

Troubleshooting Common Ansible Playbook Execution Issues

Ansible is a powerful automation tool, but users may encounter various issues while working with it. Here are some common issues and their resolutions: Ansible is an automation tool that users may find challenging to work with. This document outlines common issues and their solutions.

1. Failed to Import Required Python Library (botocore or boto3)

Description:

This error occurs when Ansible tries to run AWS-related modules (such as managing EC2, AMIs, or S3) but the required Python libraries `boto3` and `botocore` are missing from the Ansible Controller environment. It is usually related to the Python configuration on the control node, not the playbook itself.

Symptoms:

- Fatal error when executing AWS modules:

None

```
fatal: [localhost]: FAILED! => {"changed": false, "msg": "Failed to import the required Python library (botocore or boto3) on demo.example.com's Python /usr/bin/python3.8."}
```

- Playbook stops with failure count (`failed=1`) in the recap.
- Libraries missing when checking with pip.

Resolution:

1. Check which Python version Ansible is using:

Shell

```
ansible --version
python3 --version
python3.8 --version
```

2. Confirm that pip is installed for the correct Python version (e.g., `pip3.8` for Python 3.8).

3. Use pip to install the required libraries:

Shell

```
pip3.8 install --user boto3 botocore
```

4.
(Replace `pip3.8` with `pip3.9` or another version depending on your Python environment.)

5. Verify installation:

Shell

```
pip3.8 list | grep boto
```

6.
Expected output should show both `boto3` and `botocore`.
7. Re-run your playbook to confirm successful execution.

Code:

Shell

```
# Check Python version used by Ansible
ansible --version

# Install boto3 and botocore for Python 3.8
pip3.8 install --user boto3 botocore

# Verify installation
pip3.8 list | grep boto

# Test playbook again
ansible-playbook ami_search.yml
```

2. Attempting to Decrypt but No Vault Secrets Found

Description:

This error occurs when Ansible tries to read an encrypted file created with **Ansible Vault** (e.g., containing sensitive data like passwords or keys), but no vault password is provided or the wrong password is used. Ansible Vault requires a password (or vault ID) to decrypt the file during playbook execution.

Symptoms:

- Playbook fails when including or referencing a vault-encrypted file.
- Example error:

None

```
fatal: [localhost]: FAILED! => {"message": "Attempting to decrypt but no vault secrets found"}
```

- Execution stops at the task where the encrypted file is accessed.

Resolution:

1. Provide the vault password when running the playbook:
 - Use `--ask-vault-password` to be prompted interactively.
 - Or use `--vault-password-file` to supply a file with the password.
2. Ensure you are using the **correct vault password** for the encrypted file.
3. If using multiple vault IDs, specify the correct vault ID with `--vault-id`.

Code:

Shell

```
# Run playbook and get prompted for vault password
ansible-playbook -i inventory --ask-vault-password
playbook_with_vault.yml

# Alternative: provide password file
```

```
ansible-playbook -i inventory --vault-password-file ~/.vault_pass.txt
playbook_with_vault.yml
```

Example Playbook Using Vault:

```
None
# playbook_with_vault.yml
---
- name: Playbook with Vault
  hosts: all
  tasks:
    - name: include vault
      ansible.builtin.include_vars:
        file: mypassword.yml

    - name: print variable
      ansible.builtin.debug:
        var: mypassword
```

Encrypted Variable File (mypassword.yml):

```
None
$ANSIBLE_VAULT;1.1;AES256
64306633373430303333362313636383363...
```

3. Destination Does Not Exist (rc 257)

Description:

This error occurs when Ansible attempts to modify a file (e.g., using `lineinfile` or similar modules) that does not exist on the target system. It usually happens because of a typo in the file path or because the file hasn't been created yet. A common scenario is editing SSH configuration (`/etc/ssh/sshd_config`), but referencing the wrong filename.

Symptoms:

- Fatal error with return code **257** during playbook execution.

- Example error message:

None

```
fatal: [demo.example.com]: FAILED! => {"msg": "Destination /etc/ssh/sshd_config2 does not exist !", "rc": 257}
```

- Playbook fails without making any changes.

