Ansible Playbook Execution Issues

Ansible is a powerful automation tool, but users may encounter various challenges during its use. This document outlines frequently observed problems and their corresponding solutions.

1.Error: avoid-implicit

Description:
The Ansible-Lint rule `avoid-implicit` flags cases where your playbooks rely on **implicit behaviors** rather than explicit instructions. Implicit behaviors happen when Ansible interprets inputs in a way you didn't specify, which can lead to **unpredictable results, harder debugging, and reduced maintainability**.

Symptoms:

- Linter reports:

```
None
avoid-implicit: Avoid implicit behaviors
```

- Tasks behave differently than expected because of ambiguous inputs.
- Example: Passing a dictionary directly to the `content` parameter in `ansible.builtin.copy` may result in unexpected file content.

Resolution:

1. Always use **explicit Jinja2 expressions** to clarify how data should be handled.
2. Convert dictionaries, lists, or complex objects to JSON or YAML explicitly before writing them into files.
3. Review module documentation to understand what input types are accepted.
4. Use linting (`ansible-lint`) and testing to catch unintended implicit behaviors early.

Code (Incorrect → Correct):

```
None
# Incorrect: Implicit dictionary passed to content
- name: Write file content
  ansible.builtin.copy:
    content: { "foo": "bar" }   # Ambiguous, triggers
avoid-implicit
    dest: /tmp/foo.txt
```

```
None
# Correct: Explicit Jinja2 conversion to JSON
- name: Write file content
  vars:
    content: { "foo": "bar" }
  ansible.builtin.copy:
    content: "{{ content | to_json }}"
    dest: /tmp/foo.txt
```

Benefits of Avoiding Implicit Behaviors:

- Predictability — ensures tasks behave the same way every time.
- Easier Debugging — explicit instructions simplify troubleshooting.
- Documentation — clear code explains itself and avoids hidden assumptions.
- Maintainability — reduces long-term technical debt as playbooks evolve.

Best Practices:

- Use explicit instructions in modules (avoid shortcuts).
- Check module documentation to understand valid inputs.
- Test playbooks in controlled environments before production.
- Integrate `ansible-lint` into your workflow to detect implicit behaviors automatically.

2.Error: fqcn (Fully-Qualified Collection Names)

Description:
The `fqcn` rule in Ansible-Lint enforces the use of **Fully-Qualified Collection Names (FQCNs)** for modules, actions, and plugins. An FQCN specifies the full namespace (e.g., `ansible.builtin.shell`) instead of relying on short or ambiguous names like `shell`. This avoids ambiguity, ensures consistent behavior, and improves long-term maintainability of playbooks.

Symptoms:

- Linter reports violations such as:

```
None
fqcn[action-core]: Use FQCN for builtin module actions (shell).

fqcn.yml:5 Use `ansible.builtin.shell` or `ansible.legacy.shell`
instead.
```

- Other related warnings may occur:
    - `command-instead-of-shell` — advising correct module usage.
    - `no-changed-when` — reminding about idempotency in command/shell usage.
- Code without FQCN may still run but risks breaking with future versions or causing ambiguity between collections.

Resolution:

1. **Always use FQCNs** for modules and actions.
    - Correct: `ansible.builtin.shell` or `ansible.legacy.shell`
    - Incorrect: `shell`
2. **Prefer canonical module names** over aliases or redirects (e.g., use `ansible.builtin.copy` instead of older aliases).

3. **Avoid deep/nested plugin directories** — follow the flat directory guidance from Ansible's core team.

4. Use `ansible-lint --fix` to automatically correct some FQCN violations.

Code (Incorrect → Correct):

```
None
# Incorrect: Short module name without FQCN

- name: Example playbook

  hosts: all

  tasks:

    - name: Create an SSH connection

      shell: ssh ssh_user@{{ ansible_ssh_host }}
```

```
None
# Correct: Explicit FQCN for legacy shell

- name: Example playbook (1st solution)

  hosts: all

  tasks:

    - name: Create an SSH connection

      ansible.legacy.shell:

        ssh ssh_user@{{ ansible_ssh_host }} -o
IdentityFile=path/to/my_rsa
```

```
None
# Correct: Explicit FQCN for builtin shell

- name: Example playbook (2nd solution)

  hosts: all

  tasks:

    - name: Create an SSH connection

      ansible.builtin.shell: ssh ssh_user@{{ ansible_ssh_host }}
```

Benefits of Using FQCNs:

- Reliability — removes ambiguity between collections and modules.
- Standardization — aligns with modern Ansible best practices.
- Compatibility — ensures scripts are future-proof across Ansible versions.
- Performance — avoids overhead of resolving aliases or deep module paths.

