Troubleshooting Common Ansible Playbook Execution Issues

Ansible is a powerful automation tool, but users may encounter various challenges during its use. This document outlines common issues and provides their resolutions.

1.Error 303: command-instead-of-module

Description:
Ansible-Lint Error 303 (`command-instead-of-module`) warns when raw shell/command tasks are used instead of dedicated Ansible modules. Modules are **more reliable, idempotent, secure, and cross-platform**, making them the preferred way to perform automation tasks.

Symptoms:

- Linter flags violations such as:

```None
command-instead-of-module: apt-get used in place of apt-get
module
303.yml:5 Task/Handler: Run apt-get update
```

- Secondary warnings like:

```None
no-changed-when: Commands should not change things if nothing
needs doing.
```

- Playbook still runs but is **less reliable and non-idempotent**.

Resolution:

1. **Replace raw commands** with equivalent Ansible modules whenever available.
    - Example: use `ansible.builtin.apt` instead of `ansible.builtin.command: apt-get update`.

2. **Check module documentation** (`ansible-doc <module>`) for supported functionality.

3. **Use `# noqa: command-instead-of-module`** only when no suitable module exists (rare cases).

Code (Bad → Good):

```
None
# ❌ Bad: Using a raw command
- name: Update apt cache
  hosts: all
  tasks:
    - name: Run apt-get update
      ansible.builtin.command: apt-get update
```

```
None
# ✅ Good: Using the proper Ansible module
- name: Update apt cache
  hosts: all
  tasks:
    - name: Run apt-get update
      ansible.builtin.apt:
        update_cache: true
```

Benefits of Using Modules Over Commands:

- 🔄 **Reliability** — modules are idempotent (only make changes when necessary).
- 📖 **Readability** — more descriptive and easier to understand.
- ⚙️ **Extensibility** — modules offer parameters for more control.
- 🌍 **Cross-Platform Compatibility** — modules work across multiple OSes.
- 🔒 **Security** — modules handle sensitive data more safely.

Exception Handling:
If no suitable module exists and a command must be used:

```
None
- name: Run a one-off shell command
  ansible.builtin.command: some-unsupported-command # noqa:
command-instead-of-module
```

2.Error 304: inline-env-var

Description:
Ansible-Lint Error 304 (`inline-env-var`) occurs when environment variables are set directly inside the **`ansible.builtin.command`** module. This practice is discouraged because it reduces clarity, breaks idempotence, and makes playbooks harder to maintain. Instead, environment variables should be defined using the **`environment`** **keyword** or handled via the **`ansible.builtin.shell`** module.

Symptoms:

- Linter flags violations such as:

```
None
inline-env-var: Command module does not accept setting
environment variables inline.

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

- Example violation in playbook:

```
None
ansible.builtin.command: MY_ENV_VAR=my_value
```

- Execution might succeed, but fails lint checks and is considered bad practice.

Resolution:

1. **Use the `environment` keyword** with a task.
2. **Switch from `command` to `shell`** if inline environment variables are unavoidable.
3. **Ensure idempotence** by separating environment setup from the command itself.

Code (Bad → Good):

```
None
# ❌ Bad: Inline env var in command module

- name: Set environment variable

  ansible.builtin.command: MY_ENV_VAR=my_value
```

```
None
# ✅ Good: Use environment keyword

- name: Set environment variable

  ansible.builtin.shell: echo $MY_ENV_VAR

  environment:

    MY_ENV_VAR: my_value
```

```
None
# ✅ Alternative Good: Use shell with inline env var

- name: Set environment variable

  ansible.builtin.shell: MY_ENV_VAR=my_value
```

Benefits of Correct Usage:

- 📖 **Clarity** — environment variables are declared explicitly and separately.
- 🔄 **Predictability** — consistent behavior across tasks and environments.
- 🛡️ **Idempotence** — environment management does not interfere with task results.
- ⚡ **Flexibility** — easy to extend or modify environment variables without rewriting commands.

3.Error 305: command-instead-of-shell

Description:
Ansible-Lint Error 305 (`command-instead-of-shell`) flags the use of the **shell module** when the **command module** would suffice. The `command` module should be preferred for simple commands, since it is faster, safer, and more predictable. The `shell` module should only be used when shell-specific features are required (e.g., pipes, redirection, environment variable expansion).

Symptoms:

- Linter reports violations such as:

```
None
command-instead-of-shell: Use shell only when shell functionality
is required.

