Ansible & Jenkins Interview – Q&A; Outline (2025)

1. Basics – Ansible & Configuration Management

1.1. What is Ansible? Where does it fit in DevOps?

• Ansible is an IT automation and configuration management tool.

• Terraform / CloudFormation → provision infrastructure (EC2, VPC, LB, RDS, etc.).

• Ansible → post-provisioning tasks: package installs, configuration, deployments, patching, hardening.

Notes from my project / experience:

1.2. What is the difference between IaC (Terraform) and configuration management (Ansible)?

• IaC: focuses on provisioning cloud resources and managing their lifecycle (create/update/destroy).

• Configuration management: configures and manages servers after they exist (apps, services, configs).

• Common pattern: Terraform to provision, Ansible to configure and deploy.

Notes from my project / experience:

1.3. Why is Ansible a leader vs Puppet / Chef / SaltStack?

• Agentless: uses SSH/WinRM, no agent installation on target nodes.

• Idempotent modules: safe to rerun playbooks multiple times.

• Simple, human-readable YAML syntax.

• Rich ecosystem: Ansible Galaxy, collections for common tools and cloud providers.

Notes from my project / experience:

2. Ad-hoc Commands, Playbooks & Roles

2.1. What is the difference between ad-hoc commands and playbooks?

• Ad-hoc: one-off, quick operations (e.g., ping, uptime, restart a service).

• Playbooks: reusable YAML files with multiple tasks, handlers, variables, etc.

• Use ad-hoc for quick checks; playbooks for repeatable deployments and configuration.

Notes from my project / experience:

2.2. What is the difference between playbooks and roles? Why use roles?

• Playbooks can become large and hard to manage.

• Roles provide modular structure: tasks, handlers, templates, vars, files, defaults.

• Roles improve reusability, abstraction and team sharing (via Git or Galaxy).

Notes from my project / experience:

3. Inventory – Static vs Dynamic

3.1. What is an inventory in Ansible and what types exist?

• Inventory: list of managed hosts and groups where Ansible runs tasks.

• Static inventory: manually defined in INI or YAML files.

• Dynamic inventory: built automatically using plugins for AWS, Azure, GCP, etc., usually filtered by tags.

Notes from my project / experience:

4. Idempotency, Handlers, Conditions & Loops

4.1. What is idempotency in Ansible and why is it important?

• Idempotency: rerunning a playbook doesn’t cause unintended changes when state already matches.

• Example: package with state=present shows 'ok' if already installed.

• Important for safe reruns and stable production deployments.

Notes from my project / experience:

4.2. What are handlers and notify in Ansible?

• Handlers are special tasks (usually service actions) triggered only when notified.

• Tasks use 'notify' to call handlers, typically after a change (e.g., config file updated → restart service).

• Handlers run at the end of the play and only if at least one notifying task changed.

Notes from my project / experience:

4.3. How do you use conditions and loops? Give examples.

• Conditions with 'when' for OS-specific or variable-based logic (e.g., ansible_os_family).

• Loops to avoid repetitive tasks: install multiple packages, create many users, manage multiple services.

• Can combine loops with conditions and use until/retries/delay for loop-until-success patterns.

Notes from my project / experience:

5. Facts & Custom Facts

5.1. What are facts in Ansible and how are they collected?

• Facts are system information (OS, IPs, CPU, memory, etc.) gathered from remote hosts.

• Collected by the 'setup' module; enabled by default with gather_facts: yes.

• Can disable gather_facts: no for speed when facts are not needed.

Notes from my project / experience:

5.2. What are custom facts and where are they stored?

• Custom facts are user-defined data about a host (e.g., environment, business tags).

• Stored as INI/JSON under /etc/ansible/facts.d/ on the managed node.

• They are loaded by the setup module and available like normal facts.

Notes from my project / experience:

6. Error Handling & Control

6.1. How do you handle errors in Ansible playbooks?

• Use ignore_errors to continue on failure when appropriate.

• Use failed_when / changed_when for custom success/fail logic.

• Use block / rescue / always to group tasks, handle failures and perform rollback/cleanup.

• Use max_fail_percentage to stop a play when too many hosts fail.

Notes from my project / experience:

6.2. Scenario: Run a task only if the previous one fails. How?

• Register the result of the first task using 'register'.

• Use a second task with condition 'when: result is failed' or 'when: result.failed'.

• Optionally combine with ignore_errors on the first task to allow play to continue.

Notes from my project / experience:

7. delegate_to, local_action & run_once

7.1. What are delegate_to, local_action and run_once in Ansible?

• delegate_to: run a task on a specific host (e.g., LB, jump host, deploy master) instead of the current target.

• local_action: shortcut for delegate_to: localhost, useful for tasks on the control node.