3.Error galaxy

Description:
The Ansible-Lint **`galaxy` rule** validates the quality and completeness of Ansible collections defined in the `galaxy.yml` file. It checks for proper versioning, changelog presence, required tags, metadata fields, and valid dependency formats. Violations of this rule may prevent your collection from being certified on **Automation Hub** or published on **Ansible Galaxy**.

Symptoms:

- Linter flags violations such as:

```
None
galaxy[version-incorrect]: collection version should be greater
than or equal to 1.0.0
```

```
galaxy[no-changelog]: No changelog found. Please add a changelog
file.

galaxy[tags]: galaxy.yaml must have one of the required tags:
['application', 'cloud', 'database', 'infrastructure', 'linux',
'monitoring', 'networking', 'security', 'storage', 'tools',
'windows']

schema[galaxy]: $ 'readme' is a required property
```

- Playbooks or collections may not be accepted for certification.
- Issues include missing changelogs, invalid tags, versions below `1.0.0`, or malformed dependency version ranges.

Resolution:

1. **Collection versioning**
    - Ensure `version` in `galaxy.yml` is **greater than or equal to 1.0.0**.
    - Correct: `version: 1.0.0`
    - Incorrect: `version: 0.2.3`
2. **Changelog requirement**
    - Add a `CHANGELOG.md`, `CHANGELOG.rst`, or `changelogs/changelog.yaml`.
    - Document features, bug fixes, and changes clearly.
3. **Required certification tags**
    - Include at least one tag from the approved list:
      `application, cloud, database, infrastructure, linux, monitoring, networking, security, storage, tools, windows`
    - Example:

```
None
tags: [networking, security]
```

4. **Metadata completeness**
    - Ensure `readme`, `authors`, `description`, and `license` are properly defined in `galaxy.yml`.
5. **Dependency versioning**
    - Define dependencies with valid version ranges and proper syntax.

Code (Incorrect → Correct):

```
None
# Incorrect: Version too low, missing tags, no changelog,
incomplete metadata

# galaxy.yml

---

name: my_collection

namespace: my_namespace

version: 0.2.3
```

```
None
# Correct: Valid version, tags, and metadata

# galaxy.yml

---

namespace: my_namespace
```

```
name: my_collection

version: 1.0.0

authors:

  - John

description: "Collection for managing networking tasks"

tags: [networking, security]

readme: README.md

license: GPL-2.0-or-later
```

Benefits of Following the galaxy Rule:

- Standardization — ensures collections meet Galaxy and Automation Hub requirements.
- Discoverability — certified tags help users quickly find your collection.
- Transparency — changelogs provide clear documentation of changes.
- Reliability — valid versions and dependencies improve stability.
- Certification Readiness — collections remain eligible for certification and publishing.

4.Error: internal-error

Description:
The `internal-error` in Ansible-Lint indicates that an **unexpected runtime exception** or **invalid playbook syntax** occurred, often caused by:

- Internal bugs within Ansible itself.
- Invalid Jinja2 templates or YAML structures.
- Overly restrictive or broken custom linting rules.

When this error occurs, Ansible will continue processing other files but stop applying additional rules to the file where the internal error was triggered.

Symptoms:

- Linter reports:

```
None
internal-error: Unexpected error code 1 from execution of:
ansible-playbook -i localhost, --syntax-check internal-error.yml

ERROR! template error while templating string: unexpected 'end of
template'. String: Some title {{. unexpected 'end of template'
```

- Playbook fails to load due to invalid syntax or template.
- Runtime error messages appear vague and may point to deeper issues.