Resolution:

1. Verify the correct file path on the target machine.

Shell

```
ls -l /etc/ssh/sshd_config
```

2. Fix typos or incorrect references in your playbook.
3. If the file doesn't exist but should, create it before using modules like `lineinfile`.

Code (Incorrect Playbook Example):

None

```
- name: PasswordAuthentication enabled
  hosts: all
  become: true
  gather_facts: false
  tasks:
    - name: ssh PasswordAuthentication
      ansible.builtin.lineinfile:
        dest: /etc/ssh/sshd_config2  # <-- Incorrect filename
        regexp: '^PasswordAuthentication'
        line: "PasswordAuthentication yes"
        state: present
        notify: ssh restart
```

```
handlers:
  - name: ssh restart
    ansible.builtin.service:
      name: sshd
      state: restarted
```

Code (Fixed Playbook):

```
None
- name: PasswordAuthentication enabled
hosts: all
become: true
gather_facts: false
tasks:
  - name: ssh PasswordAuthentication
    ansible.builtin.lineinfile:
      dest: /etc/ssh/sshd_config # <-- Correct path
      regexp: '^PasswordAuthentication'
      line: "PasswordAuthentication yes"
      state: present
      notify: ssh restart

handlers:
  - name: ssh restart
    ansible.builtin.service:
      name: sshd
      state: restarted
```

Verification:

```
Shell
# Check updated SSH configuration
grep 'PasswordAuthentication yes' /etc/ssh/sshd_config

# Verify SSH service is running
systemctl status sshd
```

4. Error 102: No Jinja2 in **when** Conditions

Description:

when clauses are always templated by Ansible. If you wrap the expression with `{{ ... }}` or `{% ... %}`, Ansible-Lint flags **Error 102: no-jinja-when** and your logic may not evaluate as intended. **when** must contain a raw Jinja2 expression (without curly braces or blocks).

Symptoms:

- Ansible-Lint reports:

None

```
no-jinja-when: No Jinja2 in when.
```

- Optional warning about spacing:



None

```
jinja[spacing]: Jinja2 spacing could be improved: {{ production }} ->
production
```

- Task shows a **when** like `when: "{{ production }}"`.

Resolution:

1. **Use raw expressions** in **when** (no `{{ ... }}` or `{% ... %}`):

-  `when: production`
-  `when: "{{ production }}"`

2. **Compare values directly** with operators (`==`, `in`, etc.):

- `when: var_service == 'httpd'`

3. **For dynamic variable names**, use the `vars` lookup:

- `when: lookup('vars', 'batches_' ~ var_function | regex_replace('^httpd-batch-', ''))`

4. **Combine multiple conditions** using a YAML list (logical AND):

None

```
when:

  - var_service == 'httpd'

  - item.path | regex_replace(cron_regex_for_s2_deletion, '\1')

    not in lookup('vars', 'batches_' ~ var_function |
regex_replace('^httpd-batch-', ''))
```

- 5.

Use YAML multiline (>-) to keep long expressions readable and avoid lint warnings.

Code (Bad → Good):

None

```
# ❌ Bad: Curly braces in when

- name: Ensure a task runs only in the production environment

  ansible.builtin.debug:

    msg: "This is a production task"

    when: "{{ production }}"
```

None

```
# ✅ Good: Raw expression

- name: Ensure a task runs only in the production environment

  ansible.builtin.debug:

    msg: "This is a production task"
```



```
when: production
```

None

```
#  Direct comparison
```


```
- name: Run only for httpd service
```

```
ansible.builtin.debug:
```

```
  msg: "Service is httpd"
```

```
  when: var_service == 'httpd'
```

None

```
#  Dynamic variable name with vars lookup
```


```
- name: Use dynamically named variable in condition
```

```
ansible.builtin.debug:
```

```
  msg: "Dynamic batches present"
```

```
  when: lookup('vars', 'batches_' ~ var_function |  
    regex_replace('^httpd-batch-', ''))
```

