Troubleshooting Ansible Playbook Execution

Ansible is an effective automation tool, though users may encounter challenges. This document details common issues and their resolutions.

1. Error: name[prefix]

Description:
The `name[prefix]` rule is an optional Ansible-Lint rule that encourages **task name prefixes** in sub-task files (task files that are not named `main.yml`). The rule suggests prefixing each task name with the filename or logical context. For example, a task inside `tasks/deploy.yml` should be written as:

```
None
deploy | Restart server
```

This improves clarity in large projects with multiple task files by making it obvious where each task originates.

Symptoms:

- Linter reports violations like:

```
None
name[play]: All plays should be named.
name-prefix.yml:2
```

- Tasks without prefixes may cause confusion in complex roles.
- Playbooks still run, but lint checks fail, and troubleshooting task origins becomes harder.

Resolution:

- Always give **plays and tasks clear names**.
- When tasks are in a file other than `main.yml`, add a **prefix** to the task name.

    - Correct: `deploy | Restart server`
    - Incorrect: `Restart server`
- If you want to enforce this rule, enable it explicitly in your Ansible Lint configuration (`enable_list`).

Code

(Incorrect → Correct):

```
None
# Incorrect: No play name, task not prefixed
- hosts: all
  tasks:
    - name: Create placefolder file
      ansible.builtin.command: touch /tmp/.placeholder
```

```
None
# Correct: Play is named, task is clear
- name: Play for creating placeholder
  hosts: localhost
  tasks:
    - name: deploy | Create placeholder file
      ansible.builtin.command: touch /tmp/.placeholder
```

Benefits of Following name[prefix]:

- Consistency — standard naming across multi-file roles.
- Readability — makes it clear which file or context a task comes from.
- Maintainability — easier to manage and troubleshoot large playbooks.
- Collaboration — improves team understanding of complex playbook structures.

Note:

- The `name[prefix]` rule is **optional (opt-in)**. You must add it to your `enable_list` in `.ansible-lint` to enforce it.

- It is particularly useful in large roles with many sub-task files.

2. Error: name[template]

Description:
Ansible-Lint Error `name[template]` occurs when **Jinja2 templates are used incorrectly inside task names**. Task names are meant to be human-readable labels that describe the task's purpose. Using Jinja templates in the middle of a task name makes them less clear and harder to follow. The rule enforces a best practice: **if you must use a Jinja template in a task name, place it at the end**.

Symptoms:

- Linter flags violations such as:

```
None
name[template]: Jinja templates should only be at the end of
'name'
```

- Task names are unclear when read without rendering variables.

- Playbook readability and maintainability suffer.

Resolution:

- Move Jinja templates to the end of the task name.
    - Correct:

```
None
name: Create directory {{ id }}
```

    - Incorrect:

```
None
name: Create {{ id }} directory
```

- Use templates sparingly in task names to preserve readability.
- Test templates to ensure they render correctly before deploying.
- Simplify overly complex dynamic names — prefer clear static names whenever possible.

Code

(Incorrect → Correct):

```
None
# Incorrect: Template used in the middle of the task name
- name: Example playbook
  hosts: all
  vars:
    id: name
  tasks:
    - name: Create {{ id }} directory
      ansible.builtin.file:
        path: "/tmp/{{ id }}"
        state: directory
```

```
None
# Correct: Template placed at the end of the task name
```

```
- name: Example playbook
  hosts: all
  vars:
    id: name
  tasks:
    - name: Create directory {{ id }}
      ansible.builtin.file:
        path: "/tmp/{{ id }}"
        state: directory
```

Benefits of Following Rule name[template]:

- Readability — clearer task names make playbooks easier to follow.
- Maintainability — consistent naming conventions simplify future updates.
- Troubleshooting — static parts of task names remain visible even if templates fail.
- Best Practices — aligns with Ansible's standards for structured, human-readable playbooks.

3. Error: no-free-form

Description:
Ansible-Lint Error `no-free-form` occurs when **free-form (inline) syntax** is used instead of full dictionary syntax in module calls. Free-form syntax places arguments directly after the module name, which reduces readability, increases the risk of subtle bugs, and prevents editors/IDEs from providing useful features such as validation and autocompletion. The rule enforces **explicit argument syntax** to make playbooks more maintainable.