305.yml:5 Task/Handler: Echo a message
```

- Playbook runs successfully, but lint checks fail.
- Performance and security may be impacted by unnecessary use of `shell`.

Resolution:

1. **Use `ansible.builtin.command` instead of `ansible.builtin.shell`** for simple commands.
2. **Reserve `ansible.builtin.shell`** for cases requiring:

    - Pipes (`|`), redirection (`>`), `&&`, `||`.
    - Environment variable expansion (`$VAR`).
    - Other shell-specific constructs.
3. **Review existing tasks** to ensure modules align with their intended functionality.

Code (Bad → Good):

```
None
# ❌ Bad: Using shell unnecessarily

- name: Problematic example

  hosts: all

  tasks:

    - name: Echo a message

      ansible.builtin.shell: echo hello   # Shell not required

      changed_when: false
```

```
None
# ✅ Good: Using command correctly

- name: Correct example

  hosts: all

  tasks:

    - name: Echo a message

      ansible.builtin.command: echo hello
      changed_when: false
```

Why Prefer `command` Over `shell`:

- ⚡ **Efficiency** — faster execution.
- 🔮 **Predictability** — no shell interpretation quirks.
- 🔁 **Idempotence** — behaves more consistently across runs.
- 🔒 **Security** — reduces exposure to shell injection risks.

Exceptions:

- Use `shell` only when absolutely necessary (e.g., `grep pattern /etc/passwd | awk '{print $1}'`).
- Justify the trade-off if shell features are required.
  one during playbook writing?

4.**Error 306: risky-shell-pipe**

Description:
Ansible-Lint Error 306 (`risky-shell-pipe`) occurs when you use the **`shell` module with pipelines** (`|`) but don't enable the **`pipefail` option**. Without `pipefail`, the shell may report success even if the first command in the pipeline fails, leading to **unreliable or misleading task results**.

Symptoms:

- Linter flags violations such as:

```
None
risky-shell-pipe: Shells that use pipes should set the pipefail
option.
```

- Paired with other warnings, e.g.:

```None
no-changed-when: Commands should not change things if nothing
needs doing.
```

- Tasks with pipelines may **not fail as expected** if the first command in the chain fails.

Resolution:

1. **Always set `pipefail`** in tasks that use pipelines.
2. **Explicitly define the shell executable** (`/bin/bash`) since `pipefail` is a Bash option.
3. **Use multi-line commands** when readability matters.
4. **If intentional** (non-critical tasks), document why `pipefail` is omitted.

Code (Bad → Good):

```None
# ❌ Bad: Pipeline without pipefail

- name: Pipeline without pipefail

  ansible.builtin.shell: false | cat
```

```None
# ✅ Good: Pipeline with pipefail (single-line)

- name: Pipeline with pipefail

  ansible.builtin.shell:

    cmd: set -o pipefail && false | cat

    executable: /bin/bash
```

```
None
# ✅ Good: Pipeline with pipefail (multi-line for readability)

- name: Pipeline with pipefail, multi-line

  ansible.builtin.shell:

    cmd: |

      set -o pipefail   # ensures proper failure behavior

      false | cat

    executable: /bin/bash
```

Why Use the `pipefail` Option:

- ⚖️ **Predictable Failure** — ensures tasks fail when the first command in a pipeline fails.
- 🔁 **Idempotence** — aligns with Ansible's design for consistent, reliable automation.
- 🐞 **Enhanced Debugging** — makes failure sources in pipelines easier to identify.
- 🔒 **Security** — prevents silent failures that could create unintended consequences.

Exception Handling:

- In rare cases, you may omit `pipefail` (e.g., for **non-critical pipelines** where failure of early commands is acceptable).
- Document these exceptions to clarify intent for collaborators.

5.**Error 401: latest[git]**

Description:
Ansible-Lint Error 401 (`latest[git]`) warns against using **variable or floating references** in Git checkouts, such as `HEAD` or `latest`. These values can cause **unpredictable behavior** because the result depends on the latest commit of the branch at execution time. For

reproducibility, playbooks should pin Git repositories to **specific commits, tags, or stable branches**.

Symptoms:

- Linter reports:

```None
latest[git]: Result of the command may vary on subsequent runs.
```

- Example violation occurs when:

```None
version: HEAD
```

- Playbook behavior changes over time as new commits are pushed to the repository.

Resolution:

1. **Avoid `HEAD`, `latest`, or floating refs** in the `version` argument.
2. **Pin repositories** to:
    - A **specific commit hash** (e.g., `abcd1234`).
    - A **tagged release** (e.g., `v2.15.0`).
    - A **stable branch** only if immutability is not required.
3. **If you intentionally want the latest**, you can suppress the rule by adding `# noqa: latest` inline — but use this sparingly.

Code (Bad → Good):

None

```
# ❌ Bad: Risky use of HEAD

- name: Risky use of git module

  ansible.builtin.git:

    repo: "https://github.com/ansible/ansible-lint"

    version: HEAD    # Floating reference, unpredictable
```

None

```
# ✅ Good: Safe use with a specific commit hash

- name: Safe use of git module

  ansible.builtin.git:

    repo: "https://github.com/ansible/ansible-lint"

    version: abcd1234    # Pinned commit ensures reproducibility
```

None

```
# ✅ Good: Safe use with a tag

- name: Safe use with tagged release

  ansible.builtin.git:

    repo: "https://github.com/ansible/ansible-lint"

    version: v2.15.0     # Tagged release
```

```
None
# ⚠️ Intentional latest (with rule ignored)

- name: Intentionally fetch latest commit

  ansible.builtin.git:

    repo: "https://github.com/ansible/ansible-lint"

    version: HEAD   # noqa: latest
```

Benefits of Following Rule 401:

- 🔄 **Idempotency** — ensures repeated runs always produce the same results.
- 🔒 **Reliability** — prevents unexpected changes from upstream repositories.
- 📖 **Clarity** — makes the target version explicit for teammates.
- 🛠️ **Controlled Flexibility** — intentional "latest" behavior can still be documented with `# noqa`.

6.**Error 402: latest[hg]**

Description:
Ansible-Lint Error 402 (`latest[hg]`) warns when **Mercurial (hg) repositories** are checked out using variable or non-deterministic arguments such as `revision: HEAD`. Using `HEAD` means fetching the latest commit from the default branch, which can change over time and make playbook runs unpredictable. This rule is a consolidated replacement for older rules (`git-latest` and `hg-latest`) and ensures **reproducibility and stability** in source control checkouts.

Symptoms:

- Linter flags risky use of `revision: HEAD` (or other floating references).
- Example violation:

```None
revision: HEAD  # <-- HEAD value is triggering the rule
```

- Playbooks may behave inconsistently if new commits are introduced between runs.

Resolution:

1. **Use specific commit identifiers (SHA) instead of `HEAD`.**
   - ✅ `revision: abcd1234...`
   - ❌ `revision: HEAD`
2. **If intentional**, explicitly suppress the rule using `# noqa: latest`.
   - Useful when you really want to always fetch the latest commit.
3. **Document rationale** when bypassing the rule, so team members understand why reproducibility is not enforced.

Code (Bad → Good):

```None
# ❌ Bad: Risky, non-deterministic checkout

- name: Risky use of hg module

  community.general.hg:

    repo: "https://github.com/ansible/ansible"

    revision: HEAD
```

```None
# ✅ Good: Safe, deterministic checkout
```

```
- name: Safe use of hg module

  community.general.hg:

    repo: "https://github.com/ansible/ansible"

    revision: abcd1234...   # specific commit ID
```

```
None
# ✅ Intentional override (documented)

- name: Fetch latest commit intentionally

  community.general.hg:

    repo: "https://github.com/ansible/ansible"

    revision: HEAD   # noqa: latest
```

Benefits of Following Rule 402:

- 🔒 **Predictability** — same commit checked out across all runs.
- 🔄 **Reproducibility** — playbooks produce consistent results over time.
- 🛡️ **Reliability** — avoids sudden failures caused by upstream changes.
- 🎯 **Clarity** — makes it explicit whether a checkout is fixed or floating.

7.**Error 403: package-latest**

Description:
Ansible-Lint Error 403 (`package-latest`) warns when the **state parameter** of package manager modules is set to `latest`. Using `latest` installs the newest available version of a package, which can introduce **unpredictability, service disruptions, or unintended dependencies**. In production environments, it's best practice to pin packages to a specific version or use `state: present`.

Symptoms:

- Linter flags multiple violations like:

