Troubleshooting Ansible Playbook Execution

Ansible is a powerful automation tool, but users may encounter issues. This document outlines common problems and their solutions.

1.Error: sanity

Description:
The `sanity` rule checks the **`ignore-x.x.txt` file** in Ansible projects. This file defines rules or tests that developers have explicitly chosen to ignore. While some ignores are necessary, not all are allowed. The `sanity` rule enforces that only **permitted ignores** are used and that they follow the correct format. This rule is especially important for projects aiming for **Red Hat Certification**, where strict standards must be met.

Symptoms:

- Linter reports violations such as:

```
None
sanity[cannot-ignore]: Found ignore that is not permitted
sanity[bad-ignore]: Malformed ignore entry in ignore-x.x.txt
```

- Playbooks may not pass Red Hat certification checks.
- The `ignore-x.x.txt` file contains unapproved or incorrectly formatted entries.

Allowed Ignores (current list):

- `validate-modules:missing-gplv3-license`
- `action-plugin-docs`
- `import-2.6`, `import-2.6!skip`
- `import-2.7`, `import-2.7!skip`
- `import-3.5`, `import-3.5!skip`

- `compile-2.6`, `compile-2.6!skip`
- `compile-2.7`, `compile-2.7!skip`
- `compile-3.5`, `compile-3.5!skip`
- `shellcheck`
- `shebang`
- `pylint:used-before-assignment`

Resolution:

1. Review the **`ignore-x.x.txt` file** in your project.
2. Remove or replace any ignores not in the permitted list.
3. Correct formatting errors in valid ignores.
4. Rerun linting to confirm compliance.

Code (Incorrect → Correct):

```
None
# Incorrect: Disallowed ignore
random-check:skip
```

```
None
# Correct: Allowed ignore
import-2.7!skip
```

Benefits of Following the sanity Rule:

- Compliance — ensures compatibility with Red Hat Certification requirements.

- Quality Assurance — maintains high standards for playbook and role development.
- Reliability — prevents misuse of ignores that hide real issues.
- Consistency — enforces uniform best practices across the Ansible community.

2.Error: Failed to connect to the host via SSH: localhost port 22

Description:
This error occurs when Ansible attempts to connect to `localhost` over SSH on port 22, but the connection fails. By default, Ansible assumes SSH connections to managed hosts. When running playbooks locally, this leads to errors unless explicitly configured to use a **local connection** instead of SSH.

Symptoms:

- Error message in terminal:

```
None
fatal: [localhost]: UNREACHABLE! => {"changed": false, "msg":
"Failed to connect to the host via SSH: ssh: connect to host
localhost port 22: Connection refused", "unreachable": true}
```

- Playbook execution halts with the target `localhost` marked as unreachable.
- Typically appears when testing playbooks on the same machine (local development).

Resolution:

1. **Update inventory file** to specify that `localhost` should use a local connection:

```
None
[local]
localhost ansible_connection=local
```

2. **Use ansible_connection=local** directly in playbook if needed.

3. **Verify Python interpreter path** warnings if they appear, but these do not block execution.

Code (Incorrect → Correct):

```
None
# Incorrect: Default behavior tries SSH on port 22
- name: ping module Playbook
  hosts: all
  tasks:
    - name: test connection
      ansible.builtin.ping:
```

```
None
# Correct: Inventory explicitly sets local connection
[local]
localhost ansible_connection=local
```

```
None
# Correct: Playbook runs successfully with local connection
- name: ping module Playbook
  hosts: all
  tasks:
    - name: test connection
      ansible.builtin.ping:
```

Benefits of Fixing This Error:

- Ensures successful playbook execution on the local machine.
- Removes dependency on SSH when managing localhost.
- Provides predictable behavior when testing automation locally.
- Aligns with best practices for inventory configuration.

3.Error: Invalid plugin name: regex.replace

Description:
This error appears when a playbook uses the legacy filter name `regex.replace` and Ansible cannot load it. In modern Ansible, the correct filter is `ansible.builtin.regex_replace`. The error can also occur if you're running an older Ansible version that lacks support for the expected filter, or if another plugin conflicts with filter loading.

Symptoms:

- Playbook fails during templating with a message similar to:

```
None
template error while templating string: Could not load
"regex.replace": 'invalid plugin name: regex.replace'
```

- Task using `{{ value | regex.replace(...) }}` fails; recap shows one failed task.

Resolution:

1. Use the correct filter name in current releases: **`ansible.builtin.regex_replace`**.
2. If you must support very old Ansible (≤2.9), verify which filter names are available and adjust accordingly.
3. Upgrade Ansible to a supported version to avoid legacy naming mismatches:

```
Shell
pip install --upgrade ansible
```

4. Check for conflicting custom plugins or filter plugins. Temporarily disable nonessential plugins to rule out conflicts.
5. Validate playbook syntax after changes.