Symptoms:

- Linter flags violations such as:

```
None
no-free-form: Avoid using free-form when calling module actions.

no-free-form[raw]: Avoid embedding `executable=` inside raw
calls.
```

- Tasks may behave unexpectedly if arguments are misinterpreted.
- Editors and IDEs cannot provide proper feedback or autocompletion.

Resolution:

- Replace free-form syntax with full argument syntax.
    - Correct:

```
None
ansible.builtin.command:

  cmd: touch foo

  chdir: /tmp
```

    - Incorrect:

```
None
ansible.builtin.command: chdir=/tmp touch foo
```

- For the `raw` module, avoid embedding parameters inline. Use the `args` dictionary instead.
- Run `ansible-lint --fix` to automatically correct many free-form syntax issues.

Code

(Incorrect → Correct):

```
None
# Incorrect: Using free-form syntax
- name: Example with discouraged free-form syntax
  hosts: all
  tasks:
    - name: Create a placeholder file
      ansible.builtin.command: chdir=/tmp touch foo


    - name: Use raw to echo
      ansible.builtin.raw: executable=/bin/bash echo foo
      changed_when: false
```

```
None
# Correct: Using full dictionary syntax
- name: Example that avoids free-form syntax
  hosts: all
  tasks:
    - name: Create a placeholder file
      ansible.builtin.command:
        cmd: touch foo
        chdir: /tmp
```

```
- name: Use raw to echo
  ansible.builtin.raw: echo foo
  args:
    executable: /bin/bash
  changed_when: false
```

Benefits of Following Rule no-free-form:

- Clarity — tasks are easier to read and understand.
- Reliability — avoids bugs caused by ambiguous or misinterpreted free-form syntax.
- Tooling Support — enables IDEs to provide autocompletion, linting, and validation.
- Best Practices — aligns with Ansible's modern style guidelines.

4. Error: no-jinja-when

Description:

Ansible-Lint Error `no-jinja-when` is triggered when **Jinja2 expressions inside double curly brackets `{{ }}` are used in `when` conditions**. In Ansible, `when`, `failed_when`, and `changed_when` clauses already support implicit Jinja2 evaluation, so wrapping expressions in `{{ }}` is unnecessary and considered an anti-pattern. This rule enforces using facts and variables directly.

Symptoms:

- Linter reports violations such as:

```
None
no-jinja-when: No Jinja2 in when.
```

- Additional warnings may appear (for example, `jinja[spacing]` or `no-changed-when`).
- Example flagged task:

```
None
when: "{{ ansible_facts['os_family'] == 'Debian' }}"
```

Resolution:

- Remove unnecessary `{{ }}` from `when` clauses.
    - Correct:

```
None
when: ansible_facts['os_family'] == "Debian"
```

    - Incorrect:

```
None
when: "{{ ansible_facts['os_family'] == 'Debian' }}"
```

- Use variables and facts directly in conditions.
- Apply the same best practice for other implicit templating clauses (`failed_when`, `changed_when`, `until`).

Code

(Incorrect → Correct):

```
None
# Incorrect: Jinja expression wrapped in curly braces inside when
```

```
- name: Example playbook
  hosts: localhost
  tasks:
    - name: Shut down Debian systems
      ansible.builtin.command: /sbin/shutdown -t now
      when: "{{ ansible_facts['os_family'] == 'Debian' }}"
```

```
None
# Correct: Fact used directly in when clause
- name: Example playbook
  hosts: localhost
  tasks:
    - name: Shut down Debian systems
      ansible.builtin.command: /sbin/shutdown -t now
      when: ansible_facts['os_family'] == "Debian"
```

Why Avoid Nesting Jinja in when Clauses:

- Prevents unintended nested evaluations that may cause errors.
- Improves readability by making conditions clean and direct.
- Aligns with Ansible's implicit templating behavior.
- Helps ensure playbooks are easier to maintain and less error-prone.

Benefits of Following Rule no-jinja-when:

- Readability — conditions are easier to understand.
- Predictability — avoids confusing nested evaluations.
- Maintainability — consistent style across playbooks.
- Best Practices — aligns with Ansible standards for writing conditions.