Resolution:

1. **Check for invalid Jinja2 templates.**
    - Ensure all `{{ ... }}` expressions are properly closed.
    - Remove or correct incomplete or malformed templates.
2. **Review custom linting rules.**
    - Validate that they are not overly restrictive or incorrectly defined.
3. **Handle playbook indexing errors safely.**
    - Use default filters to avoid indexing errors.
    - Example:

```
None
hosts: "{{ groups['all'][1] | default([]) }}"
```

4. **Keep Ansible updated.**
    - Some internal errors are due to bugs fixed in later releases.
5. **Exclude problematic files (if intentional).**
    - Use `exclude_paths` in `.ansible-lint` configuration when testing edge cases.

Code (Incorrect → Correct):

```
None
# Incorrect: Invalid Jinja2 template

- name: Some title {{   # Invalid template

  hosts: all

  tasks: []
```

```
None
# Correct: Valid syntax, template removed

- name: Some title

  hosts: all

  tasks: []
```

Associated Runtime Error Example:

```
None
ERROR! No hosts matched the subscripted pattern
```

This occurs when attempting to access an inventory group index that does not exist. Use safe fallbacks to prevent runtime failures.

```
None
# Safer host definition

hosts: "{{ groups['all'][1] | default([]) }}"
```

Strategies for Handling Internal Errors:

- Resolve bugs by checking for known issues and applying updates.
- Review and adjust custom rules to prevent false positives.
- Use safe fallbacks in templates and inventory references.
- Exclude intentionally problematic files during linting to reduce noise.

Benefits of Proper Handling:

- Improves **resilience** of playbooks against runtime exceptions.
- Enhances **clarity and maintainability** by avoiding cryptic errors.
- Supports **predictability** in automation workflows.
- Builds **defensive coding habits**, making playbooks robust in varied environments.

5.Error: key-order

Description:
Ansible-Lint's `key-order` rule enforces a consistent order for keys in plays, tasks, and handlers. The rule ensures that:

- The **`name`** **key comes first** in plays, tasks, and handlers.
- Keys such as **`block`**, **`rescue`**, and **`always`** appear last.
  Maintaining consistent ordering improves readability, reduces indentation errors, and creates more maintainable playbooks.

Symptoms:

- Linter reports violations such as:

```
None
key-order[play]: You can improve the play key order to: name,
hosts, tasks

key-order[task]: You can improve the task key order to: name,
when, block
```

- Keys appear misplaced, e.g.:
    - `name` not listed first in a play or task.
    - `when` placed after `block`.
- Playbooks may still execute, but they fail linting checks and become harder to maintain.

Resolution:

1. Place `name` as the first key in every play, task, and handler.
2. Keep conditional keys like `when` before structural keys like `block`.
3. Ensure `block`, `rescue`, and `always` are always positioned last.
4. Use `ansible-lint --fix` to automatically reorder keys.

Code (Incorrect → Correct):

```
None
# Incorrect: name not first, when misplaced

- hosts: all

  name: This is a playbook

  tasks:

    - name: A block
```

```
    block:
      - name: Display a message
        ansible.builtin.debug:
          msg: "Hello world!"
    when: true
```

```
None
# Correct: name first, when before block
- name: This is a playbook
  hosts: all
  tasks:
    - name: A block
      when: true
      block:
        - name: Display a message
          ansible.builtin.debug:
            msg: "Hello world!"
```

Benefits of Following the key-order Rule:

- Code Maintenance — playbooks remain easy to read and navigate, even as they grow.
- Error Prevention — reduces misindentation and misplaced keys.

- Resilience — tasks are less likely to break when moved or modified.
- Automation Support — can be fixed automatically with `ansible-lint --fix`.

6.Error: load-failure

Description:
The `load-failure` error occurs during linting when **Ansible-Lint cannot parse or process a file**. This error is unskippable and indicates a critical issue with the playbook or role file. It often points to problems such as invalid YAML, unsupported file encodings, or vault decryption issues.

Symptoms:

- Linter fails with messages like:

```
None
load-failure[runtimeerror]: Failed to load YAML file

load-failure.yml:1 while parsing a tag

did not find expected tag URI
```

- Error codes may include:
    - `load-failure[not-found]` — file or folder not found.
    - `load-failure[runtimeerror]` — syntax errors, usually invalid YAML format.
- Playbooks fail linting and stop further processing.

Possible Causes:

1. **Unsupported Encoding** — only UTF-8 encoding is supported.
2. **Not an Ansible File** — non-playbook files may trigger the error.
3. **Unsupported Custom YAML Objects** — use of unsupported tags like `!!custom`.

4. **Vault Decryption Issues** — problems decrypting `!vault` blocks due to missing or incorrect vault password.

Resolution:

1. **Ensure valid YAML format.**
    - Avoid unsupported YAML objects (e.g., `!!`).
    - Use a YAML validator to confirm syntax.
2. **Confirm file encoding is UTF-8.**
3. **Check vault decryption.**
    - Provide correct vault password or vault ID.
    - If inline vault blocks fail to decrypt, verify the encryption format.
4. **Verify file paths.**
    - Ensure referenced files exist on disk.
5. **Exclude problematic files** using `exclude_paths` if the error cannot be resolved (for example, files intentionally containing encrypted content).

Code (Incorrect → Correct):

```
None
# Incorrect: Custom YAML object triggers load-failure

- name: Example playbook

  hosts: all

!! custom: true
```

```
None
# Correct: Valid YAML playbook

- name: Example playbook

  hosts: all

  tasks:

    - name: Display a message

      ansible.builtin.debug:

        msg: "Hello world!"
```

Benefits of Resolving load-failure:

- Reliability — ensures playbooks are valid YAML and processable by Ansible.
- Consistency — avoids unpredictable parsing errors across environments.
- Maintainability — removes unsupported constructs, making code easier to manage.
- Quality Assurance — ensures linting works as intended to detect deeper issues.

7.Error: loop-var-prefix

Description:
The `loop-var-prefix` rule in Ansible-Lint enforces clear and consistent naming for loop variables. By default, Ansible uses the variable name `item` in loops, which can cause ambiguity in **nested loops** or across complex tasks. This rule encourages you to explicitly define loop variables and, if configured, apply a naming prefix (for example, your role name).

Symptoms:

- Linter reports:
    - `loop-var-prefix[missing]`: indicates that the loop is using the default `item` variable and you should explicitly define a unique `loop_var`.

    - `loop-var-prefix[wrong]`: indicates that the loop variable is defined but does not match the required prefix set in your `.ansible-lint` configuration.

- Playbook may still run, but the use of ambiguous `item` variables makes code harder to understand and maintain.

Resolution:

1. **Define a custom loop variable** with `loop_control.loop_var`.
2. **Apply a naming prefix** (for example, role name) to loop variables if required by configuration.
3. **Avoid the default `item` variable** in nested loops to prevent ambiguity.

Code (Incorrect → Correct):

None

```
# Incorrect: Uses default "item" (ambiguous in nested loops)

- name: Does not set a variable name for loop variables

  ansible.builtin.debug:

    var: item

  loop:

    - foo

    - bar
```

None

```
# Incorrect: Defines a custom variable, but prefix is wrong

- name: Sets a variable name without required prefix
```

```
  ansible.builtin.debug:
    var: zz_item
  loop:
    - foo
    - bar
  loop_control:
    loop_var: zz_item   # Does not follow prefix rules
```

```
None
# Correct: Defines a unique variable name with required prefix
- name: Sets a unique variable name with role prefix
  ansible.builtin.debug:
    var: myrole_item
  loop:
    - foo
    - bar
  loop_control:
    loop_var: myrole_item   # Matches role prefix
```

Benefits of Following the Rule:

- Clarity — explicit, descriptive loop variables make code easier to read.

- Maintainability — reduces confusion when working with multiple or nested loops.
- Consistency — enforces a systematic naming convention for loops across playbooks.
- Collaboration — improves understanding for teams maintaining or reviewing code.

8.Error: meta-runtime

Description:
The `meta-runtime` rule validates the **`requires_ansible` key** in a collection's `meta/runtime.yml` file. It ensures that the collection declares a **supported version of Ansible-core** (for example, 2.13.x, 2.14.x, or 2.15.x). If an unsupported or outdated version is specified, or the schema is invalid, linting fails. This rule prevents compatibility issues and ensures that collections are only installed and used with Ansible versions known to work correctly.

Symptoms:

- Linter reports violations such as:

```
None
meta-runtime: Required ansible version in meta/runtime.yml must
be a supported version.

schema[meta-runtime]: $ None is not of type 'object'.
```

- Playbooks or collections may not load correctly.
- Unsupported version specifications (for example, `>=2.9`) trigger failures.