• run_once: ensure a task runs only once even if multiple hosts match, often combined with delegate_to.

Notes from my project / experience:

8. Multi-Environment Setup (dev / QA / prod)

8.1. How do you reuse the same playbook for dev, QA and prod?

• Use inventory groups: [dev], [qa], [prod] with separate servers.

• Maintain environment-specific variables in group_vars/dev.yml, group_vars/qa.yml, group_vars/prod.yml.

• Same playbook, different values (e.g., DB URLs, endpoints, credentials) based on inventory group.

Notes from my project / experience:

9. Zero-Downtime Deployment

9.1. How do you achieve zero-downtime deployments with Ansible?

• Rolling deployments: use 'serial' to update a subset of hosts at a time, and max_fail_percentage to stop on too many failures.

• Example: serial: 10 means 10 hosts per batch; play moves to next batch after health checks.

• Blue-Green deployments: maintain blue (live) and green (new) environments; switch traffic via load balancer after validation.

• Always include health checks/readiness checks before switching traffic.

Notes from my project / experience:

10. Package Drift & Compliance

10.1. How do you handle package version drift across many servers?

• Detect drift: use Ansible to gather versions across all hosts (facts or dedicated tasks).

• Enforce specific versions in playbooks or via variables (e.g., nginx-1.x.y).

• Use OS tools like yum-versionlock/apt pinning to prevent accidental upgrades.

• Re-run verification to confirm consistent versions after remediation.

Notes from my project / experience:

11. Performance Optimization & Large Scale

11.1. How do you optimize Ansible performance for 200–500 servers?

• Increase 'forks' in ansible.cfg (e.g., 30–50 or more depending on controller capacity).

• Enable SSH pipelining and multiplexing to reduce connection overhead.

• Disable gather_facts when not needed and consider caching facts.

• Use strategy: free so hosts run independently instead of strictly linear execution.

• Use async + poll for long-running tasks and run_once + delegate_to for heavy operations (e.g., downloading artifacts once).

• Split large playbooks into roles to keep them maintainable and reusable.

Notes from my project / experience:

12. Troubleshooting & Verbose Logs

12.1. How do you debug a failing playbook or Jenkins stage using Ansible?

• Run ansible-playbook with -v / -vv / -vvv / -vvvv for increasing verbosity.

• Use the debug module to print variable values and decision points.

• Redirect output to log files for later analysis.

• Use --start-at-task to resume from a specific task after fixing an issue.

• Combine with block/rescue for better error handling and rollback.

Notes from my project / experience:

13. Secret Management

13.1. How do you manage sensitive data (passwords, keys) in Ansible?

• Use Ansible Vault to encrypt variables and files; never commit plain-text secrets.

• Store and access vault passwords securely (e.g., Jenkins credentials, environment variables).

• Integrate with external secret managers like HashiCorp Vault, AWS Secrets Manager, Azure Key Vault as needed.

• Pass secrets into playbooks via encrypted vars or dynamic lookup plugins.

Notes from my project / experience:

14. Common Challenges / Pitfalls at Scale

14.1. What challenges have you seen with Ansible at scale and how did you solve them?

• Large inventories and slow runs → tune forks, pipelining, SSH settings and strategy.

• Configuration and package drift → enforce desired state and run regular compliance checks.

• Secret management issues → standardise on Vault/secret manager and avoid hard-coded credentials.

• Idempotency issues → prefer idempotent modules instead of raw command/shell where possible.

• Inventory management in cloud → use dynamic inventory plugins with proper tagging.

• Need for visibility → ensure proper logging, verbose output and auditing.

Notes from my project / experience:

15. CI/CD with Jenkins, Terraform & Ansible

15.1. How do you integrate Ansible into a Jenkins CI/CD pipeline?

• CI: checkout code, build (e.g., Maven for Spring Boot), run unit tests (JUnit), static analysis (SonarQube), publish artifacts (Nexus/Artifactory).

• CD: use Terraform stages to provision/modify infrastructure (terraform init/plan/apply).

• CD: run Ansible playbooks to configure servers (Tomcat/Nginx/HAProxy, etc.) and deploy artifacts from Nexus.

• Implement rolling or blue-green deployments using Ansible plus a load balancer for zero-downtime releases.

• Optional: add automated smoke/regression tests (e.g., Selenium) after deployment.

Notes from my project / experience:

15.2. If a Jenkins pipeline with Ansible fails midway, how do you resume?

• Check Jenkins and Ansible logs to find the exact failing stage and task.

• Fix the underlying issue (playbook, infra, credentials, etc.).

• Re-run Ansible with --start-at-task to resume from the failed task instead of starting from scratch.

• Optionally parameterise the Jenkins job to pass the start-at-task or environment for controlled reruns.

Notes from my project / experience: