (serves traffic)
├── Container 2: log-shipper (reads nginx logs, ships to Elasticsearch)
└── Container 3: envoy (service mesh proxy, intercepts all traffic)
```

All containers share the pod's network. The log-shipper reads logs from a shared volume. Envoy intercepts traffic on `localhost` because they share the network namespace.

**What a Pod is NOT:**

- Not highly available on its own — if a pod dies, it stays dead (a controller like Deployment recreates it)
- Not self-healing — controllers provide self-healing

---

## Q56. Explain the Deployment → ReplicaSet → Pod hierarchy.

**Answer:**

```
Deployment
└── ReplicaSet (v1) — old version, being scaled down
└── ReplicaSet (v2) — new version, being scaled up
    ├── Pod-1 (running v2 app)
    ├── Pod-2 (running v2 app)
    └── Pod-3 (running v2 app)
```

**Deployment:** Manages the rolling update strategy. Knows what version of the app to run, how many replicas, update strategy.

**ReplicaSet:** Ensures a specified number of identical pod replicas are running at all times. Automatically replaces crashed pods.

**Pod:** Runs the actual container.

**Why this three-layer design?**

During a rolling update, the Deployment creates a new ReplicaSet for the new version and gradually scales it up while scaling down the old ReplicaSet. If you rolled back, you'd simply scale the old ReplicaSet back up (it still exists). **Kubernetes never deletes old ReplicaSets immediately** — `revisionHistoryLimit` (default: 10) controls how many old ReplicaSets to keep for rollback.

**Rollback:**

```
kubectl rollout undo deployment/my-app
kubectl rollout undo deployment/my-app --to-revision=3
kubectl rollout history deployment/my-app
```

---

## Q57. What are the rolling update strategies in Kubernetes?

**Answer:**

**Strategy 1 — RollingUpdate (default):**

```
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 1          # Max pods ABOVE desired count during update
    maxUnavailable: 0  # Max pods BELOW desired count during update
```

With `replicas: 4, maxSurge: 1, maxUnavailable: 0`:

```
Start:  [v1][v1][v1][v1]
Step 1: [v1][v1][v1][v1][v2]  (surge +1)
Step 2: [v1][v1][v1][v2][v2]  (old removed, new added)
...
End:    [v2][v2][v2][v2]
```

Always 4+ pods serving traffic. Zero downtime.

**Strategy 2 — Recreate:**

```
strategy:
  type: Recreate
```

Kills ALL old pods, then creates all new pods. Causes downtime. Use for databases or apps that can't run two versions simultaneously.

**Blue-Green (not native, achieved with Services):**
- Blue = current production (v1 pods)
- Green = new version (v2 pods), running but not receiving production traffic
- Switch: change Service selector from `version: blue` to `version: green`
- Instant cutover, instant rollback capability

**Canary (native with progressive delivery tools):**
- Run 1 new pod alongside 9 old pods → 10% traffic to new version
- Monitor errors → if OK, scale up new version gradually
- Achieved natively via two Deployments with the same Service label, or via Argo Rollouts/Flagger.

---

## Q58. Explain StatefulSets vs Deployments — when do you use each?

**Answer:**

**Deployments** — for **stateless** applications:
- Pods are interchangeable (any pod can serve any request)
- Pods get random names: `nginx-7d8f9c-xkq2p`
- PersistentVolumes are NOT tied to specific pods
- Can be scaled up/down in any order

**StatefulSets** — for **stateful** applications (databases, distributed systems):
- Pods have **stable, ordered names**: `postgres-0`, `postgres-1`, `postgres-2`
- Pods have **stable network identity**: DNS = `postgres-0.postgres-svc.default.svc.cluster.local`
- Each pod gets its **own PersistentVolumeClaim** (data persists across pod restarts)
- Pods start in order (0 → 1 → 2) and terminate in reverse order (2 → 1 → 0)

**Mental Model:**
- Deployment pods = **anonymous workers** in a factory. Any worker can do any job. They're identical and interchangeable.
- StatefulSet pods = **named employees** with assigned desks. Bob always sits at desk-0. Bob's files stay at desk-0 even when Bob takes a day off.

**Use StatefulSets for:**
- Databases (PostgreSQL, MySQL, Cassandra)
- Message queues (Kafka, RabbitMQ)
- Distributed consensus systems (ZooKeeper, etcd itself)
- Any workload where pod identity and persistent storage matter

---

## Q59. What is a DaemonSet and what are its common use cases?

**Answer:**
A DaemonSet ensures that **exactly one pod runs on every node** (or a subset of nodes matching a selector). When a new node joins the cluster, a DaemonSet pod is automatically scheduled on it.

**Common use cases:**
- **Log collection:** Fluentd/Filebeat collecting logs from every node's `/var/log`
- **Metrics:** Node Exporter (Prometheus) collecting node-level metrics from every node
- **Security:** Falco runtime security scanner on every node
- **Networking/CNI plugins:** Calico, Flannel — must run on every node
- **Service mesh proxy:** Envoy/Linkerd2 (in non-sidecar injection mode)
- **Storage:** Ceph CSI driver on every node

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-exporter
spec:
  selector:
    matchLabels:
      app: node-exporter
  template:
    spec:
      hostPID: true          # Access host PID namespace
      hostNetwork: true      # Use host network (for node metrics)
      containers:
      - name: node-exporter
        image: prom/node-exporter
        ports:
        - containerPort: 9100
          hostPort: 9100
```

## Q60. What are Jobs and CronJobs?

**Answer:**

**Job** — runs a task to completion (not continuously):

```yaml
apiVersion: batch/v1
kind: Job
spec:
  completions: 3          # Run 3 successful completions
  parallelism: 2          # Run 2 pods simultaneously
  backoffLimit: 4         # Retry up to 4 times on failure
  template:
    spec:
      restartPolicy: Never  # Must be Never or OnFailure for Jobs
      containers:
      - name: db-migrator
        image: myapp:migrate
        command: ["python", "migrate.py"]
```

**CronJob** — scheduled Jobs:

```yaml
apiVersion: batch/v1
kind: CronJob
spec:
  schedule: "0 2 * * *"     # Every day at 2 AM (standard cron syntax)
  concurrencyPolicy: Forbid # Don't start new job if previous still running
  successfulJobsHistoryLimit: 3
  failedJobsHistoryLimit: 1
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: backup
            image: backup-tool:latest
```

`concurrencyPolicy` **options:**
- `Allow` (default) — run concurrently
- `Forbid` — skip if previous is running
- `Replace` — cancel previous, start new

---

# SECTION C: Kubernetes Networking (The Most Critical Section)

---

## Q61. Explain the Kubernetes networking model — the fundamental rules.

**Answer:**
Kubernetes mandates a **flat network model** with three key rules:

1. **Every pod gets a unique cluster-wide IP** (no port mapping, no NAT between pods)
2. **All pods can communicate with all other pods without NAT** (regardless of which node they're on)
3. **Nodes can communicate with all pods** (and vice versa)

**Mental Model — An office floor:**
Every employee (pod) has their own desk phone with a unique extension (pod IP). Anyone can call anyone directly by extension — no switchboard needed, no area codes. It's a flat network.

**How cross-node pod communication actually works (without CNI):**

```
Node A (10.0.1.0/24)         Node B (10.0.2.0/24)
├── Pod A1: 10.0.1.5     →→→ Node B routes to → Pod B1: 10.0.2.5
├── Pod A2: 10.0.1.6          └── Pod B2: 10.0.2.6
```

**CNI (Container Network Interface)** plugins implement this model:
- **Flannel:** Simple VXLAN overlay network. Encapsulates pod packets in UDP, sends across nodes.
- **Calico:** Uses BGP for routing. No overlay, native L3 routing. Better performance.
- **Cilium:** eBPF-based. Bypasses iptables entirely. High performance + network policy.
- **AWS VPC CNI:** Assigns real VPC IPs to pods. Native ENI-based routing. No overlay overhead.

---

## Q62. How does traffic flow inside a Kubernetes cluster? (Pod-to-Pod, Pod-to-Service, External-to-Service)

**Answer:**

**Scenario 1 — Pod-to-Pod on the SAME node:**

```
Pod A (10.0.1.5) → veth pair → cbr0 (bridge) → veth pair → Pod B (10.0.1.6)
```

Pure L2 switching via a virtual bridge (cbr0 or cni0). No kernel routing needed.

---

**Scenario 2 — Pod-to-Pod on DIFFERENT nodes (with Calico/BGP):**

```
Pod A (10.0.1.5)
  → Node A's eth0 (10.0.0.1)
  → Routed via BGP-learned route to Node B (10.0.0.2)
  → Node B's veth interface
  → Pod B (10.0.2.5)
```

With VXLAN (Flannel): packet is encapsulated in a VXLAN UDP packet (port 4789), sent to node B, decapsulated, delivered to pod.

---

**Scenario 3 — Pod-to-Service:**

```
Pod A wants to reach Service "my-svc" (ClusterIP: 10.96.0.50, port 80)

1. Pod A sends packet to 10.96.0.50:80
2. iptables/IPVS on the NODE intercepts (kube-proxy set these rules)
3. DNAT: 10.96.0.50:80 → 10.0.2.5:8080 (one of the service's backend pods)
4. Packet delivered to Pod B (10.0.2.5:8080)
5. Response: SNAT back to 10.96.0.50:80 (connection tracking handles return path)
```

kube-proxy watches the API server for Service/Endpoint changes and writes iptables rules.

---

**Scenario 4 — External traffic to a Service (LoadBalancer type):**

```
Internet Client
  → AWS NLB (provisioned by Cloud Controller Manager)
  → Node's NodePort (e.g., 30080)
  → kube-proxy iptables DNAT → Pod's actual IP:port
  → Pod
  → Response: SNAT back to NLB
  → Client
```

**With Ingress Controller (NGINX):**

```
Internet Client
  → AWS NLB (L4, routes to all nodes)
  → NGINX Ingress Pod (running on some node)
  → NGINX reads Ingress rules → routes to correct backend Service
  → Backend Pod
```

---

## Q63. What are Kubernetes Services? Explain each type.

**Answer:**

A Service provides a **stable virtual IP + DNS name** for a set of pods selected by labels — abstracting away individual pod IPs that change on restart.

```
┌──────────────────────────────────────────────────────────────┐
│  Service Types                                                │
│                                                              │
│  ClusterIP ──────── internal only                            │
│  NodePort ─────────────────────── node's IP:port             │
│  LoadBalancer ──────────────────────────────── External LB   │
│  ExternalName ─────────── DNS CNAME to external service      │
└──────────────────────────────────────────────────────────────┘
```

**1. ClusterIP (default):**

```yaml
spec:
  type: ClusterIP
  clusterIP: 10.96.0.50     # Virtual IP — only reachable inside cluster
  selector:
    app: my-app
  ports:
  - port: 80                # Service port
    targetPort: 8080        # Container port
```

- Accessible only within the cluster
- DNS: `my-svc.default.svc.cluster.local` → 10.96.0.50

**2. NodePort:**

```yaml
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 8080
    nodePort: 30080          # Opens on ALL nodes (30000-32767 range)
```

- External access: `<any-node-ip>:30080`
- Not for production (bypasses load balancer, exposes node IPs)

**3. LoadBalancer:**

```yaml
spec:
  type: LoadBalancer
  # Cloud creates an external LB (AWS NLB/CLB) automatically
```

- Cloud provisions an external load balancer
- On AWS: provisions an NLB or CLB, gets an external DNS/IP
- Production-grade external access

**4. ExternalName:**

```
spec:
  type: ExternalName
  externalName: my-database.rds.amazonaws.com
```

- DNS CNAME alias to external service
- Lets pods use a K8s Service name to reach external resources
- Zero proxying — pure DNS aliasing

**5. Headless Service (special ClusterIP = None):**

```
spec:
  clusterIP: None
  selector:
    app: postgres
```

- No virtual IP created
- DNS returns **all pod IPs** directly (A record per pod)
- Used by StatefulSets for stable pod DNS

---

## Q64. What is Ingress and how does it differ from a LoadBalancer Service?

**Answer:**

**LoadBalancer Service:** One load balancer per service. If you have 50 services → 50 load balancers → 50 AWS NLBs → expensive + complex.

**Ingress:** One load balancer for ALL HTTP/HTTPS traffic. Routes based on hostname and path.

```
Internet
    ↓
AWS NLB (1 NLB total)
    ↓
NGINX Ingress Controller (Pods)
    ↓ (reads Ingress rules)
    ├── api.myapp.com/v1/* → api-service:80
    ├── api.myapp.com/v2/* → api-v2-service:80
    └── app.myapp.com/*    → frontend-service:80
```

**Ingress resource (rules):**

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: main-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    cert-manager.io/cluster-issuer: "letsencrypt-prod"
spec:
  ingressClassName: nginx
  tls:
  - hosts: ["api.myapp.com"]
    secretName: api-tls-secret
  rules:
  - host: api.myapp.com
    http:
      paths:
      - path: /v1
        pathType: Prefix
        backend:
          service:
            name: api-v1-service
            port: { number: 80 }
      - path: /v2
        pathType: Prefix
        backend:
          service:
            name: api-v2-service
            port: { number: 80 }
```

**Ingress Controller options:**

- **NGINX Ingress** (most common, community)
- **AWS ALB Ingress Controller** (uses native AWS ALB, integrates with WAF, ACM)
- **Traefik** (automatic SSL, middleware ecosystem)
- **HAProxy Ingress**
- **Kong Ingress** (API gateway features)

---

## Q65. What is a NetworkPolicy? How does it work?

**Answer:**

NetworkPolicy is **firewall rules for pods** — it controls which pods can communicate with which other pods/namespaces/external IPs.

**Mental Model:**

By default, Kubernetes has **no firewall** between pods — every pod can talk to every other pod. NetworkPolicy adds security perimeter.

**Default behavior without NetworkPolicy:** All traffic allowed (open floor plan — anyone can walk to anyone's desk).

**With NetworkPolicy:**

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-frontend-to-api
  namespace: default
spec:
  podSelector:            # This policy applies TO: pods with app=api
    matchLabels:
      app: api
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - podSelector:        # Allow FROM: pods with app=frontend
        matchLabels:
          app: frontend
    - namespaceSelector:  # AND from: monitoring namespace
        matchLabels:
          name: monitoring
    ports:
    - protocol: TCP
      port: 8080
  egress:
  - to:
    - podSelector:
        matchLabels:
          app: postgres
    ports:
    - protocol: TCP
      port: 5432
```

**Key behavior:**

- Once a NetworkPolicy selects a pod, **all traffic not explicitly allowed is denied**
- `podSelector: {}` (empty) selects ALL pods in namespace
- NetworkPolicy requires a CNI that supports it (Calico, Cilium, Weave Net — Flannel does NOT support NetworkPolicy)

**Deny-all starting policy (security baseline):**

```yaml
spec:
  podSelector: {}   # Applies to all pods
  policyTypes: ["Ingress", "Egress"]
  # No rules = deny all ingress and egress
```

---

### Q66. Explain CoreDNS — how does DNS work inside Kubernetes?

**Answer:**

CoreDNS is the **cluster DNS server** — every pod's `/etc/resolv.conf` points to it. It translates service names into ClusterIP addresses.

**Pod's DNS configuration (auto-injected by kubelet):**

```
# /etc/resolv.conf inside any pod
nameserver 10.96.0.10        # CoreDNS ClusterIP
search default.svc.cluster.local svc.cluster.local cluster.local
options ndots:5
```

**DNS resolution flow:**

```
Pod wants to reach: "api-service"
    ↓
/etc/resolv.conf: nameserver = 10.96.0.10 (CoreDNS)
    ↓
CoreDNS receives query for "api-service"
    ↓
ndots:5 → appends search domains:
  1. api-service.default.svc.cluster.local → FOUND → return 10.96.5.20
    ↓
Pod gets IP 10.96.5.20 → connects to service
```

**DNS record formats:**

```
Service: my-svc.my-namespace.svc.cluster.local → ClusterIP
Pod:     10-0-1-5.default.pod.cluster.local   → Pod IP (if enabled)
StatefulSet pod: postgres-0.postgres-headless.default.svc.cluster.local
```

**Short names work because of search domains:**

- `api-service` → resolves if in same namespace
- `api-service.other-ns` → resolves cross-namespace
- `api-service.other-ns.svc.cluster.local` → fully qualified, always works

**CoreDNS is configurable via ConfigMap:**

```
# Forward DNS for external domains to upstream
.:53 {
  forward . 8.8.8.8 8.8.4.4
  cache 30
}
```

---

## SECTION D: Storage, Config & Secrets

---

### Q67. Explain Kubernetes storage — PV, PVC, StorageClass.

**Answer:**

```
StorageClass (How to provision)
    ↓ (dynamically creates)
PersistentVolume (The actual storage)
    ↓ (bound to)
PersistentVolumeClaim (Request for storage)
    ↓ (mounted in)
Pod
```

**Mental Model — Apartment analogy:**

- **StorageClass** = The property management company (AWS EBS provider). It knows how to build new apartments.
- **PersistentVolume** = An actual apartment unit (an EBS volume, NFS share).
- **PersistentVolumeClaim** = A lease application ("I need a 20GB apartment").
- **Pod** = The tenant (lives in the apartment = uses the storage).

**StorageClass (dynamic provisioning):**

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast-ssd
provisioner: ebs.csi.aws.com
parameters:
  type: gp3
  iops: "3000"
  encrypted: "true"
reclaimPolicy: Retain   # Delete or Retain when PVC is deleted
volumeBindingMode: WaitForFirstConsumer  # Don't provision until pod schedules
```

**PVC:**

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-data
spec:
  storageClassName: fast-ssd
  accessModes: [ReadWriteOnce]   # Only one pod can write at a time
  resources:
    requests:
      storage: 20Gi
```

**Access modes:**

- `ReadWriteOnce (RWO)` — One node reads+writes. (EBS, local disk)
- `ReadOnlyMany (ROX)` — Multiple nodes read. (S3 via CSI, NFS)
- `ReadWriteMany (RWX)` — Multiple nodes read+write. (EFS, NFS, Ceph)

**Reclaim policies:**

- `Delete` — EBS volume deleted when PVC deleted (default for dynamic)
- `Retain` — PV remains after PVC deleted (manual cleanup, protects data)

## Q68. What are ConfigMaps and Secrets? How are they consumed in pods?

**Answer:**

**ConfigMap** — stores non-sensitive configuration data (key-value pairs or files):

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  DATABASE_HOST: "postgres.default.svc.cluster.local"
  LOG_LEVEL: "info"
  app.properties: |
    server.port=8080
    feature.flag.enabled=true
```

**Secret** — stores sensitive data (base64-encoded, not encrypted by default):

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
data:
  password: cGFzc3dvcmQ=    # base64("password")
stringData:
  api_key: "my-real-key"    # auto-encoded by K8s
```

**Consuming in Pods — three methods:**

**Method 1 — Environment variables:**

```yaml
env:
- name: DB_HOST
  valueFrom:
    configMapKeyRef:
      name: app-config
      key: DATABASE_HOST
- name: DB_PASSWORD
  valueFrom:
    secretKeyRef:
      name: db-secret
      key: password
```

**Method 2 — envFrom (entire ConfigMap/Secret as env vars):**

```yaml
envFrom:
- configMapRef:
    name: app-config
- secretRef:
    name: db-secret
```

**Method 3 — Volume mount (files):**

```yaml
volumes:
- name: config-volume
  configMap:
    name: app-config
volumeMounts:
- name: config-volume
  mountPath: /etc/app
  readOnly: true
# Creates /etc/app/DATABASE_HOST, /etc/app/app.properties files
```

**Critical nuance:** ConfigMap/Secret updates propagate automatically to **volume mounts** (with ~1min delay). They do NOT auto-update **environment variables** — pod must be restarted.

**Secret encryption at rest:** By default, K8s stores secrets in etcd as base64 (not encrypted). Enable EncryptionConfiguration to encrypt with AES or KMS.