None

```
#  Multiple conditions (logical AND) with multiline trick
```

```
- name: Safe deletion filter
```

```
ansible.builtin.debug:
```

```

    msg: "Deleting old cron entry"

when:

    - var_service == 'httpd'

    - >-

    item.path | regex_replace(cron_regex_for_s2_deletion, '\\1')

    not in lookup('vars', 'batches_' ~ var_function |
regex_replace('^httpd-batch-', ''))

```

5. Error 104: Deprecated Bare Vars

Description:

Ansible-Lint Error 104 (`deprecated-bare-vars`) warns when variables are referenced ambiguously without clear syntax, making it unclear whether the expression should be interpreted as a **variable** or a **string**. This commonly occurs in loops (e.g., `with_items: foo`). Ansible requires either explicit variable syntax (`{{ foo }}`) or clear list/string usage.

Symptoms:

- Ansible-Lint reports:

```

None

deprecated-bare-vars: Possible bare variable 'foo' used in a
'with_items' loop.
You should use the full variable syntax ('{{ foo }}') or convert it to a
list if that is not really a variable.

```

- Schema validation warnings, e.g.:

None

```
schema[playbook]: ${0}.tasks[0].with_items 'production' does not match...
```

- Playbook execution may stop because this rule is **not skippable**.

Resolution:

1. If it's a string (not a variable):

- Provide it explicitly as a list of strings.

None

```
- ansible.builtin.debug:
  msg: "{{ item }}"
  with_items:
    - foo
```

2. If it's a variable:

- Use the full Jinja2 syntax with `{{ ... }}`.

None

```
- ansible.builtin.debug:
  msg: "{{ item }}"
  with_items: "{{ foo }}"
```

3. General Best Practices:

- Always make the intent explicit.
- Avoid shorthand that could confuse team members or break linting.
- Treat this rule as a code clarity enforcer for **readability, maintainability, and debugging**.

Code (Bad → Good):

None

```
# ❌ Bad: Bare var (ambiguous)
- ansible.builtin.debug:
    msg: "{{ item }}"
  with_items: foo
```

None

```
# ✅ Good: foo is a string
- ansible.builtin.debug:
    msg: "{{ item }}"
  with_items:
    - foo
```

None

```
# ✅ Good: foo is a variable
- ansible.builtin.debug:
    msg: "{{ item }}"
  with_items: "{{ foo }}"
```

6. Error 105: Deprecated Module Usage

Description:

Ansible-Lint Error 105 warns when your playbook uses **deprecated modules**—modules that are no longer actively maintained and are scheduled for removal in future releases. Continuing to use them introduces **security risks**, **compatibility problems**, and **technical debt**.

Symptoms:

- Playbook execution fails or produces warnings about unresolved/deprecated modules.
- Example error message:

None

```
syntax-check[specific]: couldn't resolve module/action
'ansible.netcommon.net_vlan'.
```

This often indicates a misspelling, missing collection, or incorrect module path.

- Linter output shows rule violation:

```
None
deprecated-module: Usage of deprecated module detected
```

Resolution:

1. **Identify deprecated modules** in your playbook using `ansible-lint`.
2. **Check the official Ansible documentation** or module index to find supported alternatives.
3. **Replace the deprecated module** with a maintained one (platform-specific or collection-based).
4. **Test the updated playbook** to confirm functionality.

Code (Bad → Good):

```
None
# ❌ Bad: Using a deprecated module

- name: Configure VLAN ID

  ansible.netcommon.net_vlan:

    vlan_id: 20
```

```
None
# ✅ Good: Using an actively maintained replacement





- name: Configure VLAN ID
```

```
dellemc.enterprise_sonic.sonic_vlans:
```

```
  config:
```

```
    - vlan_id: 20
```

Benefits of Replacing Deprecated Modules:

-  **Enhanced Security** — Maintained modules receive patches and updates.
-  **Long-Term Compatibility** — Ensures playbooks work with future Ansible releases.
-  **Community Support** — Access to documentation, bug fixes, and community help.
-  **Reduced Technical Debt** — Avoids future rewrites when deprecated modules are removed.

7. Error 106: Role Name Rules

Description:

Ansible-Lint Error 106 (**role-name**) enforces **naming conventions for roles** to ensure clarity, maintainability, and compatibility. Role names must follow specific rules: only lowercase alphanumeric characters and underscores are allowed, and names must begin with a letter. Non-compliant names can cause syntax errors, confusion, and difficulty when sharing or reusing roles.

Symptoms:

- Linter reports violations such as:

None


```
syntax-check[specific]: the role '1myrole' was not found...
```

- Playbook execution fails if role names are invalid.
- Common violations:
 - Role name starts with a number (**1myrole**).
 - Role name contains special characters (**myrole2[*^]**).



- Role name contains uppercase letters (`myRole_3`).

Resolution:



1. **Start role names with a lowercase letter.**

-  `myrole1`
-  `1myrole`

2. **Use only lowercase letters, digits, and underscores.**

-  `myrole_3`
-  `myrole2[*^`


3. **Avoid uppercase letters or special characters.**

-  `webserver_role`
-  `WebServerRole`


4. **Check role directories and names are consistent with these rules.**

Code (Bad → Good):

None

```
#  Bad: Non-compliant role names
- name: Example playbook
  hosts: localhost
  roles:
    - 1myrole      # starts with number
    - myrole2[*^   # contains special chars
    - myRole_3     # contains uppercase
```

None

```
#  Good: Compliant role names
- name: Example playbook
  hosts: localhost
```

```
roles:
  - myrole1      # starts with letter
  - myrole2      # lowercase alphanumeric
  - myrole_3     # lowercase with underscore
```

Benefits of Following Role Naming Rules:

- 📖 **Clarity & Consistency** — easier to read and understand roles across projects.
- 🛠️ **Maintainability** — simplifies updates and long-term code management.
- 🛡️ **Error Prevention** — avoids runtime and linting errors caused by invalid role names.
- 👥 **Collaboration** — fosters team and community best practices.

8. Error 202: Risky Octal Permissions

Description:

Ansible-Lint Error 202 (**risky-octal**) occurs when file permissions are defined as plain integers instead of properly formatted octal strings. Writing `mode: 644` without quotes and a leading zero causes YAML to treat the value as a decimal integer, which can lead to **unexpected and unsafe behavior**.

Symptoms:

- Linter flags violations such as:

```
None
risky-octal: `mode: 644` should have a string value with leading zero
`mode: "0644"` or use symbolic mode.
```

- Additional YAML formatting warnings may appear (trailing spaces, missing newline).
- Affected modules include:
 - `ansible.builtin.file`
 - `ansible.builtin.copy`

- `ansible.builtin.template`
- `ansible.builtin.assemble`
- `ansible.builtin.replace`

Resolution:

1. **Always quote file permission values.**
2. **Include a leading zero** (`"0644"`) or use the `0o` prefix (`"0o644"`) to ensure YAML interprets correctly.
3. **Alternatively, use symbolic modes** (e.g., `"u=rw,g=r,o=r"`).

Code (Bad → Good):

None

```
# ❌ Bad: Risky octal permissions
- name: Unsafe example of declaring numeric file permissions
  ansible.builtin.file:
    path: /etc/foo.conf
    owner: foo
    group: foo
    mode: 644    # Risky, interpreted as integer
```

None

```
# ✅ Good: Safe octal with leading zero
- name: Safe example of declaring numeric file permissions (1st
  solution)
  ansible.builtin.file:
    path: /etc/foo.conf
    owner: foo
    group: foo
    mode: "0644"    # Quoted with leading zero
```

None

```
# ✅ Good: Safe octal with 0o prefix
- name: Safe example of declaring numeric file permissions (2nd
solution)
  ansible.builtin.file:
    path: /etc/foo.conf
    owner: foo
    group: foo
    mode: "0o644"    # Using Python-style octal prefix
```

Benefits of Safe Permissions:

- 🗝️ **Predictable Behavior** — avoids YAML misinterpretation.
- 🔄 **Consistency** — ensures playbooks behave the same across environments.
- 📖 **Clarity** — makes your intentions explicit for collaborators.
- 🚫 **Avoids Surprises** — prevents hidden bugs caused by numeric parsing errors.