5. Error: no-log-password

Description:
The `no-log-password` error is raised by Ansible-Lint when a task may expose **sensitive values such as passwords** in logs. This often happens when passwords are provided inside **loops**, which increases the risk of logging secrets during task execution. To mitigate this, Ansible requires the use of `no_log: true` to explicitly prevent logging sensitive data.

Symptoms:

- Linter reports violations like:

```
None
no-log-password: Task may log sensitive data. Use `no_log: true`
to protect secrets.
```

- Passwords or secrets appear in playbook execution logs.
- Security reviews flag playbooks as unsafe due to unprotected secrets.

Resolution:

- Add `no_log: true` to tasks handling passwords or other sensitive values.
- Apply `no_log` at the **task level**, especially when using `with_items` or `loop` to assign multiple secret values.
- Review tasks that handle sensitive data (e.g., credentials, tokens, private keys) to ensure they are not inadvertently logged.

Code

Incorrect → Correct):

```
None
# Incorrect: Passwords may appear in logs
- name: Example playbook
  hosts: all
  tasks:
    - name: Log user passwords
      ansible.builtin.user:
        name: john_doe
        comment: John Doe
        uid: 1040
        group: admin
        password: "{{ item }}"
      with_items:
        - secret123
        - another_secret
        - super_secret
```

```
None
# Correct: no_log prevents sensitive data from being exposed
- name: Example playbook
```

```
hosts: all

tasks:

  - name: Do not log user passwords

    ansible.builtin.user:

      name: john_doe

      comment: John Doe

      uid: 1040

      group: admin

      password: "{{ item }}"

    with_items:

      - secret123

      - another_secret

      - super_secret

    no_log: true
```

Benefits of Following the no-log-password Rule:

- Security — prevents exposure of passwords or sensitive data in logs.
- Compliance — aligns with best practices for secure automation workflows.
- Maintainability — makes it clear which tasks handle sensitive values.
- Peace of Mind — reduces the risk of accidentally leaking secrets during debugging or audits.

6. Error: no-prompting

Description:

Ansible-Lint Error `no-prompting` is triggered when a playbook contains **user prompts (`vars_prompt`) or pauses (`ansible.builtin.pause`)**. These practices are discouraged in automation because they require manual intervention, making playbooks unsuitable for **unattended execution** in environments such as CI/CD pipelines.

Symptoms:

- Linter flags violations such as:

```
None
no-prompting: Play uses vars_prompt
```

- Playbooks stall execution while waiting for user input (e.g., username, password).
- Pipeline jobs fail or hang due to unnecessary pauses (`pause` module).

Resolution:

- Replace `vars_prompt` with predefined variables or external secrets.
    - Use static `vars`, inventory variables, or credentials from a secure vault.
- Remove `ansible.builtin.pause` tasks unless absolutely necessary.
    - If needed, use **conditionals** or handlers instead of pauses to control workflow.
- Ensure playbooks run fully unattended, especially in automated workflows.

Code

(Incorrect → Correct):

```
None
# Incorrect: Contains vars_prompt and pause

- name: Example playbook
```

```
  hosts: all
  vars_prompt:
    - name: username
      prompt: What is your username?
      private: false
    - name: password
      prompt: What is your password?
  tasks:
    - name: Pause for 5 minutes
      ansible.builtin.pause:
        minutes: 5
    - name: Display message
      ansible.builtin.debug:
        msg: "{{ username }}, {{ password }}"
```

None

```
# Correct: No prompting or pausing
- name: Example playbook
  hosts: all
  vars:
    username: username
```

```
    password: password
  tasks:
    - name: Display message
      ansible.builtin.debug:
        msg: "{{ username }}, {{ password }}"
```

Benefits of Following Rule no-prompting:

- Automation-Friendly — ensures playbooks run unattended in pipelines.
- Reliability — eliminates the risk of stalled tasks due to missing user input.
- Efficiency — faster execution without unnecessary delays or pauses.
- Best Practices — aligns with CI/CD principles for reproducible, hands-free automation.