---

## SECTION E: RBAC, Security & Scheduling

---

### Q69. Explain Kubernetes RBAC — Role, ClusterRole, RoleBinding, ClusterRoleBinding.

**Answer:**
RBAC (Role-Based Access Control) controls **who can do what to which resources**.

**Four RBAC objects:**

| Object | Scope | What it defines |
|---|---|---|
| `Role` | Namespace | Permissions within ONE namespace |
| `ClusterRole` | Cluster-wide | Permissions across ALL namespaces (or non-namespaced resources) |
| `RoleBinding` | Namespace | Grants a Role to a subject within a namespace |
| `ClusterRoleBinding` | Cluster-wide | Grants a ClusterRole to a subject cluster-wide |

**Mental Model:**
- `Role` = Job description ("can read pods in the marketing department")
- `RoleBinding` = HR letter assigning that job description to a specific employee
- `ClusterRole` = Company-wide job title ("can read pods ANYWHERE in the company")
- `ClusterRoleBinding` = Granting company-wide authority to someone

**Example — Read-only access to pods in the `production` namespace:**

```
# Role (what can be done)
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-reader
  namespace: production
rules:
- apiGroups: [""]          # "" = core API group
  resources: ["pods", "pods/log"]
  verbs: ["get", "list", "watch"]

---
# RoleBinding (who gets it)
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods-binding
  namespace: production
subjects:
- kind: User
  name: jane@company.com
  apiGroup: rbac.authorization.k8s.io
- kind: ServiceAccount
  name: monitoring-sa
  namespace: monitoring
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

**Subjects can be:**

- `User` — human user (authenticated via OIDC, certs)
- `Group` — group of users
- `ServiceAccount` — identity for pods/controllers (in-cluster)

---

### Q70. What are Taints and Tolerations? How do they work with Node Affinity?

**Answer:**

These are scheduling controls — ways to **attract or repel pods from nodes**.

**Taints (on nodes) — repel pods:**

```
kubectl taint nodes gpu-node-1 hardware=gpu:NoSchedule
# NoSchedule: Don't schedule new pods without toleration
# PreferNoSchedule: Prefer not to, but allow if necessary
# NoExecute: Evict existing pods + don't schedule new ones
```

**Tolerations (on pods) — allow landing on tainted nodes:**

```
spec:
  tolerations:
  - key: "hardware"
    operator: "Equal"
    value: "gpu"
    effect: "NoSchedule"
```

**Mental Model:**

- Taint = "No Visitors" sign on a node

- Toleration = VIP badge that lets you past the "No Visitors" sign

**Node Affinity (attract pods to specific nodes):**

```
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:  # HARD rule
        nodeSelectorTerms:
        - matchExpressions:
          - key: node-type
            operator: In
            values: ["compute-optimized"]
      preferredDuringSchedulingIgnoredDuringExecution: # SOFT rule
      - weight: 1
        preference:
          matchExpressions:
          - key: zone
            operator: In
            values: ["us-east-1a"]
```

**Pod Anti-Affinity (spread pods across nodes/zones):**

```
affinity:
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchLabels:
          app: api-server
      topologyKey: kubernetes.io/hostname   # No two api-server pods on same node
```

**Combined pattern for GPU workloads:**

1. Taint GPU nodes: `hardware=gpu:NoSchedule` (prevents CPU-only pods from wasting GPU nodes)
2. ML pods have toleration for `hardware=gpu`
3. ML pods have nodeAffinity for `hardware=gpu` (actively attracts them to GPU nodes)

---

## Q71. What are Resource Requests and Limits? What happens when you exceed them?

**Answer:**

```
resources:
  requests:              # What the scheduler uses to find a node
    memory: "256Mi"
    cpu: "250m"          # 250 millicores = 0.25 CPU
  limits:                # Hard ceiling — enforcement at runtime
    memory: "512Mi"
    cpu: "500m"
```

**Requests vs Limits:**

- **Request** = Guaranteed minimum. The scheduler finds a node with at least this much available. The container *always* gets this resource.
- **Limit** = Maximum allowed. Runtime enforcement.

**What happens when you exceed limits:**

- **CPU limit exceeded:** Container is **throttled** (slowed down). NOT killed. CPU is compressible.
- **Memory limit exceeded:** Container is **OOMKilled** (killed by kernel OOM killer). Memory is not compressible.

**OOMKilled flow:**

```
Container uses 513Mi memory → exceeds 512Mi limit
→ Linux kernel OOM killer sends SIGKILL
→ Container terminates with exit code 137
→ kubectl describe pod → "OOMKilled"
→ kubelet restarts container (based on restartPolicy)
```

**QoS Classes (affects eviction order during node pressure):**

| Class | Condition | Evicted first? |
|---|---|---|
| **Guaranteed** | requests == limits for ALL containers | Last (most protected) |
| **Burstable** | requests set, limits > requests | Middle |
| **BestEffort** | NO requests or limits set | First (most likely evicted) |

**Production rule:** Never deploy to production without setting requests AND limits. No requests = scheduler has no info = bad placement decisions.

# SECTION F: Autoscaling & High Availability

## Q72. Explain HPA, VPA, and KEDA — when to use each.

**Answer:**

**HPA (Horizontal Pod Autoscaler) — more pods:**

```
apiVersion: autoscaling/v2
kind: HPA
spec:
  scaleTargetRef:
    kind: Deployment
    name: my-api
  minReplicas: 2
  maxReplicas: 20
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70  # Scale when avg CPU > 70%
  - type: Pods
    pods:
      metric:
        name: http_requests_per_second
      target:
        type: AverageValue
        averageValue: 1000      # Custom metric from Prometheus
```

HPA queries the Metrics Server (or Prometheus via adapter) every 15s. If metric exceeds target, adds pods. Scale-down is slower (5-min cooldown by default) to prevent flapping.

**VPA (Vertical Pod Autoscaler) — bigger pods:**

- Adjusts CPU/memory requests automatically based on actual usage
- Good for: single-threaded apps that can't scale horizontally, databases
- **Cannot be used with HPA on CPU/memory at the same time** (conflict)
- Requires pod restart to apply new resource values (disruptive)

**KEDA (Kubernetes Event-Driven Autoscaling):**

- Scales based on external event sources (Kafka queue depth, SQS queue length, Prometheus queries, Redis list length, cron schedule)
- Scales to **zero** (HPA minimum is 1)
- Perfect for: batch processors, event-driven microservices, async workers

```
# KEDA scales Deployment based on SQS queue depth
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
spec:
  scaleTargetRef:
    name: order-processor
  minReplicaCount: 0
  maxReplicaCount: 50
  triggers:
  - type: aws-sqs-queue
    metadata:
      queueURL: https://sqs.us-east-1.amazonaws.com/123/orders
      targetQueueLength: "10"   # 1 pod per 10 messages
```

## Q73. What is PodDisruptionBudget (PDB)?

**Answer:**
PDB ensures a **minimum number of pods remain available** during voluntary disruptions (node drain, rolling updates, cluster upgrades).

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: api-pdb
spec:
  minAvailable: 2          # OR: maxUnavailable: 1
  selector:
    matchLabels:
      app: api-server
```

**Mental Model:**
PDB is a **service level guarantee** during maintenance. It says: "Even if someone is draining nodes for upgrades, always keep at least 2 of my API pods running."

**How it works with `kubectl drain`:**

1. `kubectl drain node-3` starts evicting pods from node-3
2. Eviction API checks PDB before evicting each pod
3. If evicting the pod would violate the PDB (only 2 api pods left, PDB says minAvailable=2), eviction is BLOCKED
4. Drain waits until other pods are rescheduled elsewhere before evicting this one

**Production requirement:** Every production Deployment with >1 replica should have a PDB.

---

# SECTION G: Observability & Troubleshooting

---

## Q74. Walk through how you troubleshoot a pod stuck in `CrashLoopBackOff`.

**Answer:**
CrashLoopBackOff = the container starts, crashes, Kubernetes restarts it, it crashes again. Back-off delay increases exponentially (10s, 20s, 40s, 80s... up to 5min).

**Systematic diagnostic process:**

```
# Step 1: Get pod status and events
kubectl describe pod <pod-name>
# Look for: Events section (OOMKilled, ImagePullBackOff, failed mounts)
# Look for: Last State (previous container's exit code)

# Step 2: Check current logs
kubectl logs <pod-name>

# Step 3: Check previous container logs (the one that crashed)
kubectl logs <pod-name> --previous

# Step 4: Check exit code
kubectl get pod <pod-name> -o jsonpath='{.status.containerStatuses[0].lastState.terminated.exitCode}'
```

**Exit codes → Root cause:**

| Exit Code | Meaning |
|---|---|
| `0` | Successful exit (shouldn't crashloop) |
| `1` | Application error — check logs |
| `137` | OOMKilled (SIGKILL — 128+9) |
| `139` | Segmentation fault |
| `143` | SIGTERM (graceful shutdown — pod terminated) |

**Common CrashLoopBackOff causes:**

1. **Application error on startup** (wrong config, missing env var) → fix app config/secrets
2. **OOMKilled** (memory limit too low) → increase memory limit
3. **Missing ConfigMap/Secret** → `kubectl describe pod` shows failed mount
4. **Wrong command/args** → check `command` and `args` in pod spec
5. **Liveness probe failing too aggressively** → increase `initialDelaySeconds`
6. **Missing dependencies** (DB not ready) → add init containers or retry logic

**Temporary debug trick:**

```
# Override command to keep pod running, then exec in
kubectl run debug --image=myapp:latest --command -- sleep 3600
kubectl exec -it debug -- /bin/sh
# Now manually run the app and see the real error
```

---

## Q75. What are Liveness, Readiness, and Startup probes?

**Answer:**

Three probe types that Kubernetes uses to understand pod health:

**Liveness Probe — "Is the app alive?"**

```
livenessProbe:
  httpGet:
    path: /healthz
    port: 8080
  initialDelaySeconds: 30    # Wait 30s before first check
  periodSeconds: 10          # Check every 10s
  failureThreshold: 3        # Kill+restart after 3 consecutive failures
```

- Failure action: **Container is killed and restarted**
- Use case: Detect deadlocks (app is running but stuck)

**Readiness Probe — "Is the app ready to receive traffic?"**

```
readinessProbe:
  httpGet:
    path: /ready
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 5
  failureThreshold: 3
```

- Failure action: **Pod removed from Service endpoints** (no traffic sent to it)
- Pod is NOT killed — it just stops receiving traffic
- Use case: App is still loading caches, waiting for DB connection

**Startup Probe — "Is the app still starting up?"**

```
startupProbe:
  httpGet:
    path: /healthz
    port: 8080
  failureThreshold: 30       # Allow up to 30 * 10s = 5 min to start
  periodSeconds: 10
```

- Disables liveness/readiness probes until it succeeds
- Use case: Slow-starting apps (JVMs, apps with large data loading) that would fail liveness probes during startup

**Mental Model:**

- **Liveness** = "Are you alive?" (if no → replace you)
- **Readiness** = "Are you ready to work?" (if no → take you off the schedule, but don't fire you)
- **Startup** = "Are you done getting ready for your first shift?" (gives you extra time to get ready)

---

### Q76. How does Kubernetes handle graceful shutdown?

**Answer:**

When a pod is terminated (by deployment update, scaling down, node drain):

```
1. Pod enters "Terminating" state
2. Pod removed from Service endpoints (readiness probe fails OR endpoint removed)
   → New requests stop being routed to this pod
3. SIGTERM sent to container (PID 1)
   → App should start graceful shutdown
4. terminationGracePeriodSeconds countdown starts (default: 30s)
5. If container still running after grace period → SIGKILL sent
```

**Production checklist for graceful shutdown:**

```yaml
spec:
  terminationGracePeriodSeconds: 60    # Give app 60s to finish requests
  containers:
  - name: api
    lifecycle:
      preStop:
        exec:
          command: ["sleep", "5"]    # Wait 5s for LB to drain
```

**The preStop hook solves the race condition:**

Even after readiness probe failure, the load balancer may take a few seconds to stop routing. The `preStop` sleep ensures the pod waits before starting shutdown, preventing mid-request termination.

---

## SECTION H: Helm & GitOps

---

### Q77. What is Helm and what problem does it solve?

**Answer:**

Helm is the **package manager for Kubernetes** — it templates, packages, and manages K8s manifests.

**Problem without Helm:**

- 20 YAML files for one application (Deployment, Service, Ingress, ConfigMap, Secret, HPA, PDB, etc.)
- Each environment (dev/staging/prod) needs slightly different values
- No versioning, no rollback mechanism

**What Helm provides:**

1. **Charts** — packaged K8s manifests with Go templating
2. **Values** — environment-specific overrides
3. **Releases** — tracked installations with revision history
4. **Rollback** — `helm rollback` to previous chart version

**Chart structure:**

```
mychart/
├── Chart.yaml          # Metadata (name, version, description)
├── values.yaml         # Default values
├── values-prod.yaml    # Prod overrides
└── templates/
    ├── deployment.yaml
    ├── service.yaml
    ├── ingress.yaml
    └── _helpers.tpl    # Named templates (like functions)
```

**Template example:**

```
# templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "mychart.fullname" . }}
  labels:
    {{- include "mychart.labels" . | nindent 4 }}
spec:
  replicas: {{ .Values.replicaCount }}
  template:
    spec:
      containers:
      - name: {{ .Chart.Name }}
        image: "{{ .Values.image.repository }}:{{ .Values.image.tag | default .Chart.AppVersi
on }}"
        resources:
          {{- toYaml .Values.resources | nindent 12 }}
```

**Deploy commands:**

```
helm install my-release ./mychart -f values-prod.yaml
helm upgrade my-release ./mychart -f values-prod.yaml --atomic
helm rollback my-release 1
helm uninstall my-release
helm list -A                              # All releases all namespaces
```

## Q78. Explain ArgoCD — how does GitOps work in Kubernetes?

**Answer:**

ArgoCD is a **declarative GitOps CD tool** for Kubernetes. It continuously syncs cluster state to match the desired state defined in Git.

**GitOps principles:**

1. Git is the single source of truth for desired cluster state
2. Changes to infra go through Git (PRs, reviews, audit trail)
3. A controller continuously reconciles actual state with desired state
4. Deployment = a Git commit, not a Jenkins job that runs kubectl

**ArgoCD architecture:**

```
Git Repo (desired state)
    ↓ watches
ArgoCD Application Controller
    ↓ compares
Kubernetes Cluster (actual state)
    ↓ if drift detected
ArgoCD syncs → applies manifests → cluster matches Git
```

**Application definition:**

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: my-api
  namespace: argocd
spec:
  project: default
  source:
    repoURL: https://github.com/myorg/k8s-manifests
    targetRevision: HEAD
    path: apps/my-api/overlays/prod
  destination:
    server: https://kubernetes.default.svc
    namespace: production
  syncPolicy:
    automated:
      prune: true       # Delete resources not in Git
      selfHeal: true    # Revert manual changes to cluster
    syncOptions:
    - CreateNamespace=true
```

**CI/CD with ArgoCD:**

```
Developer pushes code
    ↓
CI pipeline: build image → push to ECR → update image tag in Git (values.yaml)
    ↓
ArgoCD detects Git change
    ↓
ArgoCD syncs → rolls out new Deployment
```

---

## SECTION I: Networking Deep Dive

---

### Q79. What is kube-proxy and what are its modes?

**Answer:**

kube-proxy runs on every node. It watches the API server for Service and Endpoint changes and **programs networking rules** so that traffic to a Service's ClusterIP is load-balanced to the backend pods.

**Three modes:**

**1. iptables mode (default):**

- Creates iptables rules (NAT table, PREROUTING chain) for every Service
- Stateless rules — random pod selection per connection
- **Problem:** With 10,000 Services → 10,000 iptables rules → O(n) lookup time → latency spike
- Rules are not updated in-place — full iptables replacement on every change

**2. IPVS mode (recommended for large clusters):**

- Uses Linux IPVS (IP Virtual Server) — in-kernel L4 load balancer
- Hash table lookup → **O(1) regardless of service count**
- Rich LB algorithms: round-robin, least-connection, shortest expected delay
- Enable: `--proxy-mode=ipvs` in kube-proxy config

**3. eBPF mode (via Cilium — bypasses kube-proxy entirely):**

- Programs eBPF maps in kernel
- No iptables, no IPVS
- Fastest, but requires Cilium CNI

**Traffic flow with iptables:**

```
Pod sends packet to 10.96.0.50:80 (ClusterIP)
→ iptables PREROUTING hook
→ DNAT rule: 10.96.0.50:80 → randomly selects backend pod IP (e.g., 10.0.2.5:8080)
→ Packet routed to 10.0.2.5 (backend pod's node)
→ Pod receives request on port 8080
→ Response → SNAT back (connection tracking handles return path)
```

## Q80. What networking knowledge is essential for a DevOps engineer working with Kubernetes?

**Answer:**

This is the consolidated "must-know networking" list:

**Layer 3/4 Fundamentals:**

- CIDR notation and subnetting (pods get IPs from pod CIDR, e.g., 10.244.0.0/16)
- TCP/UDP, port numbers, connection states (ESTABLISHED, TIME_WAIT)
- NAT (SNAT, DNAT) — how kube-proxy translates IPs
- Routing tables — how cross-node traffic is delivered

**Kubernetes-specific networking:**

- Pod CIDR vs Service CIDR vs Node CIDR (three non-overlapping ranges required)
- CNI: what it does, how to choose one for your environment
- DNS resolution (search domains, ndots, CoreDNS)
- Service types and when to use each (ClusterIP, NodePort, LB, ExternalName)
- Ingress: host-based routing, path-based routing, TLS termination

- NetworkPolicy: default-deny approach, selector semantics
- Headless Services for StatefulSet pod DNS

**Load balancing:**

- L4 vs L7 load balancing (NLB vs ALB in AWS)
- Sticky sessions (sessionAffinity: ClientIP in Services)
- Connection draining on pod termination

**AWS-specific (for EKS):**

- VPC CNI: pods get native VPC IPs (from ENIs), no overlay
- AWS ALB Ingress Controller: creates ALB per Ingress or shares one
- Security Groups for Pods (SGfP): pod-level security group enforcement
- Private vs public endpoint for EKS API server

---

## SECTION J: Advanced Kubernetes Topics

---

### Q81. What are Init Containers and why are they useful?

**Answer:**

Init containers run **sequentially to completion BEFORE main containers start**. They're used for setup, waiting for dependencies, or populating shared volumes.

```
spec:
  initContainers:
  - name: wait-for-db
    image: busybox
    command: ['sh', '-c',
      'until nc -z postgres-service 5432; do echo waiting; sleep 2; done']
  - name: db-migrate
    image: myapp:latest
    command: ['python', 'manage.py', 'migrate']
    env:
    - name: DATABASE_URL
      valueFrom:
        secretKeyRef:
          name: db-secret
          key: url
  containers:
  - name: app
    image: myapp:latest
    # Starts only AFTER both init containers complete successfully
```

**Key properties:**

- Run in order (first completes → second starts → main starts)
- If an init container fails → it restarts until it succeeds (or backoffLimit)

- Different image than main container (run a DB migration tool, not the app image)
- Share volumes with main containers (pre-populate config files)

**Common use cases:**
- Wait for database to be ready (`nc`, `pg_isready`)
- Run database migrations
- Clone git repository into a shared volume
- Fetch secrets and write to shared volume
- Register service in external registry

---

## Q82. What is a Sidecar container? (And upcoming native Sidecar in K8s 1.28+)

**Answer:**
A sidecar is a **secondary container in a pod** that enhances or extends the main container without being part of its core logic.

**Classic patterns:**

```
containers:
- name: app              # Main: serves HTTP traffic
  image: my-api
- name: log-shipper      # Sidecar: forwards logs to ELK
  image: fluentd
- name: envoy            # Sidecar: service mesh proxy
  image: envoyproxy/envoy
```

All share the pod's network (communicate via localhost) and can share volumes.

**Native Sidecar (K8s 1.28 beta):**
Previously, if a sidecar outlived the main container (Job scenario), the Job wouldn't complete. K8s 1.28 introduced `restartPolicy: Always` at the container level for sidecars — they start before main containers and are terminated after main containers exit. This enables proper sidecar lifecycle in Jobs.