Resolution:

1. Always include the **`requires_ansible` key** in `meta/runtime.yml`.
2. Specify a **supported minimum version** of Ansible (for example, `>=2.14.0`).
3. Follow the official Ansible collection structure documentation.

4. Validate the runtime file with `ansible-lint` after updates.

Code (Incorrect → Correct):

```
None
# Incorrect: Unsupported version

# runtime.yml

---

requires_ansible: ">=2.9"
```

```
None
# Correct: Supported version

# runtime.yml

---

requires_ansible: ">=2.14.0"
```

Benefits of Following meta-runtime Rule:

- Compatibility — ensures collections run with supported Ansible-core versions.
- Reliability — prevents runtime errors due to outdated or invalid version requirements.
- Best Practices — aligns with community standards for collection development.
- Maintainability — makes it easier for users to know the minimum version required.
- User Confidence — provides clarity and reduces unexpected issues during execution.

9.Error: name[casing]

Description:

The `name[casing]` error occurs when **task or play names start with a lowercase letter**

instead of an uppercase one. Ansible-Lint enforces this convention to maintain consistency and readability in playbooks. Since names are displayed in logs, consoles, and dashboards, starting them with uppercase letters makes them easier to scan and understand.

Symptoms:

- Linter flags violations such as:

```
None
name[casing]: All names should start with an uppercase letter.

name-casing.yml:5 Task/Handler: create directory
```

- Common issues include:
    - Task names starting with lowercase letters.
    - Play names not starting with uppercase letters.
    - Inconsistent naming styles across playbooks.

Resolution:

1. Ensure all **task names** begin with an uppercase letter.
    - Correct: `Create directory`
    - Incorrect: `create directory`
2. Ensure all **play names** also begin with uppercase.
3. Use `ansible-lint --fix` to automatically correct casing where possible.

Code (Incorrect → Correct):

```
None
# Incorrect: Task name starts with lowercase
```

```
- name: Example playbook
  hosts: all
  tasks:
    - name: create directory
      ansible.builtin.file:
        path: /tmp/mydir
        state: directory
```

```
None
# Correct: Task name starts with uppercase
- name: Example playbook
  hosts: all
  tasks:
    - name: Create directory
      ansible.builtin.file:
        path: /tmp/mydir
        state: directory
```

Benefits of Following name[casing]:

- Readability — clearer task and play names in logs and outputs.
- Consistency — enforces a uniform style across the team's playbooks.

- Maintainability — makes code easier to scan, debug, and update.
- Professionalism — clean naming conventions improve overall code quality.

10.Error: name[play]

Description:
Ansible-Lint Error `name[play]` is raised when a play in a playbook is missing a **descriptive name**. While Ansible allows unnamed plays, not naming them makes it harder to understand the play's purpose and complicates troubleshooting. Play names serve as identifiers in logs, console output, and reports, so adding them improves **clarity, maintainability, and professionalism**.

Symptoms:

- Linter reports:

```
None
name[play]: All plays should be named.
```

- Playbook runs, but the output shows plays without context.
- Reviewing or debugging unnamed plays is harder since their purpose is unclear.

Resolution:

1. Add a descriptive `name` field at the play level.
2. Ensure play names describe the purpose of the play.
    - Correct:

```
None
- name: Configure web server

  hosts: web

  tasks:
```

```
    - name: Install nginx
      ansible.builtin.package:
        name: nginx
        state: present
```

- Incorrect:

```
None
- hosts: web
  tasks:
    - name: Install nginx
      ansible.builtin.package:
        name: nginx
        state: present
```

Code (Incorrect → Correct):

```
None
# Incorrect: Play without a name
- hosts: all
  tasks:
    - name: Create placeholder file
      ansible.builtin.command: touch /tmp/.placeholder
```

```
None
# Correct: Play with a descriptive name

- name: Play for creating placeholder

  hosts: all

  tasks:

    - name: Create placeholder file

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

Benefits of Following Rule name[play]:

- Readability — makes playbooks easier to understand at a glance.
- Troubleshooting — simplifies debugging by providing meaningful context in output.
- Professionalism — enforces consistent and clear naming conventions.
- Collaboration — helps teams quickly identify what each play is intended to do.