9. Error 203: No Tabs

Description:

Ansible-Lint Error 203 (**no-tabs**) occurs when **tab characters** (`\t`) are present in playbooks. Tabs cause inconsistent formatting across editors and platforms, leading to readability issues and potential parsing problems. Ansible strongly recommends using **spaces only** for indentation and alignment.

Symptoms:

- Linter flags violations such as:

None

```
no-tabs: Most files should not contain tabs.
203.yml:5 Task/Handler: Trigger the rule with a debug message
```

- Code may appear misaligned or inconsistent in different editors.
- Tabs inside a YAML string (e.g., `msg: "text \t text"`) may trigger this error — except when explicitly used with modules like `ansible.builtin.lineinfile`.

Resolution:

1. **Remove tab characters** from your playbooks.
2. **Replace tabs with spaces** (Ansible convention: **2 spaces per indentation level**).
3. **Check strings** for embedded `\t` characters and replace them with spaces.
4. **Use an editor setting** to automatically expand tabs into spaces.

Code (Bad → Good):

None

```
# ❌ Bad: Contains a tab character
- name: Example playbook
  hosts: all
  tasks:
    - name: Trigger the rule with a debug message
      ansible.builtin.debug:
        msg: "Using the \t character can cause formatting issues."
```

None

```
# ✅ Good: Tabs replaced with spaces
- name: Example playbook
  hosts: all
  tasks:
    - name: No tabs rule is triggered
      ansible.builtin.debug:
        msg: "Using space characters avoids formatting issues."
```

Benefits of Avoiding Tabs:

- 🛠️ **Consistent Formatting** — spaces ensure uniform indentation across environments.
- 👁️ **Readability** — playbooks remain clean and easy for teams to follow.
- 🛡️ **Reduced Errors** — avoids unexpected alignment or parsing problems.
- 🌐 **Cross-Platform Compatibility** — prevents display issues across different systems and editors.

Note:

Error 203 (`no-tabs`) does **not** trigger for `ansible.builtin.lineinfile` tasks where tabs may be part of the actual managed file content.

10. Error 205: `playbook-extension`

Description:

Ansible-Lint Error 205 (`playbook-extension`) ensures that playbooks are saved with the correct **YAML file extensions** (`.yaml` or `.yml`). Using the wrong extension (or none at all) can cause confusion, reduce readability, and in some cases prevent Ansible from correctly recognizing and running the file.

Symptoms:

- Playbook lacks `.yaml` or `.yml` extension.
- File is not immediately recognized as YAML, leading to:
 - Misidentification of the file type.
 - Execution issues in some environments.
 - Linter warnings or errors about extension compliance.

Resolution:

1. **Always save playbooks with `.yaml` or `.yml` extensions.**
2. **Rename incorrectly named files** to ensure consistency.

Shell

```
mv playbook playbook.yaml
```

3. **Use `.yaml` consistently** across your project for readability (though `.yml` is also valid).


Code (Bad → Good):

None





```
# ❌ Bad: Playbook without proper extension
- name: Example playbook without the proper extension
```

```
hosts: all
tasks:
  - name: Ensure some task is completed
    ansible.builtin.debug:
      msg: "This is a sample playbook"
```

None

```
#  Good: Playbook saved with .yaml extension
- name: Correctly named Ansible playbook
  hosts: all
  tasks:
    - name: Ensure a task is completed
      ansible.builtin.debug:
        msg: "This is a sample playbook"
```

Benefits of Using Correct Extensions:

-  **Recognition & Interpretation** – ensures files are parsed as YAML by Ansible.
-  **Standardization** – promotes consistent naming across automation projects.
-  **Clarity & Collaboration** – makes playbooks easier for teams to identify and use.
-  **Avoiding Errors** – prevents execution and management issues caused by misnamed files.

11. Error 206: Jinja Spacing

Description:

Ansible-Lint Error 206 (`jinja[spacing]`) enforces **proper spacing in Jinja2 templates** to improve readability and reduce the risk of typos. Variables, operators, and filters must be separated with spaces. This also applies to fields with **implicit templating** (like `when`, `changed_when`, `failed_when`, and `until`), where double curly braces `{{ }}` are not required.