```
initContainers:
- name: log-collector
  image: fluentd
  restartPolicy: Always    # New: native sidecar — starts before app, ends after app
containers:
- name: app
  image: my-job
```

---

## Q83. Explain Kubernetes Admission Controllers.

**Answer:**
Admission controllers are **plugins in the API server pipeline** that intercept requests AFTER authentication

and authorization but BEFORE persistence to etcd. They can **mutate** (modify) or **validate** (allow/reject) requests.

```
Request → Auth → AuthZ → Admission Controllers → etcd
```

**Built-in controllers (important ones):**

| Controller | What it does |
|---|---|
| `NamespaceLifecycle` | Prevents creation in terminating namespaces |
| `LimitRanger` | Enforces resource limits on pods/containers |
| `ResourceQuota` | Enforces namespace-level resource quotas |
| `PodSecurity` | Enforces Pod Security Standards (replaced PSP) |
| `DefaultStorageClass` | Adds default storage class to PVCs |
| `MutatingAdmissionWebhook` | Calls external webhook to mutate objects |
| `ValidatingAdmissionWebhook` | Calls external webhook to validate/reject objects |

**Webhook-based admission (most powerful):**

```yaml
# MutatingWebhookConfiguration — inject Istio sidecar
apiVersion: admissionregistration.k8s.io/v1
kind: MutatingWebhookConfiguration
metadata:
  name: istio-sidecar-injector
webhooks:
- name: sidecar-injector.istio.io
  rules:
  - apiGroups: [""]
    resources: ["pods"]
    operations: ["CREATE"]
  clientConfig:
    service:
      name: istiod
      namespace: istio-system
      path: /inject
```

**Real-world webhook use cases:**
- **Istio:** Inject Envoy sidecar into every pod
- **Kyverno/OPA Gatekeeper:** Policy enforcement (require labels, block privileged containers)
- **Vault Agent Injector:** Auto-inject Vault secrets into pods

---

## Q84. What are Kubernetes Custom Resources (CRD) and Operators?

**Answer:**

**CRD (Custom Resource Definition):**
Extends the Kubernetes API with custom object types. You can define your own `kind` (like `kind: Database`, `kind: RedisCluster`) and manage them via kubectl.

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: databases.mycompany.com
spec:
  group: mycompany.com
  versions:
  - name: v1
    served: true
    storage: true
    schema:
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            properties:
              engine: { type: string }
              size: { type: string }
  scope: Namespaced
  names:
    plural: databases
    singular: database
    kind: Database
```

**Operator = CRD + Controller:**

An Operator is a pattern where you:

1. Define a custom resource (CRD) for your stateful app
2. Write a controller that watches those resources
3. The controller reconciles the desired state (CR spec) with actual state

**Mental Model:**

An Operator is a **human operator's knowledge encoded as software**. A DBA knows how to: provision a new database, take backups, handle failover, perform upgrades. An Operator (like CrunchyData PGO for Postgres) does this automatically by watching `kind: PostgresCluster` objects.

**Examples of real Operators:**

- **cert-manager** (manages TLS certificates)
- **Prometheus Operator** (manages Prometheus via CRs)
- **ArgoCD** (manages GitOps deployments)
- **Strimzi** (manages Kafka)
- **CrunchyData PGO** (manages PostgreSQL)

---

## Q85. How does Pod Security work in Kubernetes (Pod Security Admission)?

**Answer:**

Pod Security Admission (PSA, K8s 1.25+, replaced deprecated PodSecurityPolicy) enforces security standards at the namespace level.

**Three pod security standards:**

| Standard | What it enforces |
|---|---|
| `privileged` | No restrictions |
| `baseline` | Blocks known privilege escalations (no privileged containers, no host PID) |
| `restricted` | Strongest — requires non-root, drops all capabilities, seccomp |

**Three modes:**

- `enforce` — reject violating pods
- `audit` — log violations but allow
- `warn` — warn at admission but allow

**Namespace label:**

```
apiVersion: v1
kind: Namespace
metadata:
  name: production
  labels:
    pod-security.kubernetes.io/enforce: restricted
    pod-security.kubernetes.io/warn: restricted
    pod-security.kubernetes.io/audit: restricted
```

**Compliant pod spec for `restricted`:**

```
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
  - name: app
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: ["ALL"]
      runAsUser: 1000
```

## Q86. What is a Service Account and how do pods use them?

**Answer:**

A ServiceAccount is a **non-human identity for pods** — it allows pods to authenticate to the Kubernetes API server and external services.

**Default behavior:** Every pod gets the `default` ServiceAccount in its namespace automatically. Its token is mounted at `/var/run/secrets/kubernetes.io/serviceaccount/token`.

**Custom ServiceAccount:**

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: monitoring-sa
  namespace: production
  annotations:
    eks.amazonaws.com/role-arn: arn:aws:iam::123456789:role/MonitoringRole  # IRSA
---
# Bind ServiceAccount to pod
spec:
  serviceAccountName: monitoring-sa
```

**IRSA (IAM Roles for Service Accounts) on EKS:**

This is how pods securely access AWS services without hardcoding credentials:

```
Pod (with monitoring-sa) → requests AWS token via OIDC
→ AWS STS verifies: "Is this really monitoring-sa in EKS cluster XYZ?"
→ Yes → Issues temporary AWS credentials for MonitoringRole
→ Pod can now call AWS APIs (S3, DynamoDB, etc.)
```

**No env vars, no secrets, fully automatic, time-limited credentials.**

---

## Q87. What is Kubernetes Federation and multi-cluster management?

**Answer:**

Kubernetes Federation allows managing multiple Kubernetes clusters as one unit. Modern tools have replaced classic Federation v1.

**Common multi-cluster patterns:**

**1. Active-Active (regional distribution):**

```
us-east-1 cluster ──┐
                    ├── Global Load Balancer (AWS Route 53 / Cloudflare)
eu-west-1 cluster ──┘
```

Same app runs in both clusters. Traffic routed by geography. Failure in one region → traffic shifts to other.

**2. Hub-and-Spoke (ArgoCD pattern):**

```
Management cluster (ArgoCD)
├── deploys to → Production cluster (us-east-1)
├── deploys to → Staging cluster
└── deploys to → DR cluster (us-west-2)
```

**Tools:**

- **ArgoCD** (GitOps multi-cluster deployment)

- **Flux** (GitOps multi-cluster)

- **Kubefed** (Kubernetes Federation v2)
- **Rancher / Tanzu** (commercial multi-cluster management)
- **Cluster API** (provision K8s clusters declaratively, similar to Terraform for clusters)

---

## Q88. How does container image pulling work in Kubernetes?

**Answer:**

**imagePullPolicy options:**

```
image: myapp:1.2.3
imagePullPolicy: IfNotPresent  # Pull only if not on node (default for tags)
# Always   → Always pull from registry (default for :latest)
# Never    → Never pull — image must exist on node
```

**Production rule:** Never use `:latest` in production. Always use specific digest or version tag. `imagePullPolicy: Always` with `:latest` = slow pod startup + potential breaking changes.

**Private registry authentication:**

```
kubectl create secret docker-registry regcred \
  --docker-server=123456789.dkr.ecr.us-east-1.amazonaws.com \
  --docker-username=AWS \
  --docker-password=$(aws ecr get-login-password)
```

```
spec:
  imagePullSecrets:
  - name: regcred
```

**On EKS:** Use the ECR credential helper or IAM role — pods don't need imagePullSecrets if the node IAM role has ECR pull permissions.

---

## Q89. What is the difference between `kubectl apply` and `kubectl create`?

**Answer:**

| Command | Behavior |
|---|---|
| `kubectl create` | Creates resource. Fails if it already exists. |
| `kubectl apply` | Creates if doesn't exist, updates if exists (3-way merge). |
| `kubectl replace` | Deletes and recreates. Destructive. |
| `kubectl patch` | Partial update using JSON Merge Patch or Strategic Merge Patch. |

**`kubectl apply` — 3-way merge:**
Apply stores a `last-applied-configuration` annotation on the object. On next apply:
1. Last applied config (what you applied before)
2. Current live config (what's in the cluster)
3. New config (what you're applying now)

Merge: add fields in new, remove fields deleted between last and new, keep fields modified in cluster but not in config.

`kubectl apply` **is idempotent** — running it multiple times is safe. This is why it's used in CI/CD, not `create`.

---

### Q90. How do you perform zero-downtime database schema migrations in Kubernetes?

**Answer:**
This is an operations/architecture question that separates practitioners from theorists.

**The problem:**
- Rolling update has both old (v1) and new (v2) app pods running simultaneously
- If v2 requires a new DB schema, v1 breaks with the new schema
- If you migrate schema first, v1 may fail with new schema

**Solution — Expand/Contract pattern (3-phase):**

**Phase 1 — Expand (backward-compatible migration):**
- Add new column as nullable (or with default)
- Old app (v1) ignores new column — works fine
- DB migration runs in init container or job before app update
- **No downtime**

**Phase 2 — Deploy new app version:**
- Rolling update v1 → v2
- v2 uses new column
- During rollout, both v1 and v2 pods run simultaneously — both work with the schema

**Phase 3 — Contract:**

- Once all v1 pods are gone, apply cleanup migration (make column non-null, drop old columns)

**Kubernetes implementation:**

```yaml
# Job runs migration before Deployment update
apiVersion: batch/v1
kind: Job
metadata:
  name: db-migrate-v2
  annotations:
    argocd.argoproj.io/hook: PreSync   # Run before sync in ArgoCD
    argocd.argoproj.io/hook-delete-policy: HookSucceeded
spec:
  template:
    spec:
      initContainers:
      - name: wait-for-db
        image: busybox
        command: ['sh', '-c', 'until nc -z postgres 5432; do sleep 1; done']
      containers:
      - name: migrator
        image: myapp:v2
        command: ["python", "manage.py", "migrate", "--run-syncdb"]
```

## Q91–Q100: Final Senior-Level Topics

### Q91. What is etcd compaction and defragmentation, and when is it needed?

etcd stores all historical revisions of objects (for watch API efficiency). Over time this grows unboundedly. Compaction removes old revisions:

```bash
# Auto-compaction (recommended)
etcd --auto-compaction-mode=periodic --auto-compaction-retention=1h

# Manual compaction
etcdctl compact $(etcdctl endpoint status --write-out="json" | jq '.
[0].Status.header.revision')

# Defragmentation (reclaim disk space after compaction)
etcdctl defrag --endpoints=https://localhost:2379
```

Run defrag during low-traffic windows. It briefly locks etcd (causes API server hiccup).

### Q92. What is a Kubernetes Headless Service and when is it used?

```
spec:
  clusterIP: None   # No virtual IP assigned
  selector:
    app: postgres
```

DNS returns ALL pod IPs (A records per pod) instead of a single ClusterIP. StatefulSets use headless services for stable pod DNS:
- `postgres-0.postgres-headless.default.svc.cluster.local` → Pod 0's IP
- Required for: Cassandra (self-discovery), Kafka (broker addressing), Postgres streaming replication

---

### Q93. How does Kubernetes handle rolling back a failed deployment automatically?

It doesn't automatically — Kubernetes doesn't know if your app "works" (that's your job with probes). But with proper configuration, failed rollouts surface quickly:

```
# Detect a failed rollout in CI
kubectl rollout status deployment/my-app --timeout=300s
# Exit code non-zero if rollout fails

# Manual rollback
kubectl rollout undo deployment/my-app

# Automatic rollback with Argo Rollouts
# Argo Rollouts + Prometheus: if error rate > 5% → automatic rollback
```

Argo Rollouts + analysis templates give you **automatic canary promotion/rollback** based on real metrics.

---

### Q94. What is `topologySpreadConstraints` and why is it better than pod anti-affinity?

```
spec:
  topologySpreadConstraints:
  - maxSkew: 1                    # Max difference in pod count between zones
    topologyKey: topology.kubernetes.io/zone
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        app: api-server
```

This evenly spreads pods across zones. Pod anti-affinity only prevents colocation — it doesn't guarantee even distribution. With 6 pods across 3 zones, anti-affinity might put 4 in zone-a and 1 each in zone-b and zone-c. `topologySpreadConstraints` enforces 2-2-2 distribution.

---

### Q95. What is Kubernetes Vertical Node Autoscaler (Cluster Autoscaler)?

Cluster Autoscaler (CA) automatically adjusts the NUMBER OF NODES:

- Scale up: pod is unschedulable (Pending) → CA adds node
- Scale down: node is underutilized for 10+ minutes → CA cordons + drains node

```
# Cluster Autoscaler annotation for nodegroup
cluster.k8s.io/min-size: "2"
cluster.k8s.io/max-size: "20"
```

On EKS: uses Auto Scaling Groups. New alternative: **Karpenter** (AWS-native):

- Provisions nodes in seconds (vs CA's minutes)
- Right-sizes node type for pending pods' requirements
- Automatically consolidates underutilized nodes
- No need to pre-configure node groups

---

### Q96. Explain Kubernetes resource quotas and LimitRange.

**ResourceQuota** — limits total resource consumption per namespace:

```
kind: ResourceQuota
spec:
  hard:
    requests.cpu: "20"
    requests.memory: 40Gi
    limits.cpu: "40"
    limits.memory: 80Gi
    pods: "100"
    services.loadbalancers: "2"
```

**LimitRange** — sets default and max/min for individual resources:

```
kind: LimitRange
spec:
  limits:
  - type: Container
    default:
      cpu: "500m"
      memory: "256Mi"
    defaultRequest:
      cpu: "100m"
      memory: "128Mi"
    max:
      cpu: "4"
      memory: "4Gi"
```

If a container has no resource requests, LimitRange injects the defaults (prevents BestEffort QoS).

---

### Q97. What is container runtime and what changed when Docker was deprecated in K8s 1.24?

The **Container Runtime Interface (CRI)** is the standard API between kubelet and container runtimes.

**Timeline:**
- Pre-1.20: Docker was supported via `dockershim` (a shim that translated CRI to Docker API)
- 1.20: dockershim deprecated
- 1.24: dockershim removed

**What this means:**
- Docker CLI and Dockerfiles still work for building images
- The RUNTIME on nodes is now containerd or CRI-O directly
- Images built with Docker are OCI-compliant → work with containerd
- Nothing changed for developers — only for node-level runtime configuration

**containerd** (most common): Docker's own runtime, extracted. Fast, battle-tested.
**CRI-O**: Lightweight runtime designed specifically for Kubernetes.

---

### Q98. How do you implement zero-trust security in Kubernetes?

Zero-trust = assume breach; verify every request; least privilege everywhere.

**Kubernetes zero-trust checklist:**
1. **Network:** Default-deny NetworkPolicy in every namespace; explicit allow rules only
2. **RBAC:** Principle of least privilege; no `cluster-admin` for workloads; audit bindings regularly
3. **Pod security:** Run as non-root, drop all capabilities, read-only root filesystem
4. **Secrets:** Encrypt etcd at rest; use Vault/Secrets Manager; short-lived dynamic secrets
5. **Image security:** Scan images (Trivy, Grype); sign images (Cosign); only pull from trusted registries (OPA policy)
6. **Service mesh (Istio/Linkerd):** mTLS between all pods; only whitelisted services can communicate
7. **Audit logging:** Enable K8s API audit logs; ship to SIEM; alert on sensitive operations
8. **Supply chain:** SBOM generation; admission webhook to block unsigned images

---

### Q99. What is the difference between `kubectl exec`, `kubectl debug`, and `kubectl cp`?

```
# exec — run a command in a running container
kubectl exec -it my-pod -- /bin/bash
kubectl exec my-pod -c sidecar -- ps aux   # specific container

# debug — create ephemeral debug container (K8s 1.23+)
kubectl debug -it my-pod --image=busybox --target=app
# Adds debug container to running pod WITHOUT restart
# Use when: production pod has no shell (distroless image)

# cp — copy files between pod and local
kubectl cp my-pod:/var/log/app.log ./app.log
kubectl cp ./config.yaml my-pod:/etc/config.yaml
```

`kubectl debug` is the modern replacement for mounting debug sidecars — it injects an ephemeral container into a running pod. The ephemeral container shares the process namespace with the target

container, so you can run `strace`, `lsof`, `tcpdump` against the app process without modifying the pod spec.

---

**Q100. Design a production-ready, highly available Kubernetes platform architecture.**

**Complete architecture:**