```
None
package-latest: Package installs should not use latest.
```

- Playbooks may:
  - Install newer versions than expected.
  - Pull in additional dependencies.
  - Cause regressions or service instability.

Resolution:

1. **Pin specific versions** for stability:
   - ✅ `state: present` + version (for yum, apt, pip).
   - ❌ `state: latest` without control.
2. **Use `update_only: true` (yum)** or **`only_upgrade: true` (apt)** if your intention is strictly to upgrade existing packages.
3. **Reserve `latest` usage** for controlled environments (dev/test), never for production.

Code (Bad → Good):

```
None
# ❌ Bad: Using latest across different modules

- name: Install Ansible

  ansible.builtin.yum:

    name: ansible
    state: latest


- name: Install Ansible-lint
  ansible.builtin.pip:
    name: ansible-lint
  args:
    state: latest


- name: Install some-package
  ansible.builtin.package:
    name: some-package
    state: latest
```

None
```
# ✅ Good: Version-pinned or safe upgrades
- name: Install Ansible (specific version)
  ansible.builtin.yum:
    name: ansible-2.12.7.0
    state: present


- name: Install Ansible-lint (specific version via pip)
```

```yaml
  ansible.builtin.pip:
    name: ansible-lint
  args:
    state: present
    version: 5.4.0

- name: Install some-package (ensures present)
  ansible.builtin.package:
    name: some-package
    state: present

- name: Update Ansible safely with yum
  ansible.builtin.yum:
    name: sudo
    state: latest
    update_only: true

- name: Update Ansible safely with apt
  ansible.builtin.apt:
    name: sudo
    state: latest
    only_upgrade: true
```

Benefits of Following Rule 403:

- 🔒 **Stability** — prevents unexpected updates breaking production.
- 🎯 **Predictability** — ensures consistent package versions across environments.
- 🛡️ **Security** — limits the risk of introducing untested dependencies.
- ⚡ **Controlled Flexibility** — allows upgrades only when explicitly intended.

8.**Error 404: no-relative-paths**

Description:
Ansible-Lint Error 404 (`no-relative-paths`) occurs when **relative paths** are used in the `src` argument of the `ansible.builtin.copy` or `ansible.builtin.template` modules. Relative paths (e.g., `../my_templates/foo.j2`) can cause confusion, project disorganization, and unpredictable results. Instead, Ansible enforces a clear structure by requiring files to be placed inside dedicated **`files/`** and **`templates/`** directories.

Symptoms:

- Linter flags violations such as:

```
None
src: ../my_templates/foo.j2   # relative path not allowed
```

- Variables containing relative paths also trigger this rule:

```
None
source_path: ../../my_templates/foo.j2

src: "{{ source_path }}"
```

- Playbooks may fail if paths are misinterpreted or unavailable.

Resolution:

1. **Use the `files/` directory** for files referenced by the `copy` module.
2. **Use the `templates/` directory** for Jinja2 templates referenced by the `template` module.
3. **Reference files by name (or subfolder paths)** inside these dedicated directories, not by relative paths.
4. **Refactor variables** to point to clean file names instead of relative paths.

Code (Bad → Good):

```
None
# ❌ Bad: Using relative paths

- name: Template a file to /etc/file.conf

  ansible.builtin.template:

    src: ../my_templates/foo.j2

    dest: /etc/file.conf

    owner: bin

    group: wheel

    mode: "0644"



- name: Copy a file to /etc/file.conf

  vars:

    source_path: ../../my_templates/foo.j2
```

```
tasks:
  - name: Copy with relative path
    ansible.builtin.copy:
      src: "{{ source_path }}"
      dest: /etc/foo.conf
      owner: foo
      group: foo
      mode: "0644"
```

None

```
# ✅ Good: Using recommended files/ and templates/ directories
- name: Template a file to /etc/file.conf
  ansible.builtin.template:
    src: foo.j2          # from templates/ directory
    dest: /etc/file.conf
    owner: bin
    group: wheel
    mode: "0644"

