ChatGPT

Ansible Interview Guide

Ansible Overview and Architecture

- **What is Ansible?** Ansible is an open-source configuration management and orchestration tool written in Python [1]. It automates software provisioning, configuration management, and application deployment without requiring agents on managed hosts [1].
- **Agentless, Push Model:** Ansible uses an *agentless* architecture – it connects over SSH (or WinRM for Windows) to managed nodes, pushes small modules/scripts, executes them, and removes them [2] [3]. Because it's push-based, a central control node runs playbooks and commands. Ansible modules are generally idempotent, so tasks only make changes if needed [2].
- **Control Node vs Managed Node:** The **control node** is where you run Ansible (playbooks or ad-hoc commands); **managed nodes** are the target servers. The control node's SSH keys or credentials allow password-less logins to managed nodes.

Interview Qs:
- **Q:** What does "agentless" mean in Ansible? *A:* Ansible communicates directly via SSH/WinRM without installing client software on targets [2].
- **Q:** How does Ansible achieve idempotency? *A:* Most Ansible modules are idempotent: they check current state and only change resources if needed, ensuring repeated runs are safe [2].

Inventory (Static vs Dynamic)

- **Static Inventory:** By default, Ansible uses a simple INI or YAML inventory file listing hosts and groupings. For example, an INI file may define groups like `[web]` and `[db]` with hosts under each [4]. Variables can be set per group or host in `group_vars/` or `host_vars/` files [5].
- **Dynamic Inventory:** For cloud or large environments, Ansible supports dynamic inventory scripts or plugins. These are executables that query a source (AWS, Azure, etc.) and output JSON of hosts/groups. Dynamic inventory can feed hosts into Ansible on-the-fly.
- **Groupings & Variables:** Groups let you run plays on a set of hosts (e.g. `hosts: webservers`). Under `group_vars/` and `host_vars/`, place YAML files (named after groups or hosts) to define variables that automatically apply to those hosts [6] [7].
- **Example (Static INI):**

```
[web]
web1 ansible_host=192.168.0.10
web2 ansible_host=192.168.0.11

[db]
db1 ansible_host=db.example.com
```

And in `group_vars/web.yml`:

```
---
package_name: nginx
```

These variables can then be used in plays (e.g. `{{ package_name }}`) [6] [7].

Interview Qs:
- **Q:** What's the difference between static and dynamic inventory? *A:* Static inventory is a file you edit by hand, while dynamic inventory uses scripts/plugins to generate hosts and groups (e.g. from cloud APIs) at runtime.
- **Q:** How do group_vars and host_vars work? *A:* Ansible automatically loads any `group_vars/<group>.yml` or `host_vars/<host>.yml` files relative to your inventory or playbook. Variables in these files are applied to the matching hosts [5] [6].

Playbooks

- **YAML Structure:** Playbooks are YAML files describing one or more *plays*. Each play maps a group of hosts to a list of tasks. A play starts with a dash (`- name:`) and includes keys like `hosts:`, `tasks:`, and optionally `become:` for privilege escalation [8] [9].
- **Tasks and Modules:** Under each play's `tasks:`, you list tasks that call Ansible **modules** (e.g. `yum`, `service`, `copy`). Each task has a `name`, specifies the module and its arguments.
- **Multiple Plays:** A single playbook can contain multiple plays, enabling orchestration across different host groups in sequence [10]. For example, one play could update webservers and another play updates databases.
- **Example:** A playbook with two plays (one for web, one for DB):

```
- name: Update web servers
  hosts: webservers
  become: true
  tasks:
    - name: Ensure apache is latest
      ansible.builtin.yum:
        name: httpd
        state: latest

    - name: Copy web config
      ansible.builtin.template:
        src: www.conf.j2
        dest: /etc/httpd.conf

- name: Update DB servers
  hosts: databases
  become: true
  tasks:
    - name: Ensure postgresql is installed
      ansible.builtin.yum:
        name: postgresql
        state: present

    - name: Ensure postgresql is running
      ansible.builtin.service:
        name: postgresql
        state: started
```

This shows how each play has `hosts:` and a `tasks:` list. Notice use of modules (`yum`, `template`, `service`) in tasks [11].

Interview Qs:
- **Q:** What are plays and tasks in an Ansible playbook? *A:* A **play** assigns a set of tasks to run on specified hosts. A **task** is a single action (using a module).
- **Q:** How do you define privilege escalation in a playbook? *A:* Include `become: true` (and possibly `become_user: root`) in the play or task to use sudo.