7. Error: no-same-owner

Description:
Ansible-Lint Error `no-same-owner` occurs when files are transferred or extracted **while preserving their original owner and group from the source system**. This practice can cause security risks, permission mismatches, or unintended access on the target host. The rule enforces disabling ownership transfer to ensure predictable and safe file operations.

Symptoms:

- Linter flags violations such as:

```
None
no-same-owner: Do not preserve the owner and group when
transferring files across hosts.
```

- Occurs when using modules like `ansible.posix.synchronize` or `ansible.builtin.unarchive` without disabling ownership preservation.

- Can result in permission errors or security concerns if files end up with unexpected owners/groups.

Resolution:

- For `ansible.posix.synchronize`:
    - Explicitly set `owner: false` and `group: false`.
- For `ansible.builtin.unarchive`:
    - Add the `--no-same-owner` option in `extra_opts`.
- Review all file transfer or archive extraction tasks to ensure they do not preserve source ownership unintentionally.

Code

(Incorrect → Correct):

```
None
# Incorrect: Synchronize without disabling owner/group transfer

- name: Synchronize conf file

  ansible.posix.synchronize:

    src: /path/conf.yaml

    dest: /path/conf.yaml
```

```
None
# Correct: Synchronize with no owner/group preservation

- name: Synchronize conf file

  ansible.posix.synchronize:
```

```
    src: /path/conf.yaml
    dest: /path/conf.yaml
    owner: false
    group: false
```

None

```
# Incorrect: Extract archive without --no-same-owner
- name: Extract tarball to path
  ansible.builtin.unarchive:
    src: "{{ file }}.tar.gz"
    dest: /my/path/
```

None

```
# Correct: Extract archive with --no-same-owner
- name: Extract tarball to path
  ansible.builtin.unarchive:
    src: "{{ file }}.tar.gz"
    dest: /my/path/
    extra_opts:
      - --no-same-owner
```

Benefits of Following no-same-owner Rule:

- Security — prevents unintended access caused by incorrect file owners/groups.
- Predictability — files adopt the correct ownership context on the target system.
- Consistency — ensures uniform handling of files across different environments.
- Reliability — avoids permission-related failures during automation runs.

8. Error: only-builtins

Description:
The `only-builtins` rule in Ansible-Lint enforces the use of modules exclusively from the **`ansible.builtin` collection**. It helps maintain consistency and avoid dependency issues that arise from relying on external collections such as `kubernetes.core` or `community.general`. This rule is optional and can be enabled in the Ansible-Lint configuration, but once active, it ensures playbooks use only built-in actions.

Symptoms:

- Linter output shows warnings such as:

```
None
only-builtins: Use only builtin actions.

only-builtin.yml:5 Task/Handler: Deploy a Helm chart for
Prometheus
```

- Playbook includes non-builtin modules (e.g., `kubernetes.core.helm`, `community.general.*`).
- Linting passes with warnings but highlights non-compliance.

Resolution:

- Restrict module usage to the `ansible.builtin` collection.

    - Correct: `ansible.builtin.shell`, `ansible.builtin.copy`, `ansible.builtin.file`
    - Incorrect: `kubernetes.core.helm`, `community.general.user`
- If external modules are required:
    - Either disable the `only-builtins` rule in your project, or
    - Justify their usage in documentation so the team understands the dependency.
- Enable the rule in `ansible-lint` config to enforce compliance:

```
None
enable_list:
  - only-builtins
```

Code

(Incorrect → Correct):

```
None
# Incorrect: Uses a non-builtin module
- name: Example playbook
  hosts: all
  tasks:
    - name: Deploy a Helm chart for Prometheus
      kubernetes.core.helm:   # Not part of ansible.builtin
        name: test
        chart_ref: stable/prometheus
```

```
        release_namespace: monitoring

        create_namespace: true
```

```
None
# Correct: Uses only built-in modules

- name: Example playbook

  hosts: localhost

  tasks:

    - name: Run a shell command

      ansible.builtin.shell: echo This playbook uses actions from
the builtin collection only.
```

Benefits of Following only-builtins:

- Consistency — ensures a uniform codebase free from external dependencies.
- Reduced Dependencies — avoids the need to install or maintain external collections.
- Maintainability — built-in modules are stable and tested across Ansible versions.
- Reliability — minimizes compatibility issues when upgrading Ansible or sharing playbooks.