```
┌──── EXTERNAL ──────────────────────────────────────────┐
│ Route 53 (DNS) → CloudFront (CDN) → AWS WAF → NLB       │
└──── EKS CLUSTER ───────────────────────────────────────┘
│                                                          │
│ INGRESS LAYER:                                           │
│ AWS ALB Ingress Controller (per-namespace ALBs)          │
│ + NGINX Ingress (for advanced routing, rate limiting)    │
│                                                          │
│ SERVICE MESH:                                            │
│ Istio (mTLS everywhere, traffic management, observability)│
│                                                          │
│ NAMESPACE ISOLATION:                                     │
│ production / staging / monitoring / platform / security  │
│ ResourceQuota + LimitRange + NetworkPolicy per namespace │
│                                                          │
│ WORKLOADS:                                               │
│ Deployments + StatefulSets + DaemonSets                  │
│ HPA + KEDA for autoscaling                               │
│ PDB for availability guarantees                          │
│                                                          │
│ STORAGE:                                                 │
│ EBS (gp3) for StatefulSets                               │
│ EFS for RWX shared storage                               │
│                                                          │
│ NODE GROUPS:                                             │
│ System nodegroup (t3.medium, On-Demand, taints)          │
│ App nodegroup (m5.xlarge, On-Demand, min 3 / max 30)    │
│ Spot nodegroup (mixed instances, for non-critical batch) │
│ Karpenter for auto-provisioning                          │
│                                                          │
│ OBSERVABILITY:                                           │
│ Prometheus + Grafana + Alertmanager                      │
│ Loki (log aggregation) + Tempo (distributed tracing)    │
│ Kube-state-metrics + Node Exporter                      │
│                                                          │
│ SECURITY:                                                │
│ IRSA for all pod AWS access                              │
│ Falco (runtime security)                                 │
│ Kyverno (policy enforcement)                             │
│ cert-manager (TLS certificate lifecycle)                 │
│ AWS Secrets Manager + External Secrets Operator          │
│                                                          │
│ GITOPS:                                                  │
│ ArgoCD (applicationsets for multi-cluster)               │
│ Image Updater (auto-update on new image push)            │
│                                                          │
│ CONTROL PLANE:                                           │
│ EKS managed (AWS handles etcd HA, API server HA)         │
│ Multi-AZ node placement (3 AZs minimum)                 │
│ Private endpoint + VPN/bastion for API access            │
└──────────────────────────────────────────────────────────┘
```

**Key design decisions:**

- EKS managed control plane (no etcd/API server management overhead)
- Multi-AZ nodes with topologySpreadConstraints for all critical workloads
- Karpenter over Cluster Autoscaler (faster, smarter provisioning)
- IRSA for all AWS access (no static credentials anywhere)
- ArgoCD as single source of truth for cluster state
- External Secrets Operator pulls from Secrets Manager (secrets never in Git)
- Separate node groups with taints (system workloads isolated from app workloads)

# APPENDIX: Quick Reference — What Networking Knowledge is Must-Know

| Topic | Why It Matters in K8s |
|---|---|
| CIDR / Subnetting | Pod CIDR, service CIDR, node CIDR must not overlap |
| NAT (SNAT/DNAT) | How kube-proxy translates ClusterIPs to pod IPs |
| iptables / IPVS | How service routing is implemented at node level |
| DNS resolution | How pods resolve service names via CoreDNS |
| TCP connection states | Debugging connection issues, graceful shutdowns |
| L4 vs L7 load balancing | Choosing NLB vs ALB for ingress |
| BGP / VXLAN | How CNIs route cross-node traffic (Calico vs Flannel) |
| HTTPS / TLS | Ingress TLS termination, cert-manager, mTLS in service mesh |
| VPC / ENI (AWS) | How VPC CNI assigns pod IPs from ENIs |
| Firewall rules | NetworkPolicy = pod-level firewall |
| Port forwarding | How NodePort and HostPort work |
| eBPF | How Cilium bypasses iptables for high-performance networking |

*End of Guide — 100 Questions, Zero Filler.*
*Built for the engineer who ships to production, not one who reads documentation.*

# PART 3 — ANSIBLE (Questions 101–150)

> ***Same rules apply:*** *Direct answer → Mental model → Production-level depth. A 5-year engineer who has run Ansible in production at scale will pass after mastering every question here.*

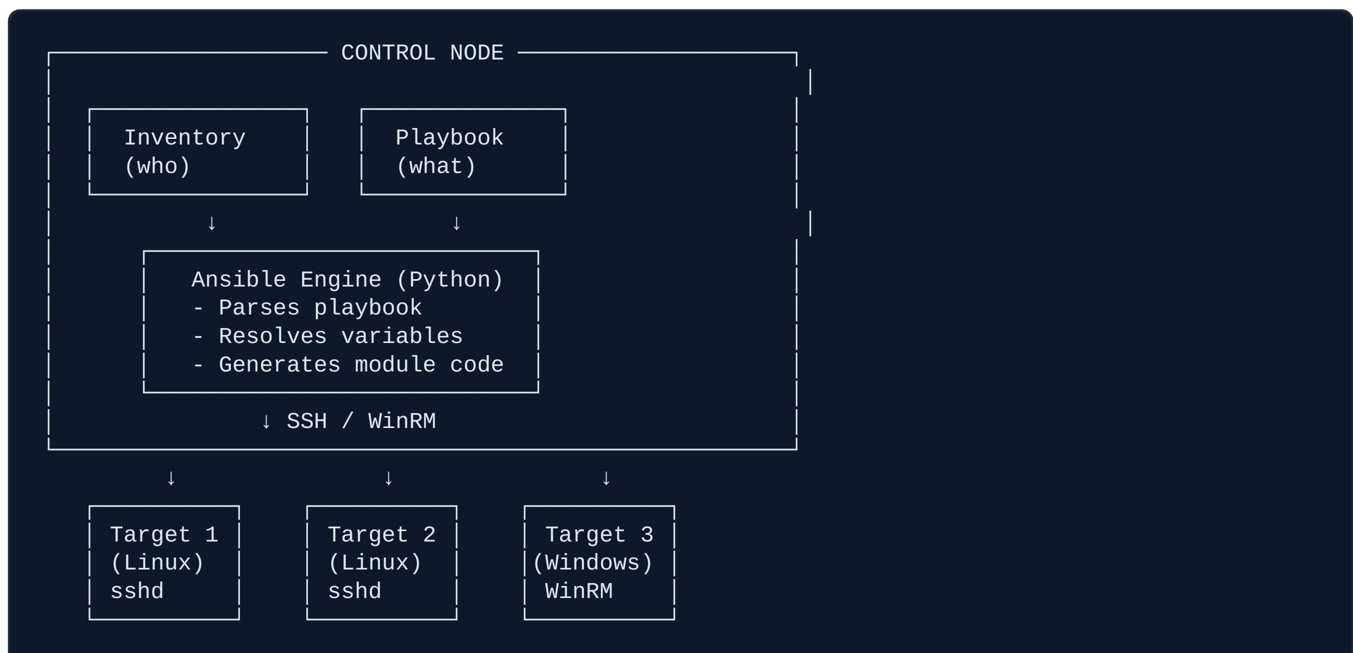
## SECTION A: Architecture & Core Concepts

### Q101. How does Ansible work? Explain its architecture and why it is agentless.

**Answer:**

Ansible is a **push-based, agentless automation tool**. It connects to target machines (over SSH for Linux, WinRM for Windows), pushes Python code (modules) as temporary scripts, executes them, collects the result, and deletes the scripts. No daemon, no agent, nothing permanent left on the target.

**Architecture:**




**Why agentless is an advantage:**

- No agent installation, patching, or version management on every server
- No inbound firewall ports needed on targets (only SSH port 22 outbound from control node)

- Works on any server with Python 2.7+/3.x and SSH — zero bootstrapping
- Immediate access to new machines the moment SSH is configured

**What runs on the target:**
Ansible assembles a Python script from the module code, copies it via SCP (or sftp/raw), executes it, reads JSON stdout, deletes it. The whole operation takes milliseconds per task.

**Limitation:** Because it's push-based and SSH-based, Ansible is slower than agent-based tools (Chef/Puppet) at scale. Every task = SSH connection overhead. Mitigated with pipelining and Mitogen.

---

### Q102. What is an Inventory? Explain static vs dynamic inventory.

**Answer:**
Inventory is Ansible's **list of target hosts** — the "who" you're automating against. Without inventory, Ansible doesn't know what to connect to.

**Static inventory (INI format):**

```
# inventory/hosts.ini
[webservers]
web1.prod.example.com
web2.prod.example.com ansible_user=ec2-user ansible_port=2222

[databases]
db1.prod.example.com ansible_host=10.0.1.50   # private IP alias

[prod:children]   # Group of groups
webservers
databases

[prod:vars]       # Variables for entire group
ansible_python_interpreter=/usr/bin/python3
env=production
```

**Static inventory (YAML format — preferred for readability):**

```
all:
  children:
    webservers:
      hosts:
        web1.prod.example.com:
          ansible_user: ec2-user
        web2.prod.example.com:
    databases:
      hosts:
        db1.prod.example.com:
          ansible_host: 10.0.1.50
    prod:
      children:
        webservers:
        databases:
      vars:
        env: production
```

**Dynamic inventory — connects to an external source:**
Instead of static files, Ansible queries an API (AWS, GCP, Azure, VMware, Vault) and builds the inventory dynamically.

**AWS EC2 plugin (most common):**

```yaml
# inventory/aws_ec2.yaml
plugin: amazon.aws.aws_ec2
regions:
  - us-east-1
  - eu-west-1
filters:
  tag:Environment: production
  instance-state-name: running
keyed_groups:
  - key: tags.Role
    prefix: role
  - key: placement.availability_zone
    prefix: az
hostnames:
  - tag:Name
  - private-ip-address
compose:
  ansible_host: private_ip_address
```

Run: `ansible-inventory -i inventory/aws_ec2.yaml --list` → shows JSON of all EC2 instances tagged `Environment=production`, grouped by their `Role` tag.

**Mental Model:**
- Static inventory = hardcoded contacts list in your phone
- Dynamic inventory = real-time sync with your company's LDAP/HR system — reflects actual live state

---

### Q103. Explain the anatomy of a Playbook — Play, Task, Module, Handler.

**Answer:**
A **Playbook** is a YAML file containing one or more **Plays**. Each Play targets hosts and runs **Tasks**. Each Task calls a **Module**.

```
Playbook (deploy-app.yml)
└── Play 1: "Configure web servers"        ← targets webservers group
    ├── Task 1: Install nginx               ← calls ansible.builtin.yum
    ├── Task 2: Deploy nginx config         ← calls ansible.builtin.template
    ├── Task 3: Start nginx                 ← calls ansible.builtin.service
    └── Handler: Restart nginx              ← triggered by notify, runs at end
└── Play 2: "Configure database"           ← targets databases group
    ├── Task 1: Install postgresql
    └── Task 2: Create database
```

**Full example:**

```yaml
---
- name: Configure and deploy web application      # Play name
  hosts: webservers                               # Target group from inventory
  become: true                                    # sudo for all tasks
  vars:
    app_version: "2.1.4"
    nginx_port: 80

  pre_tasks:
    - name: Update apt cache
      ansible.builtin.apt:
        update_cache: true
        cache_valid_time: 3600

  tasks:
    - name: Install nginx
      ansible.builtin.package:
        name: nginx
        state: present

    - name: Deploy nginx configuration
      ansible.builtin.template:
        src: templates/nginx.conf.j2
        dest: /etc/nginx/nginx.conf
        mode: '0644'
        owner: root
        group: root
      notify: Restart nginx                       # Signal handler

    - name: Ensure nginx is started and enabled
      ansible.builtin.service:
        name: nginx
        state: started
        enabled: true

  handlers:
    - name: Restart nginx                         # Runs ONCE at end if notified
      ansible.builtin.service:
        name: nginx
        state: restarted

  post_tasks:
    - name: Verify nginx is responding
      ansible.builtin.uri:
        url: "http://localhost:{{ nginx_port }}/health"
        status_code: 200
```

**Key behavioral rules:**

- Tasks run **top to bottom**, one at a time per host (or parallel across hosts with `forks`)

- **Handlers** only run if notified AND run **once** (not once per notify — deduplicated) at the end of the play

- `pre_tasks` → roles → `tasks` → `post_tasks` is the execution order

---

## Q104. What is idempotency in Ansible and how is it achieved?

**Answer:**

Idempotency means **running the same task multiple times produces the same result as running it once** — no unintended side effects on repeated runs.

**Mental Model:**
A light switch with an "ensure ON" function. Press it once → light turns on. Press it 10 more times → light stays on. Nothing explodes. That's idempotency.

**How Ansible modules achieve idempotency:**

```
# IDEMPOTENT — checks if package is installed first
- name: Install nginx
  ansible.builtin.package:
    name: nginx
    state: present    # "ensure installed", not "install right now"

# IDEMPOTENT — checks current state before acting
- name: Create user
  ansible.builtin.user:
    name: appuser
    state: present
    uid: 1050
    groups: ["sudo"]

# NOT IDEMPOTENT — runs command every time regardless
- name: Run script
  ansible.builtin.command: /opt/setup.sh   # Runs every time!
```

**Making `command` / `shell` idempotent:**

```
- name: Initialize database (idempotent)
  ansible.builtin.command: /opt/db-init.sh
  args:
    creates: /var/lib/db/.initialized    # Only runs if this file doesn't exist
  # OR:
  changed_when: false                    # Never reports changed (read-only commands)
  # OR:
  when: not db_initialized.stat.exists  # Conditional based on prior stat task
```

**The `changed` vs `ok` status distinction:**

- `ok` = task ran, state already correct, no change made
- `changed` = task ran, made a change
- `failed` = task ran, error occurred
- `skipped` = task's `when` condition was false

In a well-written playbook, a re-run should return mostly `ok` and zero `changed`. Lots of `changed` on a re-run = not idempotent.

---

### Q105. What is `ansible.cfg` and what are its most important settings?

**Answer:**

`ansible.cfg` is Ansible's **configuration file**. Ansible searches for it in this order (first found wins):

1. `ANSIBLE_CONFIG` environment variable
2. `./ansible.cfg` (current directory) ← **most common in projects**

3. `~/.ansible.cfg`
4. `/etc/ansible/ansible.cfg`

**Production `ansible.cfg`:**

```
[defaults]
inventory           = ./inventory
remote_user         = ec2-user
private_key_file    = ~/.ssh/prod-key.pem
host_key_checking   = False           # Disable SSH host key verification (careful in prod)
forks               = 20              # Parallel connections (default: 5)
timeout             = 30              # SSH connection timeout
retry_files_enabled = False           # Don't create .retry files on failure
gathering           = smart           # Cache facts between runs
fact_caching        = redis           # Store facts in Redis
fact_caching_connection = localhost:6379:0
fact_caching_timeout = 86400          # Cache facts for 24 hours
stdout_callback     = yaml            # Prettier output format
callbacks_enabled   = profile_tasks   # Show task timing at end

[privilege_escalation]
become              = True
become_method       = sudo
become_user         = root
become_ask_pass     = False

[ssh_connection]
pipelining          = True            # MAJOR PERFORMANCE BOOST — no temp file writes
control_path        = /tmp/ansible-%%r@%%h:%%p
control_master      = auto
control_persist     = 60s             # Reuse SSH connections for 60s
ssh_args            = -o ServerAliveInterval=60 -o ServerAliveCountMax=5
```

**The three settings that matter most for performance:**

1. `forks = 20+` — parallel hosts (default 5 is too low for real infrastructure)
2. `pipelining = True` — runs modules in memory instead of copying files (requires `requiretty` disabled in sudoers)
3. `gathering = smart` + fact caching — don't re-collect facts if already cached

---

## Q106. Explain `become` — how does privilege escalation work?

**Answer:**

`become` lets Ansible run tasks as a different user (typically root) after connecting as a non-root user.

**How it works under the hood:**

```
1. Ansible connects via SSH as ec2-user
2. Task needs become=true
3. Ansible prepends sudo -u root to command execution
   (or su, pbrun, pfexec, doas depending on become_method)
4. Task executes as root
5. Output returned as ec2-user connection
```

**Scope of become (most specific wins):**

```
---
- name: Deploy application
  hosts: webservers
  become: true              # Default: entire play runs as root

  tasks:
    - name: Install package (runs as root)
      ansible.builtin.package:
        name: nginx
        state: present

    - name: Create app directory (run as specific user)
      ansible.builtin.file:
        path: /opt/app
        state: directory
        owner: appuser
      become: true
      become_user: appuser  # This specific task runs as appuser

    - name: Check disk usage (no become needed)
      ansible.builtin.command: df -h
      become: false         # Override play-level become
```

`become_method` **options:**

| Method | Use case |
|---|---|
| `sudo` | Default. Most Linux systems. |
| `su` | When sudo not available |
| `pbrun` | Centrify / BeyondTrust PAM |
| `pfexec` | Solaris |
| `doas` | OpenBSD |
| `runas` | Windows |

`become_ask_pass`**:** Prompts for sudo password. In production automation, use passwordless sudo for the ansible user (restricted to specific commands via sudoers) instead.

---

## Q107. What is check mode (`--check`) and diff mode (`--diff`)?

**Answer:**

**Check mode (**`--check` **/** `-C`**)** — dry run:

```
ansible-playbook deploy.yml --check
```

Ansible runs through the playbook but **makes no changes**. Modules simulate what they would do and report `changed` or `ok` without acting. Like `terraform plan`.

**What check mode does NOT guarantee:**
- Tasks that depend on previous task results may report incorrectly (if task 1 would create a file, task 2's check that reads that file will fail — because the file doesn't exist yet in check mode)
- Command/shell modules don't run at all in check mode (unless `check_mode: false` is set)

**Diff mode (`--diff` / `-D`)** — shows what changed:

```
ansible-playbook deploy.yml --diff
ansible-playbook deploy.yml --check --diff  # Most useful combination
```

For file/template/copy tasks, diff mode shows the before/after file content (like `git diff`).

**Example output:**

```
TASK [Deploy nginx.conf]
--- before: /etc/nginx/nginx.conf
+++ after: /home/user/.ansible/tmp/nginx.conf.j2
@@ -12,7 +12,7 @@
     server {
-        listen 80;
+        listen 8080;
         server_name example.com;
```

**Per-task check mode control:**

```
- name: Always run this even in check mode (e.g., gathering info)
  ansible.builtin.command: /opt/health-check.sh
  check_mode: false

- name: Skip this task in normal runs, only in check mode
  ansible.builtin.debug:
    msg: "This would change: {{ item }}"
  when: ansible_check_mode
```

## Q108. What are ad-hoc commands and when do you use them?

**Answer:**
Ad-hoc commands run a **single module directly from the command line** without a playbook — for one-off tasks, quick checks, or emergencies.

```
# Syntax
ansible <pattern> -m <module> -a "<module_args>" [options]

# Ping all hosts (test connectivity)
ansible all -m ansible.builtin.ping

# Run a shell command on all webservers
ansible webservers -m ansible.builtin.shell -a "uptime"

# Copy a file
ansible db1.prod.example.com -m ansible.builtin.copy \
  -a "src=/local/file dest=/remote/file mode=0644"

# Install a package (with become)
ansible webservers -m ansible.builtin.package \
  -a "name=htop state=present" --become

# Gather facts
ansible web1 -m ansible.builtin.setup

# Restart a service
ansible webservers -m ansible.builtin.service \
  -a "name=nginx state=restarted" --become

# Reboot all hosts and wait for them to come back
ansible all -m ansible.builtin.reboot --become

# Check free memory on all hosts
ansible all -m ansible.builtin.shell -a "free -h"

# Run as different user
ansible webservers -u ec2-user --become --become-user=root \
  -m ansible.builtin.shell -a "whoami"
```

**When to use ad-hoc:**

- Emergency: "Quick, restart nginx on all prod web servers NOW"
- Fact gathering: "What kernel version are all my boxes running?"
- One-off config: "Add a temp SSH key for a contractor"
- Testing: "Can Ansible reach all my new instances?"

**When NOT to use ad-hoc:**

- Anything repeatable → write a playbook (tracked in git, reviewable, reusable)
- Anything multi-step → playbook with tasks
- Production changes → playbook with peer review

---

## Q109. How does Ansible execute tasks — serial, parallel, and connection management?

**Answer:**

By default, Ansible runs **each task across ALL hosts in parallel** (up to `forks` count), then moves to the next task.

**Default linear strategy (task-by-task across all hosts):**

```
Task 1: Run on [web1, web2, web3] simultaneously (up to forks limit)
  → All complete
Task 2: Run on [web1, web2, web3] simultaneously
  → All complete
Task 3: ...
```

`serial` **— rolling updates (batch processing):**

```
- name: Rolling deployment
  hosts: webservers   # 10 servers
  serial: 2           # Process 2 at a time (rolling update)
  # serial: "20%"     # 20% of hosts at a time
  # serial: [1, 3, 5] # Canary: 1 first, then 3, then 5 at a time

  tasks:
    - name: Drain from load balancer
      # ...
    - name: Deploy new app version
      # ...
    - name: Add back to load balancer
      # ...
```

This creates a true **rolling deployment pattern** — 2 servers updated, then next 2, never all at once.

`free` **strategy (as fast as possible):**

```
- name: Independent tasks
  hosts: all
  strategy: free    # Each host moves to next task as soon as it's done
                    # Don't wait for all hosts to finish a task
```

`max_fail_percentage` **— abort threshold:**

```
- hosts: webservers
  max_fail_percentage: 20   # Abort if >20% of hosts fail
  any_errors_fatal: false   # (vs this which stops on ANY failure)
```

**SSH connection multiplexing (ControlMaster):**

With `control_master = auto` and `control_persist = 60s` in ansible.cfg, Ansible reuses the same SSH connection for all tasks on a host instead of creating a new SSH connection per task. Massive performance improvement.

---

## Q110. What is the difference between `command`, `shell`, `raw`, and `script` modules?

**Answer:**

All four run commands on remote hosts but with important behavioral differences:

| Module | Uses shell | Variable expansion | Pipelines/redirects | Requires Python |
| --- | --- | --- | --- | --- |
| `command` | No | No | No | Yes |
| `shell` | Yes | Yes | Yes | Yes |
| `raw` | No (direct SSH) | No | No | No |
| `script` | No | No | No | Yes (for args) |

`command` — safest, most restrictive:

```
- ansible.builtin.command: /opt/start-app.sh
# Cannot use: pipes, redirects, shell variables, &&, ||
# ✓ Use for: simple executables with fixed arguments
```

`shell` — full shell features:

```
- ansible.builtin.shell: "ps aux | grep nginx | grep -v grep | wc -l"
# ✓ Use for: pipes, redirects, complex shell logic
# ✗ Avoid: security risk if any variable is user-controlled (shell injection)
```

`raw` — bypasses Python entirely:

```
- ansible.builtin.raw: "apt-get install -y python3"
# ✓ Use for: bootstrapping Python on a system that doesn't have it yet
# ✗ Never use for regular tasks — no idempotency, no change tracking
```

`script` — copies local script to remote and runs it:

```
- ansible.builtin.script: files/setup.sh arg1 arg2
# ✓ Use for: complex setup scripts you maintain locally
# Copies script → executes → cleans up
```

**Production rule:** Prefer `command` > `shell` > `script` > `raw`. Use `shell` only when you need its features. When you use `shell`, always set `changed_when` and `failed_when` — shell always reports `changed=true` unless told otherwise.

## SECTION B: Variables, Facts & Templating

## Q111. Explain Ansible's variable precedence — the 22 levels.

**Answer:**
Variable precedence determines **which value wins when the same variable is defined in multiple places**. Lower number = lower precedence (gets overridden). Higher number = wins.

**Condensed, production-relevant precedence (low → high):**

```
1.  Role defaults           (role/defaults/main.yml)    ← Lowest
2.  Inventory group_vars/all
3.  Inventory group_vars/<groupname>
4.  Inventory host_vars/<hostname>
5.  Playbook group_vars/all
6.  Playbook group_vars/<groupname>
7.  Playbook host_vars/<hostname>
8.  Host facts / cached facts
9.  Play vars                (vars: key: value in play)
10. Play vars_files
11. Role vars               (role/vars/main.yml)
12. Block vars
13. Task vars
14. set_fact / registered vars
15. Extra vars (-e flag)     ← Highest (ALWAYS wins)
```

**The critical rules to remember:**
- `role/defaults/main.yml` → meant to be overridden. Set safe defaults here.
- `role/vars/main.yml` → internal role variables, NOT meant to be overridden easily.
- `-e` **extra vars** → always win. Used in CI/CD to inject environment-specific values.
- `set_fact` → takes precedence over most other sources, persists for the rest of the play.

**Common mistake — vars vs defaults:**

```
# In role/defaults/main.yml (overridable)
nginx_port: 80

# In role/vars/main.yml (hard to override — high precedence)
nginx_user: www-data   # Internal implementation detail
```

If a user passes `-e nginx_port=8080`, it overrides the default. If you put `nginx_port` in `vars/main.yml`, the `-e` flag still wins, but inventory and host_vars won't be able to override it.

---

## Q112. What are Ansible Facts and how do you use them?

**Answer:**
Facts are **automatically gathered information about target hosts** — OS, IP addresses, memory, CPU, disk, network interfaces, kernel version, and more. Gathered via the `setup` module at play start.

**Accessing facts:**

```
- name: Show OS information
  ansible.builtin.debug:
    msg: "Running {{ ansible_distribution }} {{ ansible_distribution_version }}
          on {{ ansible_architecture }}"

# Common facts:
# ansible_hostname            → "web1"
# ansible_fqdn                → "web1.prod.example.com"
# ansible_os_family           → "Debian" / "RedHat"
# ansible_distribution        → "Ubuntu" / "CentOS"
# ansible_distribution_version → "22.04"
# ansible_architecture        → "x86_64"
# ansible_memtotal_mb         → 8192
# ansible_processor_vcpus     → 4
# ansible_default_ipv4.address → "10.0.1.5"
# ansible_all_ipv4_addresses → ["10.0.1.5", "172.17.0.1"]
# ansible_interfaces          → ["eth0", "lo", "docker0"]
# ansible_kernel              → "5.15.0-1034-aws"
# ansible_uptime_seconds      → 86400
```

**Fact-based conditionals:**

```
- name: Install Apache (handle different OS families)
  ansible.builtin.package:
    name: "{{ 'apache2' if ansible_os_family == 'Debian' else 'httpd' }}"
    state: present

- name: Only run on systems with enough RAM
  when: ansible_memtotal_mb >= 4096
  ansible.builtin.debug:
    msg: "This server has enough RAM for the heavy workload"
```

**Custom facts (local facts):**

```
# Drop a .fact file on the target (JSON or INI format)
# /etc/ansible/facts.d/app.fact
{
  "version": "2.1.4",
  "deploy_user": "appuser",
  "last_deployed": "2024-01-15"
}
```

```
# Access via ansible_local
- debug:
    msg: "App version: {{ ansible_local.app.version }}"
```

**Performance — disable fact gathering when not needed:**

```
- hosts: webservers
  gather_facts: false    # Skip fact gathering — saves 1-3s per host
  tasks:
    - name: Quick restart task
      ansible.builtin.service:
        name: nginx
        state: restarted
```

## Q113. What are registered variables and how do you use them?

**Answer:**

`register` captures a **task's output** into a variable for use in subsequent tasks.

```
- name: Check if app config exists
  ansible.builtin.stat:
    path: /etc/myapp/config.yaml
  register: app_config_stat

- name: Deploy default config if missing
  ansible.builtin.template:
    src: config.yaml.j2
    dest: /etc/myapp/config.yaml
  when: not app_config_stat.stat.exists

- name: Get current running processes
  ansible.builtin.shell: ps aux
  register: running_processes
  changed_when: false    # This is a read-only check

- name: Show process count
  ansible.builtin.debug:
    msg: "Running {{ running_processes.stdout_lines | length }} processes"

- name: Install app binary
  ansible.builtin.get_url:
    url: https://releases.example.com/app-{{ app_version }}.tar.gz
    dest: /tmp/app.tar.gz
  register: download_result

- name: Show download result
  ansible.builtin.debug:
    var: download_result    # Dumps entire registered object
```

**Registered variable structure (for shell/command):**

```
result:
  stdout: "hello world"
  stderr: ""
  stdout_lines: ["hello world"]
  stderr_lines: []
  rc: 0                    # Return code
  changed: true
  failed: false
  cmd: ["echo", "hello world"]
```

**Using `register` with loops:**

```
- name: Check multiple services
  ansible.builtin.service_facts:
  register: service_state

- name: Show nginx status
  ansible.builtin.debug:
    msg: "nginx is {{ service_state.ansible_facts.services['nginx.service'].state }}"
```

## Q114. Explain Jinja2 templating in Ansible — syntax and common filters.

**Answer:**

Ansible uses **Jinja2** for all variable interpolation and templating. Jinja2 expressions appear in `{{ }}` (variables), `{% %}` (control flow), and `{# #}` (comments).

**Variable substitution:**

```
vars:
  app_name: myapi
  version: "2.1"

tasks:
  - name: "Deploy {{ app_name }} version {{ version }}"
    # Task name supports Jinja2
```

**Jinja2 in template files (`.j2`):**

```
# templates/nginx.conf.j2
user {{ nginx_user | default('www-data') }};
worker_processes {{ ansible_processor_vcpus }};

http {
    upstream backend {
{% for host in groups['appservers'] %}
        server {{ hostvars[host]['ansible_default_ipv4']['address'] }}:8080;
{% endfor %}
    }

    server {
        listen {{ nginx_port }};
        server_name {{ inventory_hostname }};

{% if enable_ssl %}
        ssl_certificate {{ ssl_cert_path }};
{% endif %}
    }
}
```

**Essential filters:**

```
# String manipulation
"{{ my_var | upper }}"               # → "HELLO"
"{{ my_var | lower }}"               # → "hello"
"{{ my_var | replace('a', 'b') }}"   # → replace chars
"{{ my_var | trim }}"                # Strip whitespace
"{{ my_var | default('fallback') }}" # Use fallback if undefined

# List/dict operations
"{{ my_list | length }}"             # Count items
"{{ my_list | first }}"              # First item
"{{ my_list | last }}"              # Last item
"{{ my_list | join(',') }}"          # Join with separator
"{{ my_list | sort }}"              # Sort list
"{{ my_list | unique }}"            # Remove duplicates
"{{ my_list | select('match', 'web.*') | list }}"  # Filter by regex

# Type conversion
"{{ '42' | int }}"                   # String to int
"{{ 42 | string }}"                  # Int to string
"{{ my_var | bool }}"                # To boolean
"{{ my_dict | to_json }}"            # Dict to JSON string
"{{ my_json_string | from_json }}"   # JSON string to dict

# Path manipulation
"{{ '/etc/nginx/nginx.conf' | dirname }}"   # /etc/nginx
"{{ '/etc/nginx/nginx.conf' | basename }}"  # nginx.conf

# Crypto/encoding
"{{ 'password' | password_hash('sha512') }}"  # Hash password
"{{ data | b64encode }}"                       # Base64 encode
"{{ data | b64decode }}"                       # Base64 decode

# Conditional
"{{ value | ternary('yes_value', 'no_value') }}"
```

## Q115. What is Ansible Vault and how do you use it?

**Answer:**

Ansible Vault **encrypts sensitive data** (secrets, passwords, API keys) using AES-256 so they can be safely committed to version control.

**Creating and using encrypted files:**

```
# Create new encrypted file
ansible-vault create secrets.yml

# Encrypt existing file
ansible-vault encrypt vars/secrets.yml

# View encrypted file
ansible-vault view vars/secrets.yml

# Edit encrypted file (opens in $EDITOR)
ansible-vault edit vars/secrets.yml

# Decrypt file (in-place, careful!)
ansible-vault decrypt vars/secrets.yml

# Change vault password
ansible-vault rekey vars/secrets.yml
```

**Encrypted `secrets.yml`:**

```
$ANSIBLE_VAULT;1.1;AES256
66386439653236336462626566653063336164663538633963613562353538...
```

**Inline encrypted variables (`!vault` tag):**

```
# group_vars/prod/secrets.yml — mix encrypted and plain vars
db_user: appuser                    # Plain (not sensitive)
db_password: !vault |               # Encrypted inline
  $ANSIBLE_VAULT;1.1;AES256
  66386439653236336462626566...
```

**Running with vault password:**

```
# Prompt for password
ansible-playbook deploy.yml --ask-vault-pass

# Use password file (for CI/CD)
ansible-playbook deploy.yml --vault-password-file ~/.vault_pass

# Use environment variable (most CI-friendly)
export ANSIBLE_VAULT_PASSWORD_FILE=~/.vault_pass
ansible-playbook deploy.yml
```

**Multiple vault IDs (prod vs dev secrets with different passwords):**

```
ansible-vault encrypt_string 'prod-secret' --vault-id prod@prompt
ansible-playbook deploy.yml \
  --vault-id prod@~/.vault_pass_prod \
  --vault-id dev@~/.vault_pass_dev
```

**Production integration:** In CI/CD (GitHub Actions, Jenkins), store vault password as a CI secret, write it to a temp file, pass `--vault-password-file`. Never print the vault password in logs.

## Q116. What are lookups in Ansible?

**Answer:**
Lookups let Ansible **pull data from external sources on the control node** — files, environment variables, databases, AWS SSM, HashiCorp Vault, etc.

```
vars:
  # Read a file from control node
  ssl_cert: "{{ lookup('file', '/etc/ssl/certs/server.crt') }}"

  # Read environment variable from control node
  aws_region: "{{ lookup('env', 'AWS_DEFAULT_REGION') }}"

  # Read from AWS SSM Parameter Store
  db_password: "{{ lookup('amazon.aws.aws_ssm',
                 '/prod/db/password', region='us-east-1') }}"

  # Read from HashiCorp Vault
  api_key: "{{ lookup('community.hashi_vault.hashi_vault',
             'secret=secret/myapp/api_key:value') }}"

  # Generate a password (or retrieve if already created)
  generated_pass: "{{ lookup('password', '/tmp/pass length=20 chars=ascii') }}"

  # Read entire file as lines list
  hosts_list: "{{ lookup('file', 'hosts.txt').splitlines() }}"

  # Query a URL (HTTP GET)
  latest_version: "{{ lookup('url', 'https://api.github.com/repos/org/repo/releases/latest')
| from_json | json_query('tag_name') }}"
```

**Key distinction — lookups run on the CONTROL NODE:**

```
# This reads /etc/hosts from YOUR MACHINE (control node), not the target
- debug:
    msg: "{{ lookup('file', '/etc/hosts') }}"

# To read from target, use the slurp module
- ansible.builtin.slurp:
    src: /etc/hosts
  register: remote_hosts
- debug:
    msg: "{{ remote_hosts.content | b64decode }}"
```

---

## Q117. What are magic variables? Explain `hostvars`, `groups`, `inventory_hostname`.

**Answer:**
Magic variables are **special variables automatically set by Ansible** — not gathered from hosts, not defined in inventory. They expose metadata about the play and inventory.

**Critical magic variables:**

`inventory_hostname` — the hostname as defined in inventory:

```
- name: Create hostname-specific config
  ansible.builtin.template:
    src: config.j2
    dest: "/etc/app/{{ inventory_hostname }}.conf"
```

`ansible_hostname` — actual hostname reported by the machine (may differ from inventory name)

`groups` — dictionary of ALL inventory groups and their hosts:

```
- name: Configure all app servers in nginx upstream
  ansible.builtin.template:
    src: nginx_upstream.j2
    dest: /etc/nginx/conf.d/upstream.conf
  vars:
    app_servers: "{{ groups['appservers'] }}"
```

In template:

```
upstream backend {
{% for host in groups['appservers'] %}
    server {{ hostvars[host].ansible_default_ipv4.address }}:8080;
{% endfor %}
}
```

`hostvars` — dictionary of ALL hosts' variables and facts:

```
# Access another host's facts from current host's task
- name: Set primary DB server IP
  ansible.builtin.template:
    src: app.conf.j2
  vars:
    db_ip: "{{ hostvars['db-primary.prod.example.com']['ansible_default_ipv4']['address'] }}"
```

**Other useful magic variables:**

```
inventory_hostname_short  # "web1" (without domain)
group_names               # List of groups this host belongs to: ["webservers", "prod"]
ansible_play_hosts        # List of hosts in current play (after limit/failed)
ansible_play_batch        # Current batch of hosts (when using serial)
ansible_role_name         # Name of currently executing role
playbook_dir              # Directory of the currently running playbook
```

## SECTION C: Roles & Code Organization

## Q118. Explain Ansible Roles — structure, purpose, and best practices.

**Answer:**

A Role is a **standardized directory structure for reusable, shareable automation** — the equivalent of a Terraform module or a software library.

**Role directory structure:**

```
roles/
└── nginx/
    ├── defaults/
    │   └── main.yml       # Default variables (lowest precedence, meant to be overridden)
    ├── vars/
    │   └── main.yml       # Role-internal variables (higher precedence, NOT for override)
    ├── tasks/
    │   ├── main.yml       # Entry point — includes other task files
    │   ├── install.yml    # Package installation tasks
    │   └── configure.yml  # Configuration tasks
    ├── handlers/
    │   └── main.yml       # Handlers (e.g., restart nginx)
    ├── templates/
    │   └── nginx.conf.j2  # Jinja2 templates
    ├── files/
    │   └── mime.types     # Static files to copy
    ├── meta/
    │   └── main.yml       # Role metadata + dependencies
    └── README.md
```

**Calling a role:**

```
# Playbook
- hosts: webservers
  roles:
    - nginx                          # Simplest form
    - role: nginx                    # With params
      vars:
        nginx_port: 8080
    - role: common
      tags: [common, baseline]
```

`meta/main.yml` **— role dependencies:**

```
dependencies:
  - role: common
    vars:
      ntp_server: pool.ntp.org
  - role: firewall
    vars:
      open_ports: [80, 443]
```

Dependencies run BEFORE the current role, automatically.

`defaults/main.yml` **vs** `vars/main.yml`**:**

```
# defaults/main.yml — safe defaults, override freely
nginx_port: 80
nginx_worker_processes: auto

# vars/main.yml — internal constants, don't override
_nginx_config_dir: /etc/nginx
_nginx_log_dir: /var/log/nginx
```

## Q119. What are Ansible Collections and how do they differ from roles?

**Answer:**

Collections are **namespaced packages of related content** — roles, modules, plugins, playbooks, documentation — bundled together and distributed via Ansible Galaxy or private Automation Hub.

**Mental Model:**

- Role = a single reusable recipe (e.g., "install nginx")
- Collection = a cookbook (e.g., "everything you need for AWS: 300 modules + 50 roles + plugins")

**Collection namespace format:** `namespace.collection_name`

```
amazon.aws          # AWS modules (ec2, s3, rds, iam...)
community.docker    # Docker modules
kubernetes.core     # K8s modules (k8s, helm)
ansible.posix       # POSIX utilities (sysctl, mount, firewalld)
community.general   # General-purpose (many miscellaneous modules)
```

**Installing collections:**

```
# From Ansible Galaxy
ansible-galaxy collection install amazon.aws
ansible-galaxy collection install -r requirements.yml

# requirements.yml
collections:
  - name: amazon.aws
    version: ">=6.0.0"
  - name: community.docker
    version: "3.4.0"
  - name: https://my-hub.internal/api/galaxy/content/validated/
    type: url
```

**Using collection modules:**

```
- name: Create EC2 instance (collection module)
  amazon.aws.ec2_instance:
    name: my-web-server
    instance_type: t3.medium
    image_id: ami-0abc123
    vpc_subnet_id: "{{ subnet_id }}"
    security_groups: ["web-sg"]
    tags:
      Environment: production
```

**FQCN (Fully Qualified Collection Name)** — always use full names in production:

```
# Not this (ambiguous — which 'copy' module?)
- copy:

# This (explicit, future-proof)
- ansible.builtin.copy:
```

---

### Q120. What is the difference between `import_tasks`, `include_tasks`, `import_role`, and `include_role`?

**Answer:**

This is a senior-level nuance that trips up many candidates.

**Import (static) — processed at parse time:**

```
- import_tasks: tasks/install.yml     # Tasks are merged into playbook before run
- import_role: name=nginx             # Role is merged at parse time
```

- Tags applied to `import_tasks` apply to ALL tasks inside the imported file
- Can use task-level `when` outside the import (applies to all imported tasks)
- **Cannot** be used inside loops
- The file must exist at parse time

**Include (dynamic) — processed at runtime:**

```
- include_tasks: "tasks/{{ ansible_os_family }}.yml"   # Can use variables!
- include_role: name="{{ app_role }}"                   # Dynamic role name
```

- Tags must be applied inside the included file (outer tags don't pass in)
- **Can** be used inside loops (`with_items`, `loop`)
- The file is read at the moment that task runs
- Can conditionally include based on variables that change during run

**Choosing which to use:**

```
# Use import when: structure is fixed, you want tags to pass through
- import_tasks: common/setup.yml

# Use include when: filename is dynamic, inside a loop, conditional at runtime
- include_tasks: "{{ ansible_os_family }}-packages.yml"

- name: Deploy multiple apps
  include_role:
    name: deploy_app
  vars:
    app_name: "{{ item }}"
  loop: [frontend, backend, worker]   # include_role works in loops, import_role doesn't
```

---

## Q121. What are Handlers? What are the edge cases around handler execution?

**Answer:**

Handlers are tasks that **run once at the end of a play when notified** — typically used to restart services after config changes.

```
tasks:
  - name: Update nginx config
    ansible.builtin.template:
      src: nginx.conf.j2
      dest: /etc/nginx/nginx.conf
    notify:
      - Reload nginx
      - Clear nginx cache

  - name: Update SSL certificate
    ansible.builtin.copy:
      src: server.crt
      dest: /etc/ssl/server.crt
    notify: Reload nginx     # Same handler notified twice — still runs ONCE

handlers:
  - name: Reload nginx
    ansible.builtin.service:
      name: nginx
      state: reloaded

  - name: Clear nginx cache
    ansible.builtin.file:
      path: /var/cache/nginx
      state: absent
```

**Edge cases and important behaviors:**

**1. Handlers run at end of play, not immediately:**

```
tasks:
  - name: Deploy config (notifies handler)
    template: ...
    notify: Restart app

  - name: This runs BEFORE the handler
    debug: msg="App is still using OLD config here"

# Handler runs HERE (end of play)
```

**2. `flush_handlers` — force handlers to run immediately:**

```
- name: Update database config
  template: ...
  notify: Restart database

- name: Flush handlers NOW (before next task)
  ansible.builtin.meta: flush_handlers

- name: Run DB migration (needs restarted DB)
  command: python manage.py migrate
```

**3. Handler chaining (handlers can notify other handlers):**

```
handlers:
  - name: Restart nginx
    service: name=nginx state=restarted
    notify: Verify nginx      # Notify another handler

  - name: Verify nginx
    uri:
      url: http://localhost/health
      status_code: 200
```

**4. If a task fails, handlers from that play DON'T run** (unless `--force-handlers` flag is used).

---

## Q122. What are Tags in Ansible and how do you use them effectively?

**Answer:**

Tags let you **selectively run or skip specific tasks** in a playbook without commenting out code.

**Applying tags:**

```yaml
tasks:
  - name: Install packages
    ansible.builtin.package:
      name: nginx
      state: present
    tags:
      - install
      - packages

  - name: Deploy configuration
    ansible.builtin.template:
      src: nginx.conf.j2
      dest: /etc/nginx/nginx.conf
    tags:
      - configure
      - nginx

  - name: Restart service
    ansible.builtin.service:
      name: nginx
      state: restarted
    tags:
      - restart
      - never          # Special tag: ONLY runs when explicitly called
```

**Running with tags:**

```
ansible-playbook deploy.yml --tags install         # Only install tasks
ansible-playbook deploy.yml --tags "install,configure"
ansible-playbook deploy.yml --skip-tags restart    # All except restart
ansible-playbook deploy.yml --tags never           # Only "never"-tagged tasks
ansible-playbook deploy.yml --list-tags            # Show all available tags
ansible-playbook deploy.yml --list-tasks --tags configure  # Preview what runs
```

**Special built-in tags:**

- `always` — task runs regardless of `--tags` filter (use for fact gathering)
- `never` — task only runs when explicitly called with `--tags never`

**Tag inheritance:**

```yaml
- name: Nginx setup play
  hosts: webservers
  tags: nginx          # Tags entire play — ALL tasks inherit this

  tasks:
    - name: Install nginx    # Gets both "nginx" and "install" tags
      tags: install
      ...
```

**Production patterns:**

```
# Deploy only config changes (common during config tuning)
ansible-playbook deploy.yml --tags configure

# Quick restart without full deploy
ansible-playbook deploy.yml --tags restart

# Run smoke test tasks (tagged "smoke")
ansible-playbook full-deploy.yml --tags smoke
```

## Q123. How do you test Ansible roles with Molecule?

**Answer:**

Molecule is the **standard testing framework for Ansible roles** — it creates ephemeral test instances, runs your role, verifies the outcome, and destroys them.

**Molecule structure:**

```
roles/nginx/
└── molecule/
    └── default/
        ├── molecule.yml     # Configuration (driver, platforms)
        ├── converge.yml     # Playbook that applies your role
        ├── verify.yml       # Assertions (did it actually work?)
        └── prepare.yml      # Optional: pre-role setup
```

`molecule.yml`:

```
driver:
  name: docker             # or vagrant, ec2, delegated

platforms:
  - name: ubuntu-22
    image: geerlingguy/docker-ubuntu2204-ansible
    pre_build_image: true
  - name: centos-9
    image: geerlingguy/docker-centos9-ansible
    pre_build_image: true

provisioner:
  name: ansible
  config_options:
    defaults:
      callbacks_enabled: profile_tasks

verifier:
  name: ansible            # Use Ansible tasks for verification (or testinfra for Python)
```

`verify.yml`:

```
- name: Verify nginx installation
  hosts: all
  tasks:
    - name: Check nginx is installed
      ansible.builtin.package:
        name: nginx
        state: present
      check_mode: true
      register: result
      failed_when: result.changed     # If it reports "would install" → not installed!

    - name: Check nginx is running
      ansible.builtin.service_facts:

    - name: Assert nginx is active
      ansible.builtin.assert:
        that:
          - ansible_facts.services['nginx.service'].state == 'running'
          - ansible_facts.services['nginx.service'].status == 'enabled'

    - name: Verify nginx responds on port 80
      ansible.builtin.uri:
        url: http://localhost
        status_code: 200
```

**Molecule commands:**

```
molecule create       # Create test instances
molecule converge     # Run role on instances
molecule verify       # Run verifications
molecule destroy      # Destroy instances
molecule test         # Full pipeline: create → converge → verify → destroy
molecule lint         # Lint role code
```

## SECTION D: Control Flow & Advanced Features

### Q124. Explain `block`, `rescue`, and `always` — Ansible's exception handling.

**Answer:**

`block/rescue/always` is Ansible's equivalent of `try/except/finally` in programming.

```
tasks:
  - name: Deploy application with error handling
    block:                             # TRY block
      - name: Pull Docker image
        community.docker.docker_image:
          name: myapp:{{ version }}
          source: pull

      - name: Stop old container
        community.docker.docker_container:
          name: myapp
          state: stopped

      - name: Start new container
        community.docker.docker_container:
          name: myapp
          image: myapp:{{ version }}
          state: started

    rescue:                            # EXCEPT block — runs if ANY block task fails
      - name: Rollback to previous version
        community.docker.docker_container:
          name: myapp
          image: myapp:{{ previous_version }}
          state: started

      - name: Alert on-call team
        ansible.builtin.uri:
          url: "{{ pagerduty_webhook }}"
          method: POST
          body_format: json
          body:
            message: "Deployment of {{ version }} FAILED. Rolled back."

      - name: Mark task as failed after rescue
        ansible.builtin.fail:
          msg: "Deployment failed — rolled back to {{ previous_version }}"

    always:                            # FINALLY block — ALWAYS runs (success or failure)
      - name: Write deployment log
        ansible.builtin.lineinfile:
          path: /var/log/deployments.log
          line: "{{ '%Y-%m-%d %H:%M' | strftime }}: Deployed {{ version }} — {{ 'success' if
not ansible_failed_task is defined else 'failed' }}"
          create: true

      - name: Remove temp files
        ansible.builtin.file:
          path: /tmp/deploy-workspace
          state: absent
```

**`vars` scope in blocks:**

Variables defined in `block` are accessible in `rescue` and `always`. The `ansible_failed_task` and `ansible_failed_result` magic variables are available in `rescue`.

## Q125. How do loops work in Ansible?

**Answer:**

Loops run a task multiple times with different items.

`loop` **(modern, preferred):**

```
- name: Create multiple users
  ansible.builtin.user:
    name: "{{ item }}"
    state: present
    shell: /bin/bash
  loop:
    - alice
    - bob
    - carol

- name: Install multiple packages
  ansible.builtin.package:
    name: "{{ item.name }}"
    state: "{{ item.state }}"
  loop:
    - { name: nginx, state: present }
    - { name: apache2, state: absent }
    - { name: curl, state: present }

- name: Create users with full details
  ansible.builtin.user:
    name: "{{ item.name }}"
    uid: "{{ item.uid }}"
    groups: "{{ item.groups }}"
  loop: "{{ users }}"           # Loop over a variable
  loop_control:
    label: "{{ item.name }}"    # Show only name in output (not full item dict)
    pause: 2                    # Wait 2s between iterations
    index_var: idx              # Access current index as 'idx'
```

`loop` **with** `register`**:**

```
- name: Check multiple URLs
  ansible.builtin.uri:
    url: "{{ item }}"
    status_code: 200
  register: url_results
  loop:
    - http://web1/health
    - http://web2/health
    - http://web3/health

- name: Show results
  ansible.builtin.debug:
    msg: "{{ item.url }} returned {{ item.status }}"
  loop: "{{ url_results.results }}"
```

**Legacy** `with_*` **loops (still works but deprecated style):**

```
with_items: [a, b, c]          # Same as loop
with_dict: "{{ my_dict }}"     # Loop over dict → item.key, item.value
with_fileglob: "files/*.conf"  # Loop over matching files
with_sequence: start=1 end=10  # Generate sequence
with_nested:                   # Nested loop (cartesian product)
  - [a, b]
  - [1, 2]
# → (a,1), (a,2), (b,1), (b,2)
```

---

### Q126. Explain `when` conditionals — syntax and common patterns.

**Answer:**

`when` controls whether a task runs. It evaluates a Jinja2 expression (without `{{ }}`) that must be truthy for the task to execute.

```yaml
# Simple variable check
- name: Install nginx on Debian
  ansible.builtin.apt:
    name: nginx
  when: ansible_os_family == "Debian"

# Multiple conditions (AND)
- name: Run only on prod Ubuntu
  when:
    - env == "production"
    - ansible_distribution == "Ubuntu"
    - ansible_distribution_major_version | int >= 20

# OR condition
- name: Run on Debian or Ubuntu
  when: ansible_os_family == "Debian" or ansible_distribution == "Ubuntu"

# Check if variable is defined
- when: db_password is defined

# Check if variable is not empty
- when: db_password is defined and db_password | length > 0

# Check registered result
- name: Run migration only if db not initialized
  when: not db_init_check.stat.exists

# Check command return code
- name: Start service if not running
  ansible.builtin.service:
    name: myapp
    state: started
  when: service_check.rc != 0

# Check if item is in a list
- when: inventory_hostname in groups['databases']

# Check variable type
- when: my_var is string
- when: my_var is number
- when: my_var | type_debug == "list"

# Jinja2 tests
- when: my_path is file
- when: my_path is directory
- when: result is failed
- when: result is changed
- when: result is succeeded
```

`when` **with loops — condition evaluated per item:**

```yaml
- name: Only install items marked as active
  ansible.builtin.package:
    name: "{{ item.name }}"
  loop: "{{ packages }}"
  when: item.enabled | bool      # Checked for each item individually
```

## Q127. Explain `delegate_to` and `run_once`.

**Answer:**

**`delegate_to` — run a task on a different host than the current target:**

```
- name: Register web server in load balancer
  ansible.builtin.uri:
    url: "http://{{ lb_api }}/register"
    method: POST
    body_format: json
    body:
      server: "{{ inventory_hostname }}"    # Still refers to current host
      ip: "{{ ansible_default_ipv4.address }}"
  delegate_to: localhost                    # But task runs on CONTROL NODE

- name: Run database backup (from a specific backup server)
  ansible.builtin.shell: /opt/backup/run.sh {{ inventory_hostname }}
  delegate_to: backup-server.prod.example.com

- name: Drain server from load balancer BEFORE update
  community.aws.elb_instance:
    instance_id: "{{ instance_id }}"
    state: absent
    ec2_elbs: "{{ load_balancer_name }}"
  delegate_to: localhost                    # AWS API called from control node
  when: "'webservers' in group_names"
```

**`run_once` — execute on only ONE host in the play (first host by default):**

```
- name: Create database (only needs to happen once, not on all DB hosts)
  ansible.builtin.postgresql_db:
    name: "{{ db_name }}"
    state: present
  run_once: true

- name: Run database migration (once from a specific host)
  ansible.builtin.command: /opt/app/migrate.py
  run_once: true
  delegate_to: "{{ groups['appservers'][0] }}"   # Run once, but on first app server
```

**Common pattern — `run_once` + `delegate_to: localhost`:**

```
- name: Create AWS resource (only once, from control node)
  amazon.aws.ec2_vpc:
    state: present
    region: us-east-1
    cidr_block: 10.0.0.0/16
  run_once: true
  delegate_to: localhost
  register: vpc_result

- name: Use VPC ID on all hosts
  ansible.builtin.set_fact:
    vpc_id: "{{ hostvars[groups['all'][0]].vpc_result.vpc.id }}"
```

## Q128. What is `async` and `poll` in Ansible?

**Answer:**
For **long-running tasks**, Ansible's default synchronous mode (wait indefinitely) can hit SSH timeouts or block the play. `async` fires the task in the background; `poll` checks on it.

```
- name: Start long-running backup job (fire and forget)
  ansible.builtin.command: /opt/backup/full-backup.sh
  async: 3600      # Allow up to 1 hour to complete
  poll: 0          # Don't wait — return immediately

- name: Kick off app build on all servers simultaneously
  ansible.builtin.command: /opt/build/compile.sh
  async: 1800      # Max 30 minutes
  poll: 30         # Check every 30 seconds
  register: build_job

# ... do other tasks while build runs ...

- name: Check build job completion
  ansible.builtin.async_status:
    jid: "{{ build_job.ansible_job_id }}"
  register: job_result
  until: job_result.finished
  retries: 60
  delay: 30
```

**`poll: 0` (fire and forget) use cases:**
- Starting long-running jobs that you'll check later
- Running tasks in parallel across many hosts simultaneously (all start at once, you check later)
- Reboots (where connection is lost and you can't poll)

**`until` + `retries` + `delay` (retry loop):**

```
- name: Wait for application to be ready
  ansible.builtin.uri:
    url: http://localhost:8080/health
    status_code: 200
  register: health_check
  until: health_check.status == 200
  retries: 30          # Try up to 30 times
  delay: 10            # Wait 10 seconds between retries
  # Total max wait: 30 * 10 = 300 seconds
```

---

## Q129. Explain `changed_when`, `failed_when`, and `ignore_errors`.

**Answer:**
These three directives let you **customize what Ansible considers a change or failure**.

**`changed_when` — override change detection:**

```
# Shell always reports changed=true. Override it.
- name: Check disk usage (read-only, never changed)
  ansible.builtin.shell: df -h /
  register: disk_usage
  changed_when: false     # Always report as ok, never changed

- name: Run script — only changed if it returns rc=2
  ansible.builtin.shell: /opt/check-and-update.sh
  register: script_result
  changed_when: script_result.rc == 2    # Script returns 2 to indicate "I made a change"
  failed_when: script_result.rc > 2      # rc 0 (ok), 2 (changed), >2 (error)

- name: Add line to file (idempotent via grep check)
  ansible.builtin.shell: |
    grep -qF "{{ line }}" /etc/myapp.conf || echo "{{ line }}" >> /etc/myapp.conf
  register: line_result
  changed_when: "'added' in line_result.stdout"
```

**`failed_when` — custom failure conditions:**

```
- name: Check if service is registered in Consul
  ansible.builtin.uri:
    url: http://consul:8500/v1/health/service/myapp
  register: consul_check
  failed_when:
    - consul_check.status != 200
    - consul_check.json | length == 0      # Empty = not registered = failure

- name: Run test suite (allowed up to 5% failure rate)
  ansible.builtin.shell: pytest tests/ --json-report
  register: test_result
  failed_when:
    - test_result.rc != 0
    - (test_result.stdout | from_json).summary.failed > 5
```

**`ignore_errors` — continue even on failure:**

```
- name: Stop old service (may not exist on first run)
  ansible.builtin.service:
    name: old-app-v1
    state: stopped
  ignore_errors: true     # Continue even if service doesn't exist

# Better alternative — check first:
- name: Check if old service exists
  ansible.builtin.stat:
    path: /etc/systemd/system/old-app-v1.service
  register: old_service

- name: Stop old service if it exists
  ansible.builtin.service:
    name: old-app-v1
    state: stopped
  when: old_service.stat.exists
```

**`ignore_errors` vs `failed_when: false`:**

- `ignore_errors: true` — task fails, Ansible records failure but continues; `any_errors_fatal` still

triggers
- `failed_when: false` — task is never considered failed; cleaner semantics

## SECTION E: Inventory, Performance & Production

### Q130. How do `group_vars` and `host_vars` work?

**Answer:**

`group_vars` and `host_vars` are **directory-based variable files** that automatically apply to groups and hosts — no need to specify them in the playbook.

**Directory structure:**

```
inventory/
├── hosts.yml           # Inventory file
├── group_vars/
│   ├── all/            # Applies to ALL hosts
│   │   ├── common.yml
│   │   └── secrets.yml  # vault-encrypted
│   ├── webservers/      # Applies to "webservers" group
│   │   ├── nginx.yml
│   │   └── monitoring.yml
│   └── databases/
│       └── postgres.yml
└── host_vars/
    ├── web1.prod.example.com/
    │   └── specific.yml
    └── db1.prod.example.com.yml   # Single file also works
```

`group_vars/all/common.yml`:

```
# Applied to every host in inventory
ntp_servers:
  - 0.pool.ntp.org
  - 1.pool.ntp.org
dns_servers:
  - 10.0.0.2
  - 8.8.8.8
log_level: info
```

`group_vars/webservers/nginx.yml`:

```
# Applied only to hosts in "webservers" group
nginx_port: 80
nginx_worker_connections: 1024
enable_ssl: true
```

`host_vars/web1.prod.example.com/specific.yml`:

```
# Applied ONLY to web1.prod.example.com (highest specificity wins)
nginx_port: 8080     # This specific host uses port 8080
primary_lb: true
```

**Why directories instead of single files:**
Splitting into multiple files per group lets you encrypt only the secrets file with Ansible Vault while keeping other vars in plaintext — cleaner than encrypting the whole file.

---

## Q131. How do you target specific hosts using patterns?

**Answer:**
Ansible patterns (also called host patterns) control **which hosts from inventory are targeted** by a play or ad-hoc command.

```
# All hosts
ansible all -m ping
ansible '*' -m ping

# Specific group
ansible webservers -m ping

# Specific host
ansible web1.prod.example.com -m ping

# Multiple groups (union — OR)
ansible webservers:databases -m ping     # All webservers AND all databases

# Intersection (AND — hosts in BOTH groups)
ansible 'webservers:&prod' -m ping      # Hosts in webservers AND in prod

# Exclusion (NOT)
ansible 'webservers:!staging' -m ping   # webservers but NOT staging hosts

# Wildcard
ansible 'web*.prod.*' -m ping

# Regex
ansible '~^web[0-9]+' -m ping          # Regex match

# Limit to specific host within a play
ansible-playbook deploy.yml --limit web1.prod.example.com
ansible-playbook deploy.yml --limit 'webservers:!web3'

# Use a file as limit (one host per line)
ansible-playbook deploy.yml --limit @failed_hosts.txt
```

**In playbooks:**

```
- hosts: webservers:&prod              # Intersection in playbook
- hosts: all:!monitoring               # All except monitoring group
- hosts: "{{ target_group | default('webservers') }}"  # Dynamic targeting via variable
```

---

## Q132. How do you improve Ansible performance at scale?

**Answer:**

At 100+ hosts, default Ansible becomes noticeably slow. Here are the production-grade optimizations:

**1. SSH pipelining (biggest single improvement):**

```
# ansible.cfg
[ssh_connection]
pipelining = True     # Saves ~50% of task execution time
# Removes need to write temp files; executes modules over stdin
# Requires: Defaults !requiretty in /etc/sudoers
```

**2. Increase forks:**

```
[defaults]
forks = 50     # Default is 5; set to number of hosts or more
```

**3. SSH connection reuse (ControlMaster):**

```
[ssh_connection]
control_master = auto
control_persist = 60s
control_path = /tmp/ansible-%%r@%%h:%%p
```

**4. Fact caching (huge for large inventories):**

```
[defaults]
gathering = smart
fact_caching = redis
fact_caching_connection = localhost:6379:0
fact_caching_timeout = 86400
```

With smart gathering, facts are only collected if not in cache. Re-runs skip fact gathering entirely.

**5. Mitogen for Ansible (10x speedup):**

```
pip install mitogen
```

```
[defaults]
strategy_plugins = /path/to/mitogen/ansible_mitogen/plugins/strategy
strategy = mitogen_linear
```

Mitogen replaces Ansible's SSH transport with a custom Python-based transport. Dramatic speedup by eliminating repeated SSH negotiation and temp file operations.

**6. `gather_facts: false` where not needed:**

```
- hosts: webservers
  gather_facts: false    # Skip if you don't need OS facts — saves 1-3s per host
```

**7. Free strategy for independent tasks:**