Symptoms:

- Linter flags violations such as:

None

```
jinja[spacing]: Jinja2 spacing could be improved: {{some|dict2items}} ->
{{ some | dict2items }}
jinja[spacing]: Jinja2 spacing could be improved: {{ foo | bool }} ->
foo | bool
```





- Extra warning:

None

```
no-jinja-when: No Jinja2 in when.
```


- Playbook still runs, but readability suffers and lint checks fail.

Resolution:

1. **Add spaces** around variables, filters, and operators.
 -  `{{ some | dict2items }}`
 -  `{{some|dict2items}}`
2. **Use implicit templating** in `when` and similar clauses (no `{{ }}`).
 -  `when: foo | bool`
 -  `when: "{{ foo | bool }}"`
3. **Run automatic fixes** with `ansible-lint --fix` to reformat Jinja2 expressions.
4. **Be aware of limitations:** tilde (~) concatenation, dot notation with numbers, and newline formatting inside Jinja2 blocks may need manual cleanup.

Code (Bad → Good):

None

```
#  Bad: Improper spacing and redundant templating
- name: Example error 206
```

```

hosts: all
tasks:
  - name: Error 206
    ansible.builtin.debug:
      vars:
        foo: "{{some|dict2items}}"    # No spaces
        when: "{{ {foo | bool} }}"    # Redundant braces

```

```

None
# ✅ Good: Proper spacing and implicit templating
- name: Example error 206
  hosts: all
  tasks:
    - name: Error 206
      ansible.builtin.debug:
        vars:
          foo: "{{ { some | dict2items } }}"
          when: foo | bool

```

Benefits of Proper Jinja Spacing:

- 📖 **Improved Readability** — clearer playbooks for teams to read and maintain.
- 🛡️ **Reduced Typos** — spacing prevents accidental errors in templates.
- 🛠️ **Best Practices** — aligns with Ansible coding standards.
- ⚡ **Automatic Fixing** — can be corrected quickly with `ansible-lint --fix`.

12. Error 207: Jinja Invalid

Description:

Ansible-Lint Error 207 (`jinja[invalid]`) detects **invalid Jinja2 expressions** in playbooks—typically caused by malformed syntax within double curly braces (`{{ ... }}`). These invalid templates may pass basic syntax checks but cause runtime failures when Ansible processes them.

Symptoms:

- Linter outputs error messages such as:

None

```
jinja[invalid]: template error while templating string: unexpected char '&' at 3. String: {{ & }}. unexpected char '&' at 3
```

- Example problematic line:

None

```
bar: "{{ & }}"
```

- The playbook fails to run due to runtime template rendering errors, even though basic syntax may appear valid. ([ansiblepilot.com](https://www.ansiblepilot.com))

Resolution:

1. **Ensure valid Jinja2 syntax:** Only use supported characters and constructs within `{{ }}`.
2. **Wrap literal characters correctly:** If you intend to pass special characters like `&`, quote them inside the template:
 - From: `bar: "{{ & }}"`
 - To: `bar: "{{ ' & ' }}"`
3. **Validate templates manually or using `ansible-lint --fix`**, which can automatically address some invalid templates. ([ansiblepilot.com](https://www.ansiblepilot.com), [Ansible](https://docs.ansible.com/ansible/latest/user_guide/playbooks_templating.html))

Code (Bad → Good):

None

```
# ❌ Bad: Invalid Jinja expression with unsupported character
```



```
- name: Example error 207

hosts: all

tasks:

  - name: Error 207

    ansible.builtin.debug:

      vars:

        bar: "{{ & }}" # Invalid: unsupported character inside
template
```

None

☒ Good: Valid Jinja template with quoted special character

```
- name: Example error 207

hosts: all

tasks:

  - name: Error 207

    ansible.builtin.debug:

      vars:

        bar: "{{ '&' }}" # Valid: literal ampersand properly quoted
```

Benefits of Correcting Invalid Templates:

- Prevents unexpected **runtime errors**.
- Improves **template reliability and stability**.
- Enhances **playbook maintenance and readability**.

