Resolving Common Ansible Playbook Execution Problems

Ansible is an effective automation tool, though users might face several difficulties while using it. This document addresses common issues and their solutions.

1. Error 503: no-handler

Description:

Ansible-Lint Error 503 (`no-handler`) occurs when tasks are written to behave like handlers but are not structured as such. Instead of directly checking conditions like `when: result.changed`, best practice is to use **handlers** with the `notify` directive. Handlers provide a clean, structured way to run follow-up actions only when changes occur, improving both **readability and maintainability** of playbooks.

Symptoms:

- Linter flags violations such as:

```
None
no-handler: Tasks that run when changed should likely be
handlers.
```

- Playbook uses `when: result.changed` on a normal task to mimic handler behavior.
- Output may also include unrelated YAML formatting warnings (e.g., implicit octal values).

Resolution:

- Replace conditional follow-up tasks (`when: result.changed`) with a proper **handler**.
- Use the `notify` keyword in the main task to trigger the handler when a change occurs.
- Define handlers in a dedicated `handlers:` section for clarity.

Code

(Incorrect → Correct):

None

```
# Incorrect: Using when with result.changed instead of a handler
- name: Example of no-handler rule
  hosts: all
  tasks:
    - name: Register result of a task
      ansible.builtin.copy:
        dest: "/tmp/placeholder"
        content: "Ansible made this!"
        mode: 0600
      register: result
    - name: Second command to run
      ansible.builtin.debug:
        msg: The placeholder file was modified!
      when: result.changed
```

None

```
# Correct: Using notify and a handler
- name: Example of no-handler rule
  hosts: all
  tasks:
    - name: Register result of a task
      ansible.builtin.copy:
        dest: "/tmp/placeholder"
        content: "Ansible made this!"
        mode: "0600"
      notify:
        - Second command to run
  handlers:
    - name: Second command to run
      ansible.builtin.debug:
        msg: The placeholder file was modified!
```

Benefits of Using Handlers:

- Structure and Readability — makes it clear which tasks trigger follow-up actions.
- Efficiency — handlers run only when notified, avoiding unnecessary executions.
- Maintainability — separates normal tasks from conditional responses.
- Debugging — easier to trace why and when a handler was executed.

2. Error 504: deprecated-local-action

Description:
Ansible-Lint Error 504 (`deprecated-local-action`) is triggered when the `local_action` keyword is used in playbooks. While this method was once common for executing tasks on the control node (localhost), it is now **deprecated**. The recommended modern approach is to use `delegate_to: localhost`, which is more explicit, clear, and aligned with Ansible best practices.

Symptoms:

- Linter flags violations such as:

```
None
deprecated-local-action: Do not use 'local_action', use
'delegate_to: localhost'.
```

- Example violation appears in tasks where `local_action` is used.
- Playbooks may still run, but they are flagged as outdated and non-compliant.

Resolution:

- Replace `local_action` with `delegate_to: localhost`.
- Use the normal module syntax for tasks and explicitly delegate execution to localhost.
- Review playbooks to ensure all local actions follow this updated pattern.

Code

(Incorrect → Correct):

```
None
# Incorrect: Using deprecated local_action
- name: Example of deprecated-local-action rule
  hosts: all
  tasks:
    - name: Task example
      local_action:   # Deprecated
        module: ansible.builtin.debug
```

```
None
# Correct: Using delegate_to: localhost
- name: Example of deprecated-local-action rule
  hosts: all
  tasks:
    - name: Task example
      ansible.builtin.debug:
      delegate_to: localhost
```

Benefits of Using `delegate_to: localhost`:

- Alignment with Best Practices — ensures your playbooks remain compatible with modern Ansible standards.
- Improved Readability — makes it clear that the task runs on the control node.
- Maintainability — playbooks using current conventions are easier to update and maintain over time.
- Future-Proofing — avoids reliance on deprecated features that may be removed in future Ansible releases.

3. Error 505: missing-import

Description:

Ansible-Lint Error 505 (`missing-import`) occurs when a playbook references an **imported file, role, or variable file** that does not exist or is incorrectly specified. Imports in Ansible are used for modularity and reusability—such as including task files, playbooks, or variable files. If the reference is missing or incorrect, playbook execution fails, resulting in incomplete or malfunctioning automation.

Symptoms:

- Linter flags violations such as:

```
None
syntax-check[missing-file]: Unable to retrieve file contents

505.yml:1:1 Could not find or access 'non-existing.yml' on the
Ansible Controller.
```

- Playbook fails to run due to missing file references.
- Errors related to undefined variables or skipped tasks caused by missing role/task/vars imports.

Resolution:

- **Verify paths and filenames** in import statements.
    - Ensure included YAML, task, or variable files exist at the correct location.
- **Check role imports** and confirm required roles are properly declared and available.
- **Use correct import directives**:
    - `import_playbook` for other playbooks.
    - `include_tasks` or `import_tasks` for task files.
    - `import_vars` for variable files.
- **Organize playbook structure** with clear directories for tasks, roles, and variable files.
- **Use descriptive file names** to reduce ambiguity and misreferences.

Code

(Incorrect → Correct):

None

```
# Incorrect: Missing file reference

- name: Example of playbook

  hosts: all

  tasks:

    - name: Task example

      ansible.builtin.include: 'non-existing.yml'
```

None

```
# Correct: File reference fixed

- name: Example of playbook

  hosts: all

  tasks:

    - name: Task example

      import_tasks: tasks/example.yml   # Correct path to
existing file
```

None

```
# Correct: Role import example

- name: Example with role

  hosts: all

  roles:
```

```
    - myrole   # Role must exist in roles/ directory
```

```
None
# Correct: Variable import example

- name: Example with vars

  hosts: all

  tasks:

    - name: Import variables

      import_vars: vars/myvars.yml
```

Benefits of Avoiding Missing Imports:

- Reliability — ensures all referenced files are found and executed properly.
- Maintainability — modular, reusable playbook components are easier to manage.
- Clarity — descriptive imports make playbook intent clear to collaborators.
- Error Prevention — prevents wasted debugging time from broken or missing references.

4. Error 601: literal-compare

Description:
Ansible-Lint Error 601 (`literal-compare`) is triggered when a variable is explicitly compared to the Boolean literals `True` or `False` in a `when` condition. This comparison is redundant because Ansible already evaluates variables as Boolean values. The rule enforces a cleaner and more idiomatic way of writing conditions.

Symptoms:

- Linter reports:

```
None
literal-compare: Don't compare to literal True/False.

601.yml:5 Task/Handler: Ensure a task runs only in the production
environment
```

- Code example that triggers the error:

```
None
when: production == True
```

- Tasks still work, but linting fails and the code is unnecessarily verbose.

Resolution:

- Remove explicit comparisons to `True` or `False`.
    - Correct:

```
None
when: production
```

    - Incorrect:

```
None
when: production == True
```

- Use negation for `False` checks.
    - Correct:

```
None
when: not production
```

    - Incorrect:

```
None
when: production == False
```

- Apply this simplification consistently across all `when` conditions.

Code

(Incorrect → Correct):

```
None
# Incorrect: Redundant comparison to True
- name: Ensure production environment is configured
  hosts: all
  tasks:
    - name: Ensure a task runs only in the production environment
      ansible.builtin.debug:
        msg: "This is a production task"
      when: production == True
```

```
None
# Correct: Simplified and idiomatic condition
```

```
- name: Ensure production environment is configured
  hosts: all
  tasks:
    - name: Ensure a task runs only in the production environment
      ansible.builtin.debug:
        msg: "This is a production task"
      when: production
```

Benefits of Following Rule 601:

- Readability — cleaner, easier-to-understand playbooks.
- Consistency — aligns with Ansible best practices for conditions.
- Reduced Risk of Errors — avoids mistakes from overcomplicated expressions.
- Maintainability — makes playbooks simpler for teams to update and debug.

5. Error 602: empty-string-compare

Description:
Ansible-Lint Error 602 (`empty-string-compare`) highlights the use of **empty string comparisons** in `when` conditions. For example, `when: var == ""` or `when: var != ""`. These patterns are considered unclear and ambiguous. The rule enforces clearer alternatives using **length-based checks** (`| length > 0` or `| length == 0`) to improve readability and maintainability.

Symptoms:

- Linter flags conditional comparisons with empty strings.
- Example violation:

```
None
when: ansible_distribution == ""   # Compares with an empty
string
```

- Report may include:

```
None
empty-string-compare: Avoid comparing variables directly to empty
strings.
```

- Playbooks may still run, but style violations reduce clarity and consistency.

Resolution:

- **Replace empty string checks with length-based filters.**
    - Use `var | length > 0` instead of `var != ""`.
    - Use `var | length == 0` instead of `var == ""`.
- **Enable the rule in Ansible-lint** if not already:

```
None
enable_list:
  - empty-string-compare
```

- **Apply this consistently across all playbooks** for clarity and team-wide standards.

Code

(Incorrect → Correct):

```
None
# Incorrect: Uses empty string comparison

- name: Example playbook

  hosts: all

  tasks:

    - name: Start the service

      ansible.builtin.service:

        name: my-service

        state: started

      when: ansible_distribution == ""   # Not recommended
```

```
None
# Correct: Uses length-based comparison

- name: Example playbook

  hosts: all

  tasks:

    - name: Start the service

      ansible.builtin.service:

        name: my-service

        state: started

      when: ansible_distribution | length > 0   # Recommended
```

Benefits of Following Rule 602:

- Improved Clarity — conditions are explicit and easy to read.
- Consistency — standardizes style across your playbooks.
- Maintainability — reduces ambiguity and eases troubleshooting.
- Reliability — avoids misunderstandings when reviewing or extending automation.

6. Error 702: meta-no-tags

Description:
Ansible-Lint Error 702 (`meta-no-tags`) enforces naming conventions for **role metadata tags** in the `meta/main.yml` file of a role. Tags must consist of **only lowercase letters and digits**. Using uppercase letters or special characters in metadata tags creates inconsistency and confusion, and this rule prevents such usage.

Symptoms:

- Linter reports violations such as:

```
None
meta-no-tags: Tags must contain lowercase letters and digits
only., invalid: 'MyTag#1'

meta-no-tags: Tags must contain lowercase letters and digits
only., invalid: 'MyTag&^-'
```

- Other related violations may occur if the role name or schema is also invalid.
- Playbook execution itself may not fail, but linting stops further processing.

Resolution:

- Use only lowercase letters and digits in `galaxy_tags`.
    - Correct:

```
None
galaxy_tags: [mytag1, mytag2]
```

    - Incorrect:

```
None
galaxy_tags: [MyTag#1, MyTag&^-]
```

- Avoid uppercase characters, special symbols (`#`, `&`, `-`, `^`, etc.), and spaces.
- Validate your `meta/main.yml` file with `ansible-lint` after updates to ensure compliance.

Code

(Incorrect → Correct):

```
None
# Incorrect: Invalid tags
galaxy_info:
  author: Test
  description: test
  company: Test
  license: GPL-2.0-or-later
  min_ansible_version: 2.1
  galaxy_tags: [MyTag#1, MyTag&^-]   # Invalid tags
dependencies: []
```

```
None
# Correct: Valid tags

galaxy_info:

  author: Test

  description: test

  company: Test

  license: GPL-2.0-or-later

  min_ansible_version: "2.1"

  galaxy_tags: [mytag1, mytag2]      # Valid tags

dependencies: []
```

Benefits of Following Rule 702:

- Consistency — ensures uniform tag naming across all roles.
- Readability — metadata tags remain clear and unambiguous.
- Ease of Maintenance — consistent tags simplify long-term role management.
- Community Best Practice — aligns with Ansible Galaxy conventions, making your roles easier to share and reuse.

7. Error 703: meta-incorrect

Description:
Ansible-Lint Error 703 (`meta-incorrect`) checks that role **metadata fields in `meta/main.yml` are properly defined**. Certain fields such as `author`, `description`, `company`, and `license` should not be left with placeholder or default values. Roles without accurate metadata can appear incomplete, unprofessional, and harder to maintain.

Symptoms:

- Linter flags default or placeholder values in `meta/main.yml`:

```
None
meta-incorrect: Should change default metadata: author

meta-incorrect: Should change default metadata: company

meta-incorrect: Should change default metadata: license
```

- Other schema-related warnings may also appear if fields like `min_ansible_version` are incorrectly typed.
- Processing of the role metadata file may stop due to unskippable violations.

Resolution:

- Replace default placeholders (`your name`, `your role description`, `your company`, `license`) with meaningful values.
- Ensure the following metadata fields are **accurate and descriptive**:
    - `author` → your actual name or team name.
    - `description` → a concise explanation of the role's purpose.
    - `company` → your organization, or omit if not applicable.
    - `license` → a valid license identifier (e.g., `GPL-2.0-or-later`, `MIT`).
- Use strings for fields like `min_ansible_version`.

Code

(Incorrect → Correct):

```
None
# Incorrect: Default metadata values

galaxy_info:
```

```
  author: your name

  description: your role description

  company: your company (optional)

  license: license (GPL-2.0-or-later, MIT, etc)
```

```
None
# Correct: Properly defined metadata values

galaxy_info:

  author: Luca Berton

  description: This role will set you free.

  company: Ansible Pilot

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

Benefits of Following Rule 703:

- Clarity — metadata clearly communicates role ownership and purpose.
- Documentation — acts as built-in documentation for anyone using or maintaining the role.
- Professionalism — well-structured metadata reflects good practice and reliability.
- Community Readiness — properly filled metadata makes roles easier to share and adopt.

8. Error 704: meta-video-links

Description:
Ansible-Lint Error 704 (`meta-video-links`) enforces proper formatting of **video links in role**

metadata (`meta/main.yml`). Each entry under `video_links` must be a dictionary with exactly two keys:

- `url` — a supported video link (YouTube, Vimeo, or Google Drive shared link).
- `title` — a descriptive title for the video.

Using plain strings, unsupported keys, or invalid URL formats will trigger this error.

Symptoms:

- Linter reports violations such as:

```
None
meta-video-links: Expected item in 'video_links' to be a
dictionary

meta-video-links: Expected item in 'video_links' to contain only
keys 'url' and 'title'

meta-video-links: URL format 'www.acme.com/vid' is not
recognized. Expected it be a shared link from Vimeo, YouTube, or
Google Drive.
```

- Schema errors may also appear if required metadata fields are missing.
- Processing of the file stops until violations are fixed.

Resolution:

- Ensure all items in `video_links` are dictionaries with `url` and `title` keys.
- Use **valid shared links** from supported platforms (YouTube, Vimeo, or Google Drive).
- Avoid unsupported keys, plain strings, or malformed URLs.
- Always include required fields in `galaxy_info` to avoid additional schema violations.

Code

(Incorrect → Correct):

```
None
# Incorrect: Invalid video_links formatting

galaxy_info:

  video_links:

    - https://www.youtube.com/@AnsiblePilot/   # Missing 'url'
key

    - my_bad_key: https://www.youtube.com/@AnsiblePilot/   #
Unsupported key

      title: Incorrect key

    - url: www.acme.com/vid   # Invalid URL format

      title: Incorrect URL format
```

```
None
# Correct: Properly formatted video_links

galaxy_info:

  video_links:

    - url: https://www.youtube.com/@AnsiblePilot/

      title: Correctly formatted video link
```

Benefits of Following Rule 704:

- Consistency — structured video links make metadata uniform across roles.

- Enhanced Documentation — video links serve as clear references for role usage.
- Ease of Maintenance — standard formatting simplifies updates and collaboration.
- Community Alignment — conforms to Ansible Galaxy expectations, improving role sharing.

9. Error 911: syntax-check

Description:
Ansible-Lint Error 911 (`syntax-check`) is triggered when a playbook fails **basic syntax validation**. This check ensures playbooks can be parsed correctly before execution. Since this rule is **unskippable**, playbooks with syntax errors must be corrected before they can run.

Symptoms:

- Linter reports fatal syntax errors such as:

```
None
syntax-check[specific]: The field 'hosts' has an invalid value, which includes an undefined variable. The error was: 'my_hosts' is undefined.
```

- Playbook execution halts immediately.
- Common causes include:
    - Undefined variables used in key fields (`hosts`, `vars`, `tasks`).
    - Misformatted YAML (indentation errors, missing colons).
    - Incorrect module arguments or structure.

Resolution:

- Run a syntax check before execution:

```shell
ansible-playbook myplaybook.yml --syntax-check
```

- Ensure all required variables are defined or provide safe defaults.
    - Example:

```
hosts: "{{ my_hosts | default([]) }}"
```

- Validate YAML formatting with a linter such as `yamllint`.
- Fix indentation, colons, and key-value formatting errors.
- Re-run syntax checks until no violations are reported.

Code

(Incorrect → Correct):

```
# Incorrect: Undefined variable in hosts
- name:
    Bad use of variable inside hosts block
  hosts: "{{ my_hosts }}"   # Fails if my_hosts is not defined
  tasks: []
```

```
# Correct: Safe handling of variable with default filter
```

```
- name: Good use of variable inside hosts, without assumptions
  hosts: "{{ my_hosts | default([]) }}"
  tasks: []
```

Benefits of Following Rule 911:

- Reliability — ensures playbooks won't break at runtime due to syntax issues.
- Safety — prevents undefined variables and misconfigurations from propagating.
- Consistency — validates all playbooks adhere to proper YAML and Ansible structure.
- Prerequisite for Orchestration — guarantees that automation workflows start on a clean foundation.

10. Error: args

Description:
The Ansible-Lint **`args` rule** validates that task arguments match the **plugin's documentation** for each module. It ensures all required parameters are present, mutually dependent options are respected, and values are of the correct type. This prevents tasks from failing at runtime due to missing or invalid arguments.

Symptoms:

- Linter warnings such as:

```
None
args[module]: missing required arguments: repo

args[module]: missing parameter(s) required by 'enabled': name
```

- Tasks fail to run because required arguments are missing or incorrect.

- Invalid values are flagged (e.g., string instead of boolean).

Resolution:

- Review the module's documentation and ensure all **required parameters** are provided.
- Check for **parameter dependencies** (e.g., if `enabled` is used in `systemd`, `name` must also be provided).
- Verify that **values are of the correct type** (boolean, string, integer, etc.).
- In special cases (e.g., Jinja expressions), if the linter cannot validate arguments, you may bypass the rule with:

```
None
# noqa: args[module]
```

Code

(Incorrect → Correct):

```
None
# Incorrect: Missing required arguments and invalid values
- name: Clone content repository
  ansible.builtin.git:
    dest: /home/www
    accept_hostkey: true
    version: master
    update: false    # Missing required 'repo'


- name: Enable service httpd and ensure it is not masked
```

```
  ansible.builtin.systemd:
    enabled: true
    masked: false    # Missing required 'name'

- name: Use quiet to avoid verbose output
  ansible.builtin.assert:
    test:
      - my_param <= 100
      - my_param >= 0
    quiet: invalid   # Invalid value type
```

None

```
# Correct: Required arguments included and valid values used
- name: Clone content repository
  ansible.builtin.git:
    repo: https://github.com/ansible/ansible-examples
    dest: /home/www
    accept_hostkey: true
    version: master
    update: false
```

```
- name: Enable service httpd and ensure it is not masked
  ansible.builtin.systemd:
    name: httpd
    enabled: false
    masked: false


- name: Use quiet to avoid verbose output
  ansible.builtin.assert:
    that:
      - my_param <= 100
      - my_param >= 0
    quiet: true      # Correct type (boolean)
```

Benefits of Following the args Rule:

- Ensures **correct module usage** by matching arguments with official documentation.
- Prevents **missing parameters** that could cause incomplete or failed tasks.
- Detects **invalid values**, avoiding misconfiguration.
- Promotes **best practices** and consistent, reliable playbooks.
- Provides flexibility with `# noqa: args[module]` when validation is not possible.