Roles

- **Purpose:** Roles let you automatically structure and organize playbook content (tasks, handlers, files, templates, vars, etc.) into reusable units [12]. Roles encapsulate the components for a particular function (e.g. an "nginx" role).
- **Standard Structure:** A role has a fixed directory layout. For example:

```
roles/
  myrole/
    tasks/main.yml
    handlers/main.yml
    defaults/main.yml
    vars/main.yml
    files/
    templates/
    meta/main.yml
```

- `tasks/main.yml`: core tasks
- `handlers/main.yml`: handlers (e.g. restart service)
- `defaults/main.yml`: default variables (lowest precedence)
- `vars/main.yml`: other variables (higher precedence)
- `files/` and `templates/`: static and Jinja2 files to copy
- `meta/main.yml`: role metadata and dependencies [13].
- **Usage:** In a playbook, include a role by name using the `roles:` keyword. For example:

```
- hosts: web
  roles:
    - role: myrole
      vars:
        custom_var: value
```

  This runs the tasks and handlers from that role.
- **Best Practices:** Use roles for organizing large playbooks. Name tasks, use default vars to avoid undefined errors, and keep role responsibilities focused.

Interview Qs:
- **Q:** What directories are typically in an Ansible role? *A:* At minimum: `tasks/`, `handlers/`, `defaults/`, `vars/`, `files/`, `templates/`, and `meta/` [13].
- **Q:** Why use roles? *A:* Roles promote reuse and organization: encapsulating related tasks/files/vars makes playbooks cleaner and shareable.

Collections

- **Definition:** Ansible **collections** are a distribution format that bundles playbooks, roles, modules, and plugins into a single package [14]. Collections allow sharing and organizing Ansible content.
- **Installing Collections:** Use `ansible-galaxy` to install collections from Ansible Galaxy or Automation Hub. For example:

  ```
  ansible-galaxy collection install my_namespace.my_collection
  ```

  This installs the collection (by default to `~/.ansible/collections/ansible_collections/`) [15]. Use `--upgrade` to update.
- **Using in Playbooks:** Collections change how you reference modules. You can use the Fully Qualified Collection Name (FQCN) (e.g. `community.general.command`) or declare at play level with `collections:`. You may also specify in `ansible.cfg` or `requirements.yml`.

Interview Qs:
- **Q:** How do you install an Ansible collection? *A:* `ansible-galaxy collection install namespace.collection_name` [15].
- **Q:** What does a collection include? *A:* Collections can include roles, modules, plugins, playbooks, and documentation – essentially packaged Ansible content [14].

Common Modules

- **Package Managers:**
- **apt** – Manages APT packages (Debian/Ubuntu). Example:

  ```
  - name: Install nginx (Debian/Ubuntu)
    ansible.builtin.apt:
      name: nginx
      state: present
  ```

  (apt is idempotent; installs only if missing) [16].
- **yum/dnf** – Manages RPM packages (RHEL/CentOS). Example:

  ```
  - name: Install httpd (CentOS)
    ansible.builtin.yum:
      name: httpd
      state: latest
  ```

  (works on Python2; for Python3, use `ansible.builtin.dnf`) [17].
- **service** – Controls services. Example:

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

Manages services across init systems (systemd, SysV, etc.) [18] .

- **file** – Creates or removes files/dirs. Example:

```yaml
- name: Create /data directory
  ansible.builtin.file:
    path: /data
    state: directory
    owner: deploy
    mode: '0755'
```

- **copy** – Copies a file from control to target. Example:

```yaml
- name: Copy config
  ansible.builtin.copy:
    src: configs/app.conf
    dest: /etc/app.conf
    mode: '0644'
```

- **template** – Renders Jinja2 template on control and copies. Example:

```yaml
- name: Deploy nginx config
  ansible.builtin.template:
    src: nginx.conf.j2
    dest: /etc/nginx/nginx.conf
```

- **user** – Manages user accounts. Example:

```yaml
- name: Ensure deploy user exists
  ansible.builtin.user:
    name: deploy
    state: present
    groups: sudo
```

- **shell/command** – Runs shell commands. Example:

```yaml
- name: Run custom script
  ansible.builtin.shell: /usr/local/bin/setup.sh
```

Use `shell` if you need shell features; `command` if you don't need a shell.

