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.*

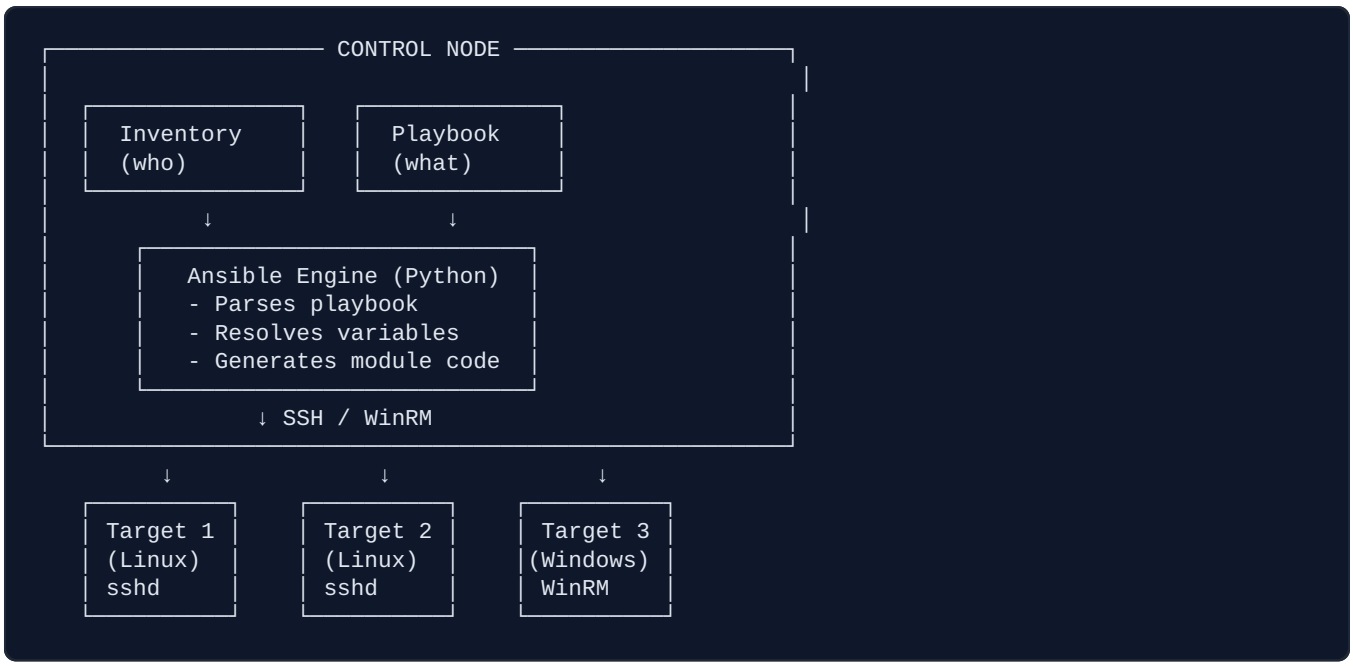
SECTION A: Architecture & Core Concepts

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):

```yaml
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):**

```yaml
- 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):**

```yaml
- 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:**

```yaml
- 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:

```yaml
# 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.

```yaml
- 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):

```yaml
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:

```yaml
- 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:

```yaml
# 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:**

```yaml
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:

```yaml
- 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:

```yaml
# 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:

```yaml
- 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:

```yaml
- 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:

```bash
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`:

```yaml
- 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.

```yaml
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:

```yaml
- 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):

```yaml
- 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`:

```yaml
- 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`:

```yaml
# 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`:

```yaml
# 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:

```yaml
- 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

```yaml
- 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.

```bash
# 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.

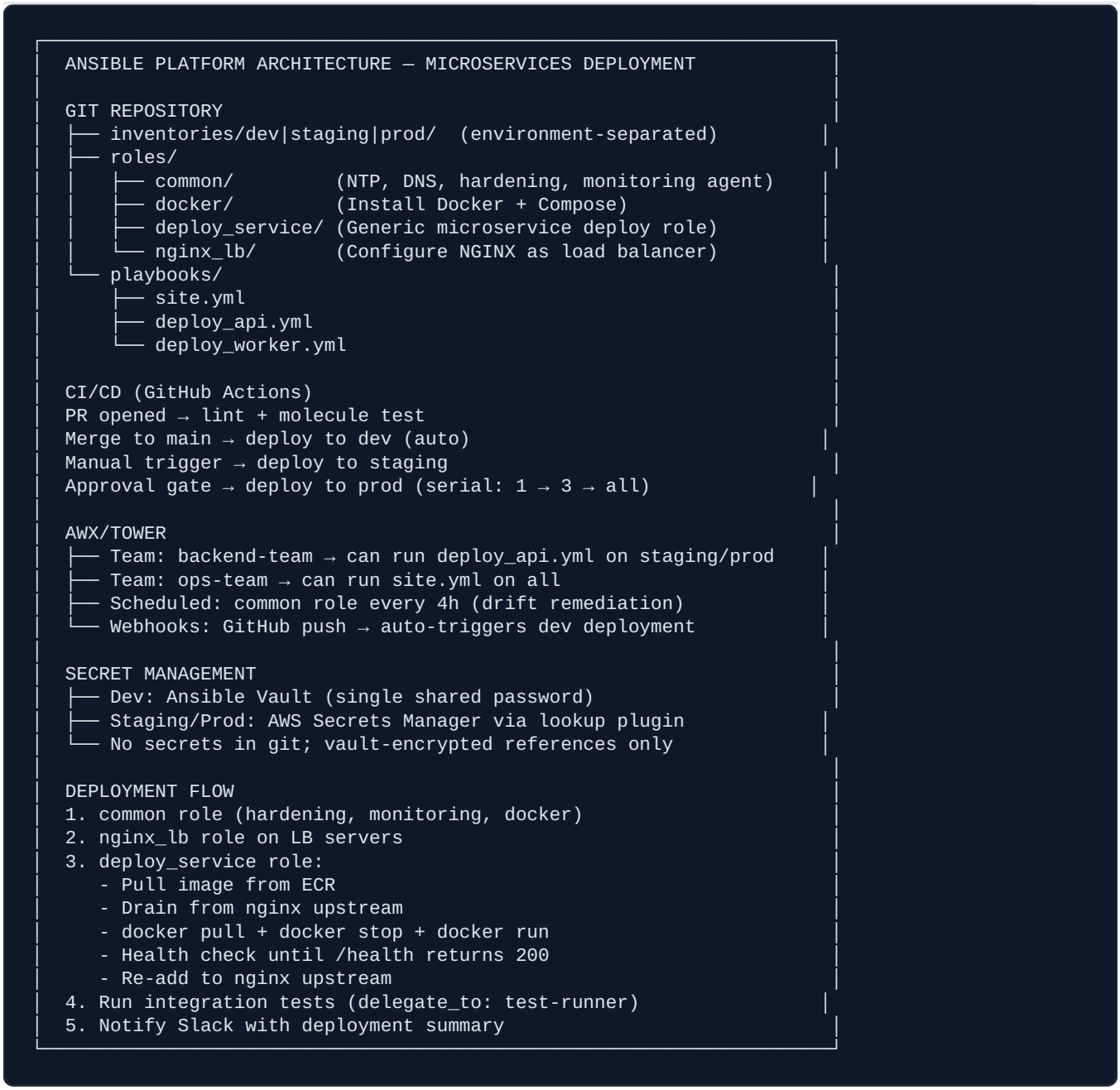
Key hardening tasks:

```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:



`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.