- Can be partly fixed automatically with lint tools. (ansiblepilot.com, [Ansible](#))

13. Error 208: Risky File Permissions

Description:

Ansible-Lint Error 208 (`risky-file-permissions`) warns when modules such as `copy`, `file`, `template`, or `ini_file` create or modify files without explicit permissions, potentially resulting in unsecured or unpredictable defaults. ([Ansible Pilot](#))

Symptoms:

- Linter outputs:

None

```
risky-file-permissions: File permissions unset or incorrect.
```

- Occurs in tasks using modules like `community.general.ini_file` without specifying `mode`, or using `create: true` without controlling permissions. ([Ansible Pilot](#))

Resolution:

1. **Prevent unintended file creation**, if not needed:

None

```
community.general.ini_file:
```

```
  path: foo
```

```
  create: false
```

- 2.

Explicitly set secure file modes:

None

```
community.general.ini_file:
```

```
path: foo
```

```
mode: 0600
```

3.

Preserve permissions when copying:

None

```
ansible.builtin.copy:
```

```
src: foo
```

```
dest: bar
```

```
mode: preserve
```

Code (Bad → Good):

None

```
# ❌ Risky: no mode, default permissions
```

```
community.general.ini_file:
```

```
path: foo
```

```
create: true
```

```
# ✅ Safe: prevent creation or set explicit mode
```

```
community.general.ini_file:
```

```
path: foo
```

```
create: false
```

```
# Or

community.general.ini_file:

    path: foo

    mode: 0600


# Or preserve when copying

ansible.builtin.copy:

    src: foo

    dest: bar

    mode: preserve
```

Benefits:

- Ensures **secure and predictable permissions**.
- Reduces risk of exposing sensitive data.
- Enhances **cross-system consistency** in behavior. ([Ansible Pilot](#))

14. Error 301: No **changed_when**

Description:

Error 301 (**no-changed-when**) flags tasks that may cause side-effects but do not explicitly declare when they should be marked as changed. This is especially relevant for command or shell tasks that rely on return codes to indicate change. ([Ansible Pilot](#))

Symptoms:

- Linter warns:

None

`no-changed-when: Commands should not change things if nothing needs doing.`

-

For example, a `command: cat {{ my_file }}` without logic to detect changes. ([Ansible Pilot](#))

Resolution:

- **Register output** and define `changed_when`, e.g.:

None

```
- ansible.builtin.command: cat {{ my_file | quote }}  
  
  register: my_output  
  
  changed_when: my_output.rc != 0
```

Code (Bad → Good):

None

```
# ❌ Task with no change detection  
  
ansible.builtin.command: cat {{ my_file | quote }}  
  
  
# ✅ Task with explicit change logic  
  
ansible.builtin.command: cat {{ my_file | quote }}  
  
  register: my_output  
  
  changed_when: my_output.rc != 0
```

Benefits:

- Enhances **accurate reporting** of changes.
- Improves playbook **idempotency** and diagnostics.
- Supports reliable automation and clarity. ([Ansible Pilot](#))

15. Error 302: Deprecated Command Syntax

Description:

Error 302 ([deprecated-command-syntax](#)) warns against using shorthand or free-form command syntax inside playbooks. Structured parameter blocks ([args:](#)) are preferred for clarity and maintainability. ([Ansible Pilot](#))

Symptoms:

- Linter flags shorthand like:

None

```
ansible.builtin.command: creates=B chmod 644 A
```

- Inelegant inline parameters stink of ambiguity. ([Ansible Pilot](#))

Resolution:

- Use structured argument format:

None

```
ansible.builtin.command: chmod 644 A
```

```
args:
```

```
  creates: B
```

Code (Bad → Good):

None

❌ Shorthand (deprecated)

```
ansible.builtin.command: creates=B chmod 644 A
```

✅ Structured syntax

```
ansible.builtin.command: chmod 644 A
```

```
args:
```

```
  creates: B
```

Benefits:

- Improves **readability** and reduces confusion.
- Enhances **troubleshooting and maintenance**.
- Adheres to evolving **best practices**. ([Ansible Pilot](#))