- **debug** – Prints messages for debugging. Example:

```yaml
- name: Show variable
  ansible.builtin.debug:
    msg: "The value is {{ my_var }}"
```

Interview Qs:
- **Q:** How do you manage packages on Debian vs Red Hat? *A:* Use the `apt` module on Debian/Ubuntu [16] and `yum` (or `dnf`) on RHEL/CentOS [17].
- **Q:** What module would you use to ensure a file is absent? *A:* The `file` module with `state: absent` or `absent`.

Variables and Facts

- **Variable Types & Precedence:** Ansible supports many variable scopes. You can set variables in inventory (`vars`), playbooks (`vars:` or `vars_files:`), roles (`defaults/` or `vars/`), or via `set_fact` in a play [19] [6]. The precedence determines which value wins if duplicates exist.
- **Host and Group Vars:** As discussed, `group_vars/` and `host_vars/` automatically apply variables by group or host name [6]. For example, putting `db_host: db.example.com` in `group_vars/database.yml` makes `{{ db_host }}` available to plays targeting the `database` group.
- **Facts:** By default Ansible gathers facts about each host at the start of a play (unless `gather_facts: false`). Facts include OS info, network interfaces, etc., and are available under `ansible_facts` or top-level keys like `ansible_facts['distribution']` [7]. For example, `ansible_distribution` or `ansible_facts['hostname']` can be used in conditionals or templates.
- **set_fact:** You can create or modify variables during a play using `set_fact`. This sets host-level variables for the rest of the playbook run [20]. Example:

```
- set_fact:
    temp_path: "/tmp/{{ inventory_hostname }}"
```

  Now `{{ temp_path }}` is available later. Set `cacheable: yes` to save it across runs.

Interview Qs:
- **Q:** What are Ansible facts? *A:* Facts are data collected about each host (os, ip, etc.) by the `setup` module. They're stored in `ansible_facts` and accessible in playbooks [7].
- **Q:** How do `group_vars` and `host_vars` differ? *A:* `group_vars` files define variables for all hosts in a named group; `host_vars` files define for a specific host. Ansible loads them automatically based on inventory names [6].

Handlers and Notify

- **Definition:** Handlers are special tasks that run only when notified by another task. They're typically used for actions like restarting a service after a configuration change. Handlers are declared under a `handlers:` section (often at end of play or in a role). Tasks can trigger them using the `notify:` keyword [21] [22].
- **Execution:** Handlers run **after** all tasks in a play have completed (but before the play ends) and **only if** notified. Even if multiple tasks notify the same handler, it runs only once per play per host [21]. If you notify multiple handlers, they run in the order defined in `handlers:`. Handlers run towards the end of the play on successful tasks; they are skipped if the notifying task fails (unless `--force-handlers` is used).
- **Example:**

```
tasks:
  - name: Update config file
    ansible.builtin.template:
      src: app.conf.j2
      dest: /etc/app.conf
    notify: Restart App

handlers:
  - name: Restart App
    ansible.builtin.service:
      name: myapp
      state: restarted
```

Here, the “Restart App” handler will run only if the template task reports a change [22].
- **Best Practice:** Name your handlers clearly. Use `listen:` on a handler to allow multiple tasks to notify it by alias. Remember handlers ignore tags (handlers run if notified, regardless of tags on the tasks [23]).

Interview Qs:
- **Q:** When are handlers executed? *A:* Handlers run at the end of a play if at least one task notified them, and only once per play per host [21].
- **Q:** How do you trigger a handler? *A:* By adding `notify: Handler Name` to a task. If that task changes, the named handler is queued to run after all tasks finish.

Conditionals and Loops

- **Conditionals (`when`):** Use the `when:` keyword to run tasks only if a condition is true. Conditions are Python/Jinja expressions without `{{}}`. Example:

```
- name: Install postfix only on Ubuntu
  ansible.builtin.apt:
    name: postfix
    state: present
  when: ansible_facts['distribution'] == "Ubuntu"
```

This task runs only if the host’s OS is Ubuntu [24]. You can combine conditions with `and`/`or`.
- **Loops:** Use `loop:` to iterate over a list. For example:

```
- name: Install multiple packages
  ansible.builtin.yum:
    name: "{{ item }}"
    state: present
  loop:
    - git
    - wget
    - vim
```

This runs the yum task three times (once per item). Older syntax `with_items:` is similar but `loop` is preferred in modern Ansible [25]. You can loop over lists of items, dictionaries (with `loop: "{{ my_dict | dict2items }}"`), or even nested loops. Use `loop_control` to set `index_var` or label.