9. Error: parser-error

Description:

The `parser-error` is a generic error reported by Ansible Lint when there are **syntax issues** in a playbook. It indicates that Ansible cannot properly interpret the YAML structure due to problems like inconsistent indentation, missing spaces, or invalid formatting. Because YAML is strict about indentation and spacing, even small mistakes can trigger this error.

Symptoms:

- Linter reports a failure such as:

```
None
parser-error: Failed to parse playbook due to syntax error
```

- Playbook execution may fail immediately without running any tasks.
- Common triggers include:
    - Inconsistent indentation.
    - Missing spaces in key-value pairs.
    - Misaligned YAML blocks.

Resolution:

- **Ensure consistent indentation** throughout the playbook.
    - Use spaces only (not tabs).
    - Stick to a uniform number of spaces per level (typically 2).
- **Add missing spaces** between keys and values.
    - Example:
        - Incorrect: `name:apache2`
        - Correct: `name: apache2`
- **Validate YAML syntax** using tools like `yamllint` before running playbooks.
- **Re-check task structure** to ensure modules and parameters are properly aligned.

Code

(Incorrect → Correct):

None

```
# Incorrect: Inconsistent indentation and missing spaces

- name: Example playbook

  hosts: all

  tasks:

  - name: Install apache

    ansible.builtin.apt:

     name:apache2

     state:present
```

None

```
# Correct: Consistent indentation and proper spacing

- name: Example playbook

  hosts: all

  tasks:

    - name: Install apache

      ansible.builtin.apt:

        name: apache2

        state: present
```

Benefits of Resolving parser-error:

- Reliability — ensures playbooks run without syntax-related interruptions.

- Readability — consistent spacing makes playbooks easier to read and maintain.
- Debugging Efficiency — clear structure simplifies troubleshooting and collaboration.
- Best Practices — aligns with YAML standards and Ansible coding guidelines.

10. Error: run-once

Description:
The `run_once` directive ensures that a task executes only once, regardless of how many hosts are targeted. While powerful, it can cause issues if combined with `strategy: free`. The **run-once linting rule** warns when `run_once` is used in such a context because parallel execution with `strategy: free` may result in the task running more than once or unpredictably.

Symptoms:

- Linter reports violations such as:

```
None
run-once[task]: Using run_once may behave differently if strategy is set to free.
```

- YAML truthy formatting warnings may also appear if `true`/`false` is not lowercase.
- Tasks do not behave predictably when executed with `strategy: free`.

Resolution:

- **Avoid using `run_once` with `strategy: free`.**
    - Instead, use a safer strategy such as `linear` when you need deterministic behavior.
- **If you must combine them intentionally**, suppress the linting error with:

```
None
run_once: true # noqa: run-once[task]
```

- Be sure to document why this exception is necessary.
- **Test execution** carefully to confirm tasks behave as intended when overriding linting rules.

Code

(Incorrect → Correct):

```
None
# Incorrect: run_once used with strategy: free

- name: Example with run_once

  hosts: all

  strategy: free

  gather_facts: false

  tasks:

    - name: Task with run_once

      ansible.builtin.debug:

        msg: "Test"

      run_once: true
```

```
None
# Correct: run_once used with a safe strategy
```

```
- name: Example of using run_once with a strategy other than free
  hosts: all
  strategy: linear
  gather_facts: false
  tasks:
    - name: Task with run_once
      ansible.builtin.debug:
        msg: "Test"
      run_once: true
```

None

```
# Correct (with justification): run_once with strategy: free
intentionally
- name: Example with run_once
  hosts: all
  strategy: free
  gather_facts: false
  tasks:
    - name: Task with run_once
      ansible.builtin.debug:
        msg: "Test"
```

```
    run_once: true # noqa: run-once[task]
```

Benefits of Following the run-once Rule:

- Predictability — ensures tasks execute exactly once as intended.
- Reliability — avoids unexpected results caused by parallelization in `strategy: free`.
- Clarity — makes task execution behavior clear to reviewers and collaborators.
- Flexibility with Documentation — exceptions can still be made safely by explicitly disabling the lint rule when justified.