- name: Copy a file to /etc/file.conf
  vars:
    source_path: foo.j2  # from files/ directory

  tasks:

    - name: Copy with safe path

      ansible.builtin.copy:

        src: "{{ source_path }}"

        dest: /etc/foo.conf

        owner: foo

        group: foo

        mode: "0644"
```

Benefits of Following Rule 404:

- 📂 **Organized Project Structure** — files and templates stored in dedicated locations.
- 🔍 **Clarity & Predictability** — eliminates confusion about where resources come from.
- 🔄 **Consistency** — ensures playbooks run reliably in different environments.
- 🛡️ **Error Prevention** — avoids issues from misconfigured or missing relative paths.

9.Error 501: partial-become

Description:
Ansible-Lint Error 501 (`partial-become`) is triggered when **`become_user` is used without `become: true`**. Ansible requires both directives together to reliably change users. Without `become: true`, the `become_user` directive is ignored, leading to inconsistent or unexpected behavior. This rule enforces **explicit and consistent privilege escalation** at the task or play level.

Symptoms:

- Linter reports:

```
None
partial-become[task]: `become_user` should have a corresponding
`become` at the play or task level.
```

-
  Tasks specifying `become_user` do not actually change the user.
- Privilege escalation appears partially configured but doesn't take effect.

Resolution:

1. Always pair `become_user` with `become: true`.
   - Correct:

```
None
become: true

become_user: apache
```

   - Incorrect:

```
None
become_user: apache   # Without become: true
```

2. Define privilege escalation at the **task level** for specific actions.
3. Apply `become: true` and `become_user` at the **play level** if escalation is needed across the entire play.

Code (Incorrect → Correct):

None

```
# Incorrect: Incomplete privilege escalation

- name: Example playbook

  hosts: all

  tasks:

    - name: Start the httpd service as the apache user

      ansible.builtin.service:

        name: httpd

        state: started

      become_user: apache   # Missing "become: true"
```

None

```
# Correct: Proper privilege escalation at task level

- name: Example playbook

  hosts: all

  tasks:

    - name: Start the httpd service as the apache user

      ansible.builtin.service:

        name: httpd

        state: started

      become: true

      become_user: apache
```

```
None
# Correct: Privilege escalation defined at play level

- name: Example playbook

  hosts: localhost

  become: true

  become_user: apache

  tasks:

    - name: Start the httpd service as the apache user

      ansible.builtin.service:

        name: httpd

        state: started
```

Benefits of Following Rule 501:

- Security and Predictability — ensures privilege escalation works as intended.
- Clarity — makes privilege escalation explicit for reviewers and collaborators.
- Error Prevention — avoids tasks silently ignoring `become_user`.
- Consistency — guarantees user changes behave reliably across tasks and plays.

10.Error 502: name[missing]

Description:
Ansible-Lint Error 502 (`name[missing]`) is triggered when tasks or plays are missing a **descriptive `name` field**. Task names are not just cosmetic—they are essential for readability, traceability in logs, and effective debugging. Without them, playbook output becomes harder to follow, and automation workflows become less maintainable.

Symptoms:

- Linter reports:

```
None
name[missing]: All tasks should be named.

name[play]: All plays should be named.
```

- Unnamed tasks appear in execution logs as raw module calls (e.g., `command touch /tmp/.placeholder`).
- Playbooks are harder to debug and understand.

Resolution:

1. Always provide a descriptive `name` for **every play** and **every task**.
2. Choose names that reflect the **purpose of the action** (not just the module being used).
3. Ensure names are **concise but clear** so logs and reports are easily interpretable.

Code (Incorrect → Correct):

```
None
# Incorrect: Unnamed play and unnamed task

- hosts: all

  tasks:

    - ansible.builtin.command: touch /tmp/.placeholder
```

```
None
# Correct: Play and task both have descriptive names
```

```yaml
- name: Play for creating placeholder

  hosts: all

  tasks:

    - name: Create a placeholder file

      ansible.builtin.command: touch /tmp/.placeholder
```

Benefits of Following Rule 502:

- Readability — makes it clear what each task or play is doing.
- Traceability — improves log output and makes debugging easier.
- Maintainability — descriptive names help teams quickly understand automation code.
- Best Practices — aligns with Ansible’s idiomatic style, fostering consistency across projects.