- **Until (Retries):** The `until:` keyword can rerun a task until a condition is met (useful for polling). Example:

```
- name: Wait for service to be up
  ansible.builtin.shell: curl -f http://localhost/
  register: result
  until: result.rc == 0
  retries: 5
  delay: 10
```

  This retries the command up to 5 times or until it succeeds.

Interview Qs:
- **Q:** How do you skip a task for certain hosts? *A:* Use `when:` with a condition. E.g. `when: ansible_os_family == "RedHat"`.
- **Q:** How do you loop over multiple users to create them? *A:* Use a loop:

```
- user:
    name: "{{ item }}"
    state: present
  loop:
    - alice
    - bob
```

Tags and ansible.cfg

- **Tags:** Tags let you run or skip specific tasks/plays. Add `tags: [one, two]` to tasks, plays, or even roles. Then run with `--tags` or `--skip-tags`. For example, `ansible-playbook site.yml --tags ntp` runs only tasks tagged `ntp` [26]. Tags help in testing or partial runs. Note that handlers ignore tags and always run if notified, regardless of tags on tasks [23].

```
- name: Install NTP
  ansible.builtin.yum:
    name: ntp
    state: present
  tags:
    - ntp
```

- **Special Tags:** `always` and `never` are reserved tags. Tasks tagged `never` will never run.
- **ansible.cfg:** The `ansible.cfg` file (in /etc/ansible/, ~/.ansible.cfg, or project directory) configures defaults. Common settings include `inventory = /path/to/hosts`, `roles_path`, `library` (for custom modules), `remote_user`, and `vault_password_file`. You can generate a full default config via `ansible-config init` [27]. The file controls behavior like retry files, forks, and timeouts.

- **Example ansible.cfg snippet:**

  ```
  [defaults]
  inventory = inventory/hosts
  roles_path = ./roles
  vault_password_file = ~/.vault_pass.txt
  ```

Interview Qs:
- **Q:** How do tags affect task execution? *A:* You run playbooks with `--tags TAG` or `--skip-tags TAG`. Only tasks with matching tags (or without skipped tags) will run [26].
- **Q:** Where does Ansible look for ansible.cfg? *A:* Ansible looks in (1) ANSIBLE_CONFIG env var, (2) current directory, (3) ~/.ansible.cfg, (4) /etc/ansible/ansible.cfg.

Idempotency

- **Definition:** Idempotency means running a task (or play) multiple times yields the same result as running it once. In Ansible, this means modules check current state vs desired state and change the system only if needed. For example, `state: present` on a package won't reinstall if it's already installed [2]. Idempotent modules allow safe re-runs.
- **Interview Significance:** Interviewers often ask about idempotency. Emphasize that *idempotent tasks ensure predictable, repeatable runs*. It's critical in production: playbooks won't do unwanted changes on second execution.

Interview Qs:
- **Q:** Why is idempotency important in Ansible? *A:* It ensures playbooks can run multiple times without causing unintended changes. It's the principle that desired state is ensured without duplication.
- **Q:** Give an example of idempotency. *A:* Using `ansible.builtin.user` to create a user will do nothing if the user already exists (idempotent behavior).

Error Handling (`ignore_errors`, `failed_when`, `block/rescue/always`)

- **ignore_errors:** Add `ignore_errors: yes` to a task to continue the play on failure of that task. For example:

  ```
  - name: This may fail but continue
    ansible.builtin.command: /bin/false
    ignore_errors: yes
  ```

  This suppresses the failure for that task (noting it as "failed" in output) and proceeds [28]. It does *not* catch syntax errors or unreachable hosts.
- **failed_when:** You can override what constitutes failure. By default, non-zero return codes fail. Use `failed_when:` to set your own condition. E.g.:

  ```
  - name: Check disk space
    ansible.builtin.command: df / | tail -n1 | awk '{print $5}'
  ```

```
  register: disk
  failed_when: disk.stdout[:-1]|int > 90
```

Here we mark failure if disk usage is over 90%.

- **Blocks, Rescue, Always:** Use `block:` to group tasks, and handle errors similarly to try/catch:

```
- block:
    - command: /bin/false
    - debug: msg="Won't get here"
  rescue:
    - debug: msg="A failure was caught"
  always:
    - debug: msg="This runs regardless"
```

  In this example, after the forced failure, the `rescue` section runs (printing the catch message). The `always` section runs no matter what [29]. If a rescue task succeeds, it clears the failure status.
- **Best Practices:** Use `block` with `rescue/always` to gracefully handle known-error scenarios. Use `ignore_errors` sparingly. Prefer `failed_when` for fine-grained control over what constitutes a failure.