```
- hosts: webservers
  strategy: free    # Don't wait for all hosts to finish each task before moving on
```

**8. Use `async` for long tasks:**
Instead of all hosts running a slow task sequentially, fire all async simultaneously and collect results.

---

## Q133. What is Ansible Tower / AWX?

**Answer:**
**AWX** is the open-source project. **Ansible Tower** is the commercial, supported version (now part of Red Hat Ansible Automation Platform).

Both provide a **web UI + REST API + RBAC layer on top of Ansible** — transforming CLI automation into an enterprise platform.

**Core features:**
- **Visual Dashboard:** Job status, recent runs, inventory health
- **RBAC:** Teams, organizations, credentials — control who runs what on which systems
- **Credentials Management:** Encrypted storage for SSH keys, vault passwords, cloud credentials
- **Job Templates:** Standardized, repeatable playbook runs with locked-in parameters
- **Workflows:** Chain multiple job templates with success/failure branching
- **Surveys:** Simple forms that collect variables from non-technical users before job runs
- **Scheduling:** Run playbooks on cron-like schedule
- **Notifications:** Slack/email on job success/failure
- **API:** Every action available via REST — integrate with ITSM, CI/CD, chatops

**Mental Model:**
AWX/Tower = Jenkins for Ansible. Just as Jenkins provides a UI, scheduling, and RBAC around running scripts, AWX provides the same for Ansible playbooks.

**Workflow example:**

```
Workflow: Deploy Production
├── Job 1: Run smoke tests (inventory: staging)
│   ├── SUCCESS → Job 2: Deploy to prod-canary (5% of hosts)
│   │   ├── SUCCESS → Job 3: Run canary health checks
│   │   │   ├── SUCCESS → Job 4: Full prod deployment
│   │   │   └── FAILURE → Job 5: Rollback canary
│   └── FAILURE → Job 6: Notify team
```

## Q134. How do you integrate Ansible with Terraform?

**Answer:**

Terraform provisions infrastructure. Ansible configures it. Integration is at the handoff point.

**Pattern 1 — Terraform `local-exec` provisioner calls Ansible:**

```
resource "aws_instance" "web" {
  ami           = data.aws_ami.amazon_linux.id
  instance_type = "t3.medium"

  provisioner "local-exec" {
    command = <<-EOT
      sleep 30  # Wait for SSH to be ready
      ansible-playbook -i '${self.public_ip},' \
        --private-key ~/.ssh/prod-key.pem \
        -e "env=${var.environment}" \
        playbooks/configure-web.yml
    EOT
  }
}
```

**Pattern 2 — Dynamic inventory from Terraform state (recommended):**

Ansible reads Terraform state file to build inventory:

```
# inventory/terraform.yaml
plugin: cloud.terraform.terraform_provider
project_path: ../terraform/
```

Or use `terraform output -json` to generate inventory:

```
terraform output -json instance_ips | python3 -c "
import json, sys
ips = json.load(sys.stdin)
print('[webservers]')
for ip in ips: print(ip)
" > inventory/tf-hosts.ini
ansible-playbook -i inventory/tf-hosts.ini configure.yml
```

**Pattern 3 — Terraform Remote State data source + Ansible variables:**

```
# Dynamic inventory script
import subprocess, json

result = subprocess.run(
    ['terraform', 'output', '-json'],
    cwd='../terraform', capture_output=True, text=True
)
outputs = json.loads(result.stdout)
# Build Ansible inventory JSON from Terraform outputs
```

**Pattern 4 — CI/CD pipeline orchestration:**

```
# GitHub Actions
- name: Terraform Apply
  run: terraform apply -auto-approve

- name: Get Instance IPs
  run: terraform output -json instance_ips > /tmp/ips.json

- name: Run Ansible
  run: ansible-playbook -i /tmp/ips.json configure.yml
```

## Q135. How do you troubleshoot Ansible playbook issues?

**Answer:**

**Verbosity levels:**

```
ansible-playbook deploy.yml -v      # Show task results
ansible-playbook deploy.yml -vv     # Show file/connection info
ansible-playbook deploy.yml -vvv    # Show SSH commands
ansible-playbook deploy.yml -vvvv   # Show SSH connection details (debug level)
```

**Step-by-step execution:**

```
ansible-playbook deploy.yml --step    # Confirm each task (y/n/c to continue all)
```

**Start at specific task:**

```
ansible-playbook deploy.yml --start-at-task="Deploy nginx configuration"
```

**Debug module:**

```
- name: Show variable value
  ansible.builtin.debug:
    var: my_variable          # Pretty-prints the variable

- name: Show message
  ansible.builtin.debug:
    msg: "Value is {{ my_var }}, type is {{ my_var | type_debug }}"
    verbosity: 2              # Only show at -vv or higher

- name: Dump all variables for this host
  ansible.builtin.debug:
    var: hostvars[inventory_hostname]
```

**Test connectivity and variables:**

```
# Test if host is reachable
ansible web1 -m ping

# Show all facts for a host
ansible web1 -m setup
ansible web1 -m setup -a "filter=ansible_distribution*"

# Show all variables (inventory + facts) for a host
ansible-inventory -i inventory/ --host web1.prod.example.com

# Validate playbook syntax (no connection to hosts)
ansible-playbook deploy.yml --syntax-check

# Preview what hosts would run
ansible-playbook deploy.yml --list-hosts

# Preview what tasks would run
ansible-playbook deploy.yml --list-tasks

# Use template rendering standalone
ansible localhost -m debug -a "msg={{ 'hello' | upper }}"
```

**Common error patterns:**

```
"MODULE FAILURE" → Python error on remote; check python version
"Permission denied" → SSH key issue or become problem
"Timeout" → Increase timeout; check network path
"UNREACHABLE" → Host down; wrong IP; SSH misconfigured
"No module named X" → Collection not installed; wrong FQCN
"variable undefined" → Variable name typo; wrong scope; conditional not met
```

---

## Q136. How do Ansible Callback Plugins work? Which ones are most useful?

**Answer:**

Callback plugins let Ansible **hook into events** (task start, task result, play end) to customize output or send data to external systems.

**Enable in `ansible.cfg`:**

```
[defaults]
stdout_callback    = yaml          # Replaces default output with readable YAML format
callbacks_enabled  = profile_tasks,timer,mail,slack
```

**Useful callbacks:**

`yaml` — Readable output format (much better than default):

```
TASK [Install nginx] ****
ok: [web1.prod.example.com]
```

`profile_tasks` — Timing for every task:

```
Wednesday 24 January 2024  14:23:05 +0000 (0:00:03.241)  ========
Install nginx ------------------------------------------------- 3.24s
Deploy configuration ------------------------------------------ 1.15s
Restart service ----------------------------------------------- 0.89s
```

`timer` — Total playbook execution time

`json` — Output entire run as JSON (for machine parsing / CI integration)

`slack` — Post notifications to Slack:

```
[callback_slack]
webhook_url = https://hooks.slack.com/services/T.../B.../xxx
channel = #deployments
username = Ansible
```

`junit` — Output JUnit XML (for CI test reporting: Jenkins, GitLab)

**Custom callback plugin example (simplified):**

```
# callback_plugins/send_to_datadog.py
from ansible.plugins.callback import CallbackBase

class CallbackModule(CallbackBase):
    def v2_runner_on_ok(self, result):
        # Send successful task metric to Datadog
        pass

    def v2_playbook_on_stats(self, stats):
        # Send play summary to Datadog
        pass
```

### Q137. How do you implement a zero-downtime rolling deployment with Ansible?

**Answer:**

This combines serial execution, load balancer drain, health verification, and rollback capability.

```yaml
---
- name: Zero-downtime rolling deployment
  hosts: webservers
  serial: "25%"              # Deploy to 25% of servers at a time
  max_fail_percentage: 0     # Abort entire play if ANY batch fails
  become: true

  vars:
    app_version: "{{ version | mandatory }}"  # Must pass -e version=X.Y.Z
    previous_version: "{{ current_version.stdout }}"

  pre_tasks:
    - name: Get current app version (for rollback reference)
      ansible.builtin.command: cat /opt/app/VERSION
      register: current_version
      changed_when: false
      failed_when: false

  tasks:
    - name: Remove host from load balancer
      community.aws.elb_instance:
        instance_id: "{{ instance_id }}"
        state: absent
        ec2_elbs: ["prod-web-lb"]
        wait: true
        wait_timeout: 60
      delegate_to: localhost

    - name: Wait for in-flight requests to drain
      ansible.builtin.pause:
        seconds: 30

    - block:
        - name: Stop application service
          ansible.builtin.systemd:
            name: myapp
            state: stopped

        - name: Deploy new application version
          ansible.builtin.unarchive:
            src: "s3://releases/myapp-{{ app_version }}.tar.gz"
            dest: /opt/app
            remote_src: true

        - name: Update version file
          ansible.builtin.copy:
            content: "{{ app_version }}"
            dest: /opt/app/VERSION

        - name: Run database migrations (once per batch)
          ansible.builtin.command: /opt/app/migrate.sh
          run_once: true

        - name: Start application service
          ansible.builtin.systemd:
            name: myapp
            state: started

        - name: Wait for app to be healthy
          ansible.builtin.uri:
            url: http://localhost:8080/health
            status_code: 200
          register: health
          until: health.status == 200
          retries: 20
          delay: 5
```

```
      rescue:
        - name: Rollback to previous version
          ansible.builtin.unarchive:
            src: "s3://releases/myapp-{{ previous_version }}.tar.gz"
            dest: /opt/app
            remote_src: true

        - name: Restart with previous version
          ansible.builtin.systemd:
            name: myapp
            state: restarted

        - name: Mark play as failed after rollback
          ansible.builtin.fail:
            msg: "Deployment of {{ app_version }} failed. Rolled back to {{
previous_version }}."

    - name: Add host back to load balancer
      community.aws.elb_instance:
        instance_id: "{{ instance_id }}"
        state: present
        ec2_elbs: ["prod-web-lb"]
        wait: true
      delegate_to: localhost

    - name: Verify host is healthy in load balancer
      ansible.builtin.pause:
        seconds: 30       # Observe before moving to next batch
```

## Q138. How do you manage secrets in Ansible at enterprise scale?

**Answer:**

Beyond basic Ansible Vault, enterprise secret management integrates with dedicated secret stores.

**Tier 1 — Ansible Vault (small teams):**

- Encrypted files in git

- Password shared via secure channel (1Password, LastPass team vault)

- CI/CD: vault password as pipeline secret

**Tier 2 — HashiCorp Vault integration:**

```
# Using the hashi_vault lookup
- name: Get database password from Vault
  vars:
    db_pass: "{{ lookup('community.hashi_vault.hashi_vault',
      'secret=secret/prod/database:password
       url=https://vault.internal:8200
       auth_method=aws_iam
       role_id=prod-ansible') }}"
  ansible.builtin.template:
    src: db.conf.j2
    dest: /etc/app/db.conf
```

**Tier 3 — AWS Secrets Manager:**

```
- name: Retrieve all secrets for prod environment
  set_fact:
    db_credentials: "{{ lookup('amazon.aws.aws_secret',
      'prod/myapp/db', region='us-east-1') | from_json }}"

- name: Configure app
  template:
    src: app.conf.j2
    dest: /etc/app/app.conf
  vars:
    db_host: "{{ db_credentials.host }}"
    db_pass: "{{ db_credentials.password }}"
```

**Security practices:**

- Never log sensitive variables: `no_log: true` on tasks that handle secrets
- Use `no_log` on loops that iterate over credentials
- Rotate vault passwords regularly
- Audit who has access to vault password file/CI secret

```
- name: Create user with password (sensitive — hide from logs)
  ansible.builtin.user:
    name: serviceaccount
    password: "{{ service_password | password_hash('sha512') }}"
  no_log: true          # Hides entire task output from console/logs
```

---

## Q139. How do Ansible Galaxy and private Automation Hub work?

**Answer:**

**Ansible Galaxy** (`galaxy.ansible.com`) is the **public marketplace** for roles and collections — the npm registry or Terraform Registry equivalent.

```
# Install a role from Galaxy
ansible-galaxy role install geerlingguy.nginx

# Install a collection from Galaxy
ansible-galaxy collection install amazon.aws

# Install from requirements file (recommended for reproducible builds)
ansible-galaxy install -r requirements.yml

# Generate a role skeleton
ansible-galaxy role init my_nginx_role
```

`requirements.yml`:

```yaml
roles:
  - name: geerlingguy.nginx
    version: "3.2.0"
  - name: my_internal_role
    src: https://git.company.internal/ansible/roles/internal-hardening.git
    scm: git
    version: main

collections:
  - name: amazon.aws
    version: ">=6.0.0"
  - name: community.docker
    version: "3.4.0"
  - name: https://my-private-hub.internal/
    type: url
```

**Private Automation Hub (Red Hat):** Self-hosted Galaxy for enterprises. Store proprietary/certified collections internally. Mirror approved public collections. Apply organizational governance (only approved collections allowed in playbooks).

**Vendir / Ansible Collections in CI:**

```
# Pin all dependencies for reproducible CI
ansible-galaxy collection install -r requirements.yml \
  --collections-path ./collections \
  --force    # Ensure exact versions
```

---

### Q140. How do you handle multi-environment configuration (dev/staging/prod) in Ansible?

**Answer:**

The **inventory-per-environment** pattern is the gold standard for separating environments.

**Directory structure:**

```
ansible/
├── inventories/
│   ├── dev/
│   │   ├── hosts.yml
│   │   ├── group_vars/
│   │   │   ├── all/
│   │   │   │   ├── common.yml
│   │   │   │   └── secrets.yml   # vault-encrypted dev secrets
│   │   │   └── webservers.yml
│   │   └── host_vars/
│   ├── staging/
│   │   ├── hosts.yml             # Staging-specific hosts
│   │   ├── group_vars/
│   │   │   └── all/
│   │   │       └── common.yml    # Override: staging-specific values
│   │   └── ...
│   └── prod/
│       ├── hosts.yml             # Prod hosts
│       ├── group_vars/
│       │   └── all/
│       │       ├── common.yml    # Override: prod-specific values
│       │       └── secrets.yml   # DIFFERENT vault password than dev
│       └── ...
├── playbooks/
│   ├── deploy-app.yml
│   └── configure-baseline.yml
└── roles/
    └── ...
```

**Running against specific environment:**

```
ansible-playbook -i inventories/prod playbooks/deploy-app.yml
ansible-playbook -i inventories/dev  playbooks/deploy-app.yml
```

**Environment-specific variables:**

```yaml
# inventories/dev/group_vars/all/common.yml
env: dev
log_level: debug
replicas: 1
db_instance_class: db.t3.micro

# inventories/prod/group_vars/all/common.yml
env: prod
log_level: warn
replicas: 3
db_instance_class: db.r5.xlarge
```

**Never mix environments** — don't use `--limit` to target prod hosts from a dev inventory. Each environment has its own inventory, its own vault password, its own credentials.

---

## Q141. What are Ansible Filters vs Tests? Show production examples.

**Answer:**

**Filters** — transform a value (pipe syntax `|`):

```
# String filters
"{{ 'Hello World' | lower }}"          # hello world
"{{ my_list | join(', ') }}"           # "a, b, c"
"{{ my_string | regex_replace('^foo', 'bar') }}"
"{{ my_dict | combine(override_dict) }}" # Merge dicts (right wins)
"{{ my_list | flatten }}"              # [[1,2],[3]] → [1,2,3]
"{{ my_list | zip(other_list) | list }}" # Zip two lists

# Conditional
"{{ my_var | default('fallback', boolean=true) }}"  # Use fallback if falsy/undefined
"{{ (x > 10) | ternary('big', 'small') }}"

# Data format
"{{ my_dict | to_nice_json(indent=2) }}"
"{{ my_dict | to_nice_yaml }}"
"{{ my_string | from_yaml }}"

# Math
"{{ 1024 * 1024 | int }}"              # Integer math
"{{ [1,5,3,2] | max }}"               # 5
"{{ [1,5,3,2] | sum }}"               # 11

# Select/reject from lists
"{{ my_list | select('match', 'web.*') | list }}"
"{{ my_list | reject('equalto', 'localhost') | list }}"
"{{ my_list | map('upper') | list }}"
"{{ my_list | map(attribute='name') | list }}"  # Extract attribute from list of dicts

# Dict manipulation
"{{ my_dict | dict2items }}"           # [{'key': k, 'value': v}, ...]
"{{ my_list | items2dict }}"           # Reverse of above
"{{ my_dict | dict2items | selectattr('key', 'match', 'prod_.*') | list | items2dict }}"
```

**Tests** — return boolean (used in `when`, `assert`, and `|` for filtering):

```
- when: my_var is defined
- when: my_var is undefined
- when: my_var is none
- when: my_var is string
- when: my_var is number
- when: my_var is iterable
- when: my_var is mapping         # Dict
- when: my_var is sequence        # List or string
- when: result is failed
- when: result is changed
- when: result is succeeded
- when: result is skipped
- when: my_path is file
- when: my_path is directory
- when: my_path is link
- when: my_string is match('web.*')        # Regex from start
- when: my_string is search('nginx')       # Regex anywhere
- when: my_item is in my_list
- when: my_item is not in my_list
- when: my_var is version('2.0', '>=')     # Version comparison
```

## Q142. What is the difference between `include_vars` and `vars_files`?

**Answer:**

**`vars_files` (play-level, static):**

```
- name: Deploy application
  hosts: webservers
  vars_files:
    - vars/common.yml
    - vars/{{ env }}.yml          # Dynamic filename — resolved at play start
    - "{{ 'vars/secrets.yml' if env == 'prod' else 'vars/dev-secrets.yml' }}"
```

Loaded once at play start, before any tasks run.

**`include_vars` (task-level, dynamic):**

```
tasks:
  - name: Load OS-specific variables
    ansible.builtin.include_vars:
      file: "vars/{{ ansible_os_family }}.yml"

  - name: Load environment variables after fact collection
    ansible.builtin.include_vars:
      dir: vars/
      extensions: [yaml, yml]
      ignore_files: [secrets.yml]
      depth: 1

  - name: Load and name variables from file
    ansible.builtin.include_vars:
      file: vars/app-config.yml
      name: app_config    # Load into this variable namespace
      # Access as: app_config.key
```

**When to use which:**
- `vars_files` → known at play write time, loaded before any task runs
- `include_vars` → dynamic filename based on facts (gathered at runtime), conditional loading, loading into a named namespace

---

## Q143. How does Ansible handle Windows targets?

**Answer:**
Windows uses **WinRM (Windows Remote Management)** instead of SSH as the transport.

**Setup requirements on Windows target:**

```
# Run on Windows target as Administrator
[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12
$url = "https://raw.githubusercontent.com/ansible/ansible/devel/examples/scripts/
ConfigureRemotingForAnsible.ps1"
$file = "$env:temp\ConfigureRemotingForAnsible.ps1"
(New-Object -TypeName System.Net.WebClient).DownloadFile($url, $file)
powershell.exe -ExecutionPolicy ByPass -File $file
```

**Inventory for Windows:**

```
hosts:
  windows_server:
    ansible_host: 192.168.1.50
    ansible_user: Administrator
    ansible_password: "{{ vault_win_password }}"
    ansible_connection: winrm
    ansible_winrm_transport: ntlm     # or credssp, kerberos
    ansible_winrm_server_cert_validation: ignore
    ansible_port: 5985                # HTTP (5986 for HTTPS)
```

**Windows-specific modules:**

```
tasks:
  - name: Install IIS
    ansible.windows.win_feature:
      name: Web-Server
      state: present
      include_management_tools: true

  - name: Copy file to Windows
    ansible.windows.win_copy:
      src: files/app.exe
      dest: C:\App\app.exe

  - name: Run PowerShell script
    ansible.windows.win_powershell:
      script: |
        $processes = Get-Process
        $processes | Where-Object { $_.CPU -gt 90 }

  - name: Manage Windows service
    ansible.windows.win_service:
      name: MyService
      state: started
      start_mode: auto

  - name: Set registry key
    ansible.windows.win_regedit:
      path: HKLM:\SOFTWARE\MyApp
      name: Version
      data: "2.1.4"
      type: String
```

## SECTION F: Design & Senior-Level Topics

### Q144. Design an Ansible project structure for a company with 500+ servers, 10 teams.

**Answer:**

```
ansible-platform/
├── inventories/
│   ├── production/
│   │   ├── 00_aws_ec2.yaml         # Dynamic: AWS EC2 instances
│   │   ├── 01_on_prem.yaml         # Static: on-prem hosts
│   │   ├── group_vars/
│   │   │   ├── all/
│   │   │   │   ├── 00_global.yml   # Global vars (numbered for order)
│   │   │   │   └── 01_secrets.yml  # Vault-encrypted
│   │   │   ├── webservers/
│   │   │   ├── databases/
│   │   │   └── kubernetes_nodes/
│   │   └── host_vars/
│   ├── staging/
│   └── dev/
│
├── playbooks/
│   ├── site.yml                    # Master playbook (imports all roles)
│   ├── webservers.yml
│   ├── databases.yml
│   └── common-baseline.yml
│
├── roles/
│   ├── common/                     # Runs on every server (NTP, DNS, logging, hardening)
│   ├── monitoring/                 # Prometheus node exporter, Filebeat
│   ├── nginx/
│   ├── postgresql/
│   ├── docker/
│   └── kubernetes_node/
│
├── collections/                    # Vendored collections (pinned versions)
│   └── ansible_collections/
│       ├── amazon/aws/
│       └── community/docker/
│
├── molecule/                       # Testing for all roles
│   └── (per-role in roles/*/molecule/)
│
├── library/                        # Custom modules for company-specific tasks
│   └── deploy_service.py
│
├── filter_plugins/                 # Custom Jinja2 filters
│   └── company_filters.py
│
├── callback_plugins/               # Custom callbacks
│   └── datadog_events.py
│
├── requirements.yml                # Galaxy dependencies
├── ansible.cfg                     # Project configuration
└── Makefile                        # make deploy-prod, make lint, make test
```

**Governance model:**

- `common` role mandatory for all servers (enforced in `site.yml`)
- Team-specific roles in namespaced subdirectories

- PR required to merge to `main` → triggers Molecule tests in CI
- AWX/Tower enforces RBAC: team A can only run their playbooks against their inventory groups

---

## Q145. What are custom Ansible modules and when do you write one?

**Answer:**

Write a custom module when:

- No existing module does what you need
- Interacting with a proprietary internal API/system
- Encapsulating complex idempotency logic that shell/command can't cleanly handle

**Custom module in Python:**

```python
#!/usr/bin/python3
# library/my_app_deploy.py

from ansible.module_utils.basic import AnsibleModule
import requests

DOCUMENTATION = '''
module: my_app_deploy
short_description: Deploy application via internal deployment API
'''

def run_module():
    module_args = dict(
        app_name=dict(type='str', required=True),
        version=dict(type='str', required=True),
        environment=dict(type='str', default='staging',
                         choices=['dev', 'staging', 'prod']),
        api_token=dict(type='str', required=True, no_log=True),  # Sensitive
    )

    result = dict(changed=False, message='', version='')

    module = AnsibleModule(
        argument_spec=module_args,
        supports_check_mode=True    # Implement dry run support
    )

    # Idempotency check — get current deployed version
    current = requests.get(
        f"https://deploy-api.internal/apps/{module.params['app_name']}/version",
        headers={'Authorization': f"Bearer {module.params['api_token']}"}
    ).json()['version']

    if current == module.params['version']:
        result['message'] = 'Already at target version'
        module.exit_json(**result)    # changed=False, no action

    if module.check_mode:            # Dry run — don't make actual change
        result['changed'] = True
        result['message'] = f"Would deploy {module.params['version']}"
        module.exit_json(**result)

    # Make the actual change
    response = requests.post(
        f"https://deploy-api.internal/deploy",
        json={
            'app': module.params['app_name'],
            'version': module.params['version'],
            'env': module.params['environment'],
        },
        headers={'Authorization': f"Bearer {module.params['api_token']}"}
    )

    if response.status_code != 200:
        module.fail_json(msg=f"API error: {response.text}", **result)

    result['changed'] = True
    result['version'] = module.params['version']
    result['message'] = f"Deployed {module.params['version']} successfully"
    module.exit_json(**result)

if __name__ == '__main__':
    run_module()
```

**Usage in playbook:**

```
- name: Deploy application via internal API
  my_app_deploy:
    app_name: payment-service
    version: "{{ deploy_version }}"
    environment: "{{ env }}"
    api_token: "{{ vault_deploy_api_token }}"
  register: deploy_result

- debug:
    msg: "Deployed: {{ deploy_result.version }}"
  when: deploy_result.changed
```

## Q146. How do you implement CI/CD for Ansible playbooks?

**Answer:**

**Pipeline stages:**

```
# .github/workflows/ansible-ci.yml
stages:
  lint → test → deploy-dev → gate → deploy-staging → gate → deploy-prod
```

**Full GitHub Actions pipeline:**

```yaml
name: Ansible CI/CD

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Install tools
        run: pip install ansible ansible-lint yamllint

      - name: YAML lint
        run: yamllint .

      - name: Ansible syntax check
        run: ansible-playbook playbooks/site.yml --syntax-check -i inventories/dev

      - name: Ansible lint
        run: ansible-lint

  test:
    runs-on: ubuntu-latest
    needs: lint
    strategy:
      matrix:
        role: [nginx, postgresql, common]
    steps:
      - uses: actions/checkout@v4

      - name: Install Molecule + Docker driver
        run: pip install molecule molecule-docker docker

      - name: Run Molecule tests for {{ matrix.role }}
        run: molecule test
        working-directory: roles/${{ matrix.role }}

  deploy-dev:
    needs: test
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'
    steps:
      - name: Deploy to Dev
        run: |
          ansible-playbook \
            -i inventories/dev \
            --vault-password-file <(echo "$VAULT_PASS_DEV") \
            playbooks/site.yml
        env:
          VAULT_PASS_DEV: ${{ secrets.ANSIBLE_VAULT_PASS_DEV }}

  deploy-prod:
    needs: deploy-dev
    environment: production      # Requires manual approval in GitHub
    runs-on: ubuntu-latest
    steps:
      - name: Deploy to Production
        run: |
          ansible-playbook \
            -i inventories/prod \
            --vault-password-file <(echo "$VAULT_PASS_PROD") \
```

```
            --diff \
            playbooks/site.yml
        env:
          VAULT_PASS_PROD: ${{ secrets.ANSIBLE_VAULT_PASS_PROD }}
```

**Ansible Lint rules to enforce:**

```
# .ansible-lint
warn_list:
  - yaml[truthy]
skip_list:
  - role-name
rules:
  no-free-form: true
  fqcn: true                      # Enforce fully qualified module names
  no-changed-when: true           # Enforce changed_when on shell/command
  var-naming: true
```

---

### Q147. How do you manage package and service state across multiple Linux distributions?

**Answer:**

The core challenge: Ubuntu uses `apt`, CentOS uses `dnf/yum`, service names differ, config paths differ.

**Pattern 1 — `ansible.builtin.package` (generic module):**

```
# Works on all distros — automatically uses correct package manager
- name: Install nginx
  ansible.builtin.package:
    name: nginx
    state: present
```

Limitation: package names can differ between distros (`httpd` vs `apache2`).

**Pattern 2 — OS-specific variable files:**

```yaml
# tasks/main.yml
- name: Load OS-specific variables
  ansible.builtin.include_vars:
    file: "{{ ansible_os_family }}.yml"

# vars/Debian.yml
packages:
  web_server: apache2
  python: python3
config_dir: /etc/apache2
service_name: apache2

# vars/RedHat.yml
packages:
  web_server: httpd
  python: python3
config_dir: /etc/httpd
service_name: httpd
```

```yaml
# tasks/main.yml (continued)
- name: Install web server
  ansible.builtin.package:
    name: "{{ packages.web_server }}"
    state: present

- name: Deploy config
  ansible.builtin.template:
    src: web.conf.j2
    dest: "{{ config_dir }}/web.conf"

- name: Start and enable service
  ansible.builtin.service:
    name: "{{ service_name }}"
    state: started
    enabled: true
```

**Pattern 3 — Conditional tasks with `when`:**

```yaml
- name: Install on Debian-based systems
  ansible.builtin.apt:
    name: nginx
    state: present
    update_cache: true
  when: ansible_os_family == "Debian"

- name: Install on RedHat-based systems
  ansible.builtin.dnf:
    name: nginx
    state: present
  when: ansible_os_family == "RedHat"
```

**Best approach:** Combine pattern 2 (variable files for all distro-specific values) with pattern 1 (generic modules where possible). Result: clean tasks, all differences in variable files.

## Q148. What is `ansible-pull` and when do you use it?

**Answer:**

`ansible-pull` **inverts the Ansible push model** — each target node pulls its own playbook from a git repository and runs it locally. No control node required.

```
# Run on the TARGET machine (not control node)
ansible-pull -U https://github.com/myorg/ansible-configs.git \
  --checkout main \
  playbooks/configure-node.yml
```

**Architecture comparison:**

```
PUSH (normal):
Control Node → SSH → [target1, target2, target3, ...]
One control node manages all targets.

PULL (ansible-pull):
git repo → target1 (pulls own config and runs)
git repo → target2 (pulls own config and runs)
git repo → target3 (pulls own config and runs)
No central control node needed at runtime.
```

**When to use `ansible-pull`:**

- Large-scale deployments (thousands of nodes) — eliminates control node bottleneck
- Immutable infrastructure with periodic config enforcement (run via cron every 15 min)
- Edge computing / IoT devices that call home periodically
- Bootstrapping new machines (user data runs `ansible-pull` on first boot)

**EC2 User Data bootstrapping:**

```
#!/bin/bash
apt-get install -y ansible git
ansible-pull \
  -U https://github.com/myorg/ansible-configs.git \
  -i localhost, \
  --checkout "$(aws ssm get-parameter --name /config/branch --query Parameter.Value --output
text)" \
  playbooks/bootstrap-node.yml
```

**Limitation:** Secrets management is harder (no centralized vault password distribution — must store vault password on each node or use cloud secrets manager).

## Q149. How do you harden a Linux server using Ansible?

**Answer:**

Security hardening via Ansible follows CIS Benchmarks or STIG standards — typically a dedicated `hardening` or `common` role.

**Key hardening tasks:**

```yaml
---
- name: System hardening baseline
  hosts: all
  become: true

  tasks:
    # SSH hardening
    - name: Harden SSH configuration
      ansible.builtin.template:
        src: sshd_config.j2
        dest: /etc/ssh/sshd_config
        validate: /usr/sbin/sshd -t -f %s    # Validate before deploying
      notify: Restart sshd

    # sshd_config.j2 key settings:
    # PermitRootLogin no
    # PasswordAuthentication no
    # PubkeyAuthentication yes
    # X11Forwarding no
    # MaxAuthTries 3
    # AllowTcpForwarding no
    # Protocol 2

    # Kernel hardening (sysctl)
    - name: Apply sysctl hardening
      ansible.posix.sysctl:
        name: "{{ item.name }}"
        value: "{{ item.value }}"
        state: present
        reload: true
      loop:
        - { name: net.ipv4.ip_forward, value: "0" }
        - { name: net.ipv4.conf.all.send_redirects, value: "0" }
        - { name: net.ipv4.conf.all.accept_redirects, value: "0" }
        - { name: net.ipv4.tcp_syncookies, value: "1" }
        - { name: kernel.dmesg_restrict, value: "1" }
        - { name: fs.suid_dumpable, value: "0" }

    # User management
    - name: Ensure root password is locked
      ansible.builtin.user:
        name: root
        password_lock: true

    - name: Remove unauthorized users
      ansible.builtin.user:
        name: "{{ item }}"
        state: absent
      loop: "{{ users_to_remove | default([]) }}"

    # Package management
    - name: Remove unnecessary packages
      ansible.builtin.package:
        name: "{{ item }}"
        state: absent
      loop:
        - telnet
        - ftp
        - rsh-client

    # Firewall (UFW for Debian, firewalld for RedHat)
    - name: Configure UFW - default deny
      community.general.ufw:
        state: enabled
        policy: deny
        direction: incoming
```

```
      when: ansible_os_family == "Debian"

    - name: Configure UFW - allow SSH
      community.general.ufw:
        rule: allow
        port: "22"
        proto: tcp

    # Auditd (audit logging)
    - name: Install and configure auditd
      ansible.builtin.package:
        name: auditd
        state: present

    - name: Configure audit rules
      ansible.builtin.template:
        src: audit.rules.j2
        dest: /etc/audit/rules.d/hardening.rules
      notify: Restart auditd

    # Compliance check with openscap
    - name: Run OpenSCAP compliance scan
      ansible.builtin.command: >
        oscap xccdf eval
        --profile xccdf_org.ssgproject.content_profile_cis
        --results /var/log/openscap-results.xml
        /usr/share/xml/scap/ssg/content/ssg-ubuntu2204-ds.xml
      changed_when: false
      failed_when: false
      register: scap_result

    - name: Archive compliance report
      ansible.builtin.fetch:
        src: /var/log/openscap-results.xml
        dest: "reports/{{ inventory_hostname }}-scap-{{ ansible_date_time.date }}.xml"
        flat: true
```

**Q150. Design an end-to-end Ansible architecture for deploying a microservices app across 3 environments.**

**Answer:**

```
┌──────────────────────────────────────────────────────────────────────┐
│ ANSIBLE PLATFORM ARCHITECTURE — MICROSERVICES DEPLOYMENT              │
│                                                                      │
│ GIT REPOSITORY                                                       │
│ ├── inventories/dev|staging|prod/  (environment-separated)          │
│ ├── roles/                                                           │
│ │   ├── common/         (NTP, DNS, hardening, monitoring agent)     │
│ │   ├── docker/         (Install Docker + Compose)                  │
│ │   ├── deploy_service/ (Generic microservice deploy role)          │
│ │   └── nginx_lb/       (Configure NGINX as load balancer)          │
│ └── playbooks/                                                       │
│     ├── site.yml                                                     │
│     ├── deploy_api.yml                                               │
│     └── deploy_worker.yml                                            │
│                                                                      │
│ CI/CD (GitHub Actions)                                               │
│ PR opened → lint + molecule test                                     │
│ Merge to main → deploy to dev (auto)                                │
│ Manual trigger → deploy to staging                                   │
│ Approval gate → deploy to prod (serial: 1 → 3 → all)               │
│                                                                      │
│ AWX/TOWER                                                            │
│ ├── Team: backend-team → can run deploy_api.yml on staging/prod     │
│ ├── Team: ops-team → can run site.yml on all                        │
│ ├── Scheduled: common role every 4h (drift remediation)             │
│ └── Webhooks: GitHub push → auto-triggers dev deployment            │
│                                                                      │
│ SECRET MANAGEMENT                                                    │
│ ├── Dev: Ansible Vault (single shared password)                      │
│ ├── Staging/Prod: AWS Secrets Manager via lookup plugin             │
│ └── No secrets in git; vault-encrypted references only              │
│                                                                      │
│ DEPLOYMENT FLOW                                                      │
│ 1. common role (hardening, monitoring, docker)                       │
│ 2. nginx_lb role on LB servers                                       │
│ 3. deploy_service role:                                              │
│    - Pull image from ECR                                             │
│    - Drain from nginx upstream                                       │
│    - docker pull + docker stop + docker run                          │
│    - Health check until /health returns 200                          │
│    - Re-add to nginx upstream                                        │
│ 4. Run integration tests (delegate_to: test-runner)                 │
│ 5. Notify Slack with deployment summary                              │
└──────────────────────────────────────────────────────────────────────┘
```

`deploy_service` **role (generic, called per service):**

```yaml
# Calling the role for each service
- name: Deploy all microservices
  hosts: appservers
  serial: "33%"               # Rolling: 1/3 of servers at a time

  tasks:
    - name: Deploy API service
      ansible.builtin.include_role:
        name: deploy_service
      vars:
        service_name: api
        image: "123456789.dkr.ecr.us-east-1.amazonaws.com/api:{{ api_version }}"
        port: 8080
        health_path: /health
        env_vars:
          DATABASE_URL: "{{ vault_db_url }}"
          LOG_LEVEL: "{{ log_level }}"

    - name: Deploy worker service
      ansible.builtin.include_role:
        name: deploy_service
      vars:
        service_name: worker
        image: "123456789.dkr.ecr.us-east-1.amazonaws.com/worker:{{ worker_version }}"
        port: 0          # No HTTP (queue consumer)
        health_path: ""
```

**Key architectural decisions:**

- One generic `deploy_service` role handles all services (DRY)
- Inventory groups match service responsibilities (not just OS type)
- Separate vault passwords per environment
- AWX enforces RBAC — teams can only deploy their services to their environments
- `serial: 33%` ensures 2/3 of capacity always serving during rollout
- Health check before re-adding to load balancer = zero-downtime guarantee

# APPENDIX: Ansible Quick Reference

## Must-Know Module Index

| Category | Module | Use |
|---|---|---|
| Files | `ansible.builtin.file` | Create dirs, set permissions, symlinks |
| Files | `ansible.builtin.copy` | Copy local file to remote |
| Files | `ansible.builtin.template` | Jinja2 template → remote file |
| Files | `ansible.builtin.fetch` | Copy remote file to local |
| Files | `ansible.builtin.lineinfile` | Manage single lines in files |
| Files | `ansible.builtin.blockinfile` | Manage blocks of text in files |
| Files | `ansible.builtin.replace` | Regex-based file editing |
| Files | `ansible.builtin.stat` | Check if file/dir exists, get metadata |
| Files | `ansible.builtin.find` | Find files matching criteria |
| Packages | `ansible.builtin.package` | Generic package manager |
| Packages | `ansible.builtin.apt` | Debian/Ubuntu packages |
| Packages | `ansible.builtin.dnf/yum` | RedHat packages |
| Services | `ansible.builtin.service` | Start/stop/enable services |
| Services | `ansible.builtin.systemd` | Full systemd management |
| Commands | `ansible.builtin.command` | Run command (no shell) |
| Commands | `ansible.builtin.shell` | Run command with shell features |
| Commands | `ansible.builtin.script` | Run local script on remote |
| Commands | `ansible.builtin.raw` | Direct SSH exec (no Python) |
| Users | `ansible.builtin.user` | Manage Unix users |
| Users | `ansible.builtin.group` | Manage Unix groups |
| Users | `ansible.builtin.authorized_key` | Manage SSH authorized_keys |
| Network | `ansible.builtin.uri` | HTTP requests |
| Network | `ansible.builtin.get_url` | Download files via HTTP |
| Network | `ansible.builtin.wait_for` | Wait for port/file/condition |

| Category | Module | Use |
| --- | --- | --- |
| System | `ansible.builtin.cron` | Manage cron jobs |
| System | `ansible.posix.sysctl` | Kernel parameters |
| System | `ansible.builtin.mount` | Manage filesystems |
| System | `ansible.builtin.reboot` | Reboot and wait |
| Flow | `ansible.builtin.debug` | Print debug messages |
| Flow | `ansible.builtin.set_fact` | Set runtime variable |
| Flow | `ansible.builtin.fail` | Explicitly fail with message |
| Flow | `ansible.builtin.assert` | Assert condition is true |
| Flow | `ansible.builtin.pause` | Wait or prompt user |
| Flow | `ansible.builtin.meta` | Ansible meta actions (flush_handlers) |
| Crypto | `ansible.builtin.openssl_*` | Certificate/key management |
| AWS | `amazon.aws.ec2_instance` | EC2 management |
| AWS | `amazon.aws.s3_object` | S3 operations |
| AWS | `amazon.aws.route53` | DNS management |
| K8s | `kubernetes.core.k8s` | Apply K8s manifests |
| Docker | `community.docker.docker_container` | Manage containers |

---

*End of Ansible Section — 50 questions, zero theory-only answers.*
*Every answer is grounded in what you'll encounter running Ansible against real infrastructure at scale.*