Interview Qs:
- **Q:** How can you make a task fail only under custom conditions? *A:* Use `failed_when:` with a boolean expression referencing the task's registered result.
- **Q:** How do you ensure cleanup tasks run even if earlier tasks failed? *A:* Use a `block` with an `always:` section; tasks there run regardless of failures [29].

Ansible Vault

- **Purpose:** Ansible Vault encrypts sensitive data (passwords, keys, etc.) so it isn't stored in plaintext. It integrates encrypted files into playbooks transparently [30].
- **Basic Usage (CLI):** Use the `ansible-vault` command-line tool. Common commands:
- `ansible-vault create secret.yml` – creates a new encrypted file (prompts for a password) [31].
- `ansible-vault edit secret.yml` – opens an encrypted file in editor for editing.
- `ansible-vault encrypt existing.yml` – encrypts an existing file in-place (asks for password) [32].
- `ansible-vault decrypt secret.yml` – decrypts file to plaintext.
- `ansible-vault view secret.yml` – displays decrypted content without editing.
- **Examples:**

```
# Create a new vault file
ansible-vault create vault.yml

# Encrypt an existing file
ansible-vault encrypt creds.yml
```

```
# In a playbook run with --ask-vault-pass or --vault-password-file
ansible-playbook site.yml --ask-vault-pass
```

Vault uses AES256 symmetric encryption. You can also encrypt single values in YAML with `!vault` tags (variable-level encryption). Use `--vault-id` or `--vault-password-file` options for non-interactive runs [33] [31].

- **Best Practice:** Do not include vault passwords in source control. Use separate vault files or IDs for dev/prod.

Interview Qs:
- **Q:** How do you encrypt a file with Ansible Vault? *A:* Use `ansible-vault encrypt filename.yml`. It will prompt for a password and encrypt the file [32].
- **Q:** How does Ansible decrypt vault files during a play? *A:* You supply the vault password via `--ask-vault-pass`, `--vault-password-file`, or `--vault-id`, and Ansible automatically decrypts vault-encrypted files at runtime [33].

Jinja2 Templating (with `template` Module)

- **Overview:** Ansible uses Jinja2 for templates. Templates allow dynamic content based on variables/facts. Use the `template` module to copy a `.j2` file to the target after rendering. All rendering happens on the control node [34].
- **Syntax:** Within templates, use `{{ }}` to insert variables and expressions, and `{% %}` for logic. You can apply filters (e.g. `{{ my_var | upper }}`) and tests.
- **Example:**

```
- name: Write hostname to file
  ansible.builtin.template:
    src: templates/test.j2
    dest: /tmp/hostname
```

And the template `templates/test.j2` could be:

```
My name is {{ ansible_facts['hostname'] }}
```

After the play, `/tmp/hostname` on the target will contain, for example, “My name is host1” [35].

- **Best Practices:** Keep logic minimal in templates. Use defaults for variables to avoid undefined errors. Remember that the template must be UTF-8.

Interview Qs:
- **Q:** How do you use a variable inside an Ansible template? *A:* Enclose it in `{{ }}`. Example: `{{ ansible_facts['hostname'] }}` in a `.j2` file will be replaced with the host’s name [35].
- **Q:** Can you use loops or conditionals in templates? *A:* Yes, Jinja2 supports loops (`{% for x in list %}`) and conditionals (`{% if %}`) inside templates.

Ad-hoc Commands vs Playbooks

- **Ad-hoc Commands:** Quick one-off commands from CLI without writing a playbook. Syntax: `ansible <pattern> -m <module> -a "<args>"`. Example:

```
ansible all -m ping
ansible web -m apt -a "name=nginx state=present"
```

Good for simple checks or operations on the fly.

- **Playbooks:** Structured, repeatable YAML files (as covered above). Use playbooks for any complex or multi-step automation. Playbooks are version-controllable and shareable.
- **When to Use:** Use ad-hoc for quick tasks (restart a service on a few hosts, check connectivity). Use playbooks for configuration management, deployments, and any repeatable procedure.

Interview Qs:
- **Q:** When would you use an ad-hoc command instead of a playbook? *A:* For simple, one-time tasks (e.g. checking SSH connectivity with `ping` module). For complex or repeatable tasks, write a playbook.
- **Q:** How does Ansible run ad-hoc tasks? *A:* It still uses SSH to connect and invokes a single module on target hosts, just without a playbook file.

Extras (Brief)

- **Ansible Galaxy:** A community hub for sharing Ansible roles and collections. Use `ansible-galaxy` CLI to install roles/collections (e.g. `ansible-galaxy role install geerlingguy.nginx`) or to create new roles. Galaxy also provides a web UI to search for roles.
- **Connection Plugins:** Ansible supports different connection types. By default it uses SSH (`ssh`) for Linux/Unix hosts. For Windows, it uses WinRM (`ansible.windows.winrm` and `win_*` modules) [3]. Others include `docker`, `kubectl`, or `local`.
- **Pull Mode:** Ansible can run in pull mode using `ansible-pull`. In this mode, a managed node periodically pulls a playbook from a VCS repo (e.g. git) and applies it. This inverts the push model and can scale to many nodes. Typical usage is via cron on each node:

```
ansible-pull -U https://github.com/example/ansible-repo.git -i
localhost, site.yml
```

The `ansible-pull` command automatically clones/updates the repo and runs the specified playbook locally [36]. This is less common but occasionally asked in interviews.

Interview Q:
- **Q:** What is `ansible-pull` and when would you use it? *A:* `ansible-pull` is a mode where nodes pull playbooks from a central repo (cron-driven), inverting the usual push model. Useful for large, auto-scaling environments [36].

Sources: Authoritative Ansible documentation and guides were used to ensure accuracy [1] [4] [11] [13] [14] [15] [16] [17] [18] [37] [7] [20] [22] [24] [26] [27] [29] [33] [31] [35] [36].

---

[1] [2] [4] [5] Ansible Collaborative - How Ansible Works
https://www.redhat.com/en/ansible-collaborative/how-ansible-works

[3] [18] ansible.builtin.service module – Manage services — Ansible Community Documentation
https://docs.ansible.com/ansible/latest/collections/ansible/builtin/service_module.html

[6] [7] [37] Ansible Playbook variables and facts | Get started with Ansible Playbooks | Red Hat Developer
https://developers.redhat.com/learning/learn:ansible:get-started-ansible-playbooks/resource/resources:ansible-playbook-variables-and-facts

[8] [9] [10] [11] Ansible playbooks — Ansible Community Documentation
https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_intro.html

[12] [13] Roles — Ansible Community Documentation
https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_reuse_roles.html

[14] Using Ansible collections — Ansible Community Documentation
https://docs.ansible.com/ansible/latest/collections_guide/index.html

[15] Installing collections — Ansible Community Documentation
https://docs.ansible.com/ansible/latest/collections_guide/collections_installing.html

[16] ansible.builtin.apt module – Manages apt-packages — Ansible Community Documentation
https://docs.ansible.com/ansible/latest/collections/ansible/builtin/apt_module.html

[17] ansible.builtin.yum module – Manages packages with the yum package manager — Ansible Community Documentation
https://docs.ansible.com/ansible/9/collections/ansible/builtin/yum_module.html

[19] Using variables — Ansible Community Documentation
https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_variables.html

[20] ansible.builtin.set_fact module – Set host variable(s) and fact(s). — Ansible Community Documentation
https://docs.ansible.com/ansible/latest/collections/ansible/builtin/set_fact_module.html

[21] [22] Handlers in Ansible Playbooks: How to Use Them
https://spacelift.io/blog/ansible-handlers

[23] [26] Tags — Ansible Community Documentation
https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_tags.html

[24] Conditionals — Ansible Community Documentation
https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_conditionals.html

[25] Loops — Ansible Community Documentation
https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_loops.html

[27] Configuring Ansible — Ansible Community Documentation
https://docs.ansible.com/ansible/latest/installation_guide/intro_configuration.html

[28] Error handling in playbooks — Ansible Community Documentation
https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_error_handling.html

[29] Blocks — Ansible Community Documentation
https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_blocks.html

[30] [33] Ansible Vault — Ansible Documentation
https://docs.ansible.com/ansible/2.9/user_guide/vault.html

[31] [32] How To Use Ansible Vault to Protect Sensitive Playbook Data | DigitalOcean
https://www.digitalocean.com/community/tutorials/how-to-use-vault-to-protect-sensitive-ansible-data

[34] [35] Templating (Jinja2) — Ansible Community Documentation
https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_templating.html

36 ansible-pull — Ansible Community Documentation
https://docs.ansible.com/ansible/latest/cli/ansible-pull.html