Code (Incorrect → Correct):

```
None
# Incorrect: uses legacy filter name that cannot be loaded
- name: Regex Playbook
  hosts: all
  vars:
    example: '<HTML>example'
  tasks:
    - name: Regex
      ansible.builtin.debug:
        msg: "{{ example | regex.replace('<.*?>') }}"
```

```
None
# Correct: uses the supported built-in filter name
- name: Regex Playbook
  hosts: all
  vars:
    example: '<HTML>example'
  tasks:
    - name: Regex
      ansible.builtin.debug:
        msg: "{{ example | ansible.builtin.regex_replace('<.*?>')
}}"
```

Notes and tips:

- The filter removes HTML-like tags in the example: input `"<HTML>example"` becomes `"example"`.
- Keep Ansible and collections up to date to minimize filter and plugin name drift.
- If you maintain shared roles, document minimum Ansible versions to prevent naming mismatches across environments.

4.Error: Kubernetes K8s/OpenShift OCP 401 Unauthorized

Description:
The **401 Unauthorized** error occurs when Ansible attempts to interact with a Kubernetes (K8s) or OpenShift (OCP) cluster without valid authentication credentials. This error is not caused by Ansible itself but by missing or invalid Kubernetes authentication tokens or misconfigured contexts in your kubeconfig.

Symptoms:

- Fatal error during execution:

```
None
"msg": "Namespace example: Failed to retrieve requested object...
\"reason\":\"Unauthorized\",\"code\":401"
```

- Running `oc get namespace` or `kubectl get namespace` also fails with:

```
None
error: You must be logged in to the server (Unauthorized)
```

- Playbooks attempting to use `kubernetes.core.k8s` or related modules fail immediately.

Resolution:

1. **Check login status:**
    - Run `oc whoami` or `kubectl config view --minify --flatten` to verify if you are authenticated.
2. **Log in to the cluster:**
    - For OpenShift CRC:

```shell
eval $(crc oc-env)
oc login -u kubeadmin https://api.crc.testing:6443
```

- Use appropriate credentials (username and password or token).

3. **Verify kubeconfig context:**
   - Ensure `~/.kube/config` has the correct user, token, and context.
   - Confirm the current context matches your intended cluster and namespace.
4. **Re-run playbook with valid credentials:**
   Once logged in, rerun your playbook:

```shell
ansible-playbook kubernetes/namespace.yml
```

Code (Incorrect → Correct):

```none
# Incorrect: Unauthorized because no valid credentials
- name: k8s Playbook
  hosts: localhost
  gather_facts: false
  connection: local
  vars:
    myproject: "example"
  tasks:
    - name: create {{ myproject }} namespace
      kubernetes.core.k8s:
        api_version: v1
        kind: Namespace
        name: "{{ myproject }}"
        state: present
```

```
# Fails with 401 Unauthorized
```

```
None
# Correct: After logging in with valid kubeconfig context
- name: k8s Playbook
  hosts: localhost
  gather_facts: false
  connection: local
  vars:
    myproject: "example"
  tasks:
    - name: create {{ myproject }} namespace
      kubernetes.core.k8s:
        api_version: v1
        kind: Namespace
        name: "{{ myproject }}"
        state: present
# Succeeds once authentication is established
```

Benefits of Resolving 401 Unauthorized:

- Predictable access to Kubernetes or OpenShift clusters.
- Ensures Ansible modules can interact with cluster resources securely.
- Prevents wasted time troubleshooting playbooks when the issue lies in authentication.
- Maintains compliance by requiring valid credentials for automation.

5.Module Failure on Windows-target

Description:
This failure happens when a **Linux/Unix module** (which expects Python on the target) is used against a **Windows host**. Windows hosts run Ansible modules via **PowerShell** and require Windows-specific modules (the `ansible.windows` collection). Using

`ansible.builtin.get_url` on Windows leads Ansible to push Python-oriented module code that PowerShell cannot parse, causing a module crash.

Symptoms:

- Task fails on a Windows host with a **MODULE FAILURE** and PowerShell parse errors.
- Warning about **no Python interpreters** on the Windows host.
- Example error (abridged):

```
None
[WARNING]: No python interpreters found for host WindowsServer
fatal: [WindowsServer]: FAILED! => {"msg": "MODULE FAILURE",
"module_stderr": "... ParseException ..."}
```

Resolution:

1. **Use the Windows module variant** from `ansible.windows` (PowerShell-based) instead of the Unix module.
    - Correct module for downloading files on Windows: `ansible.windows.win_get_url`
    - Incorrect on Windows: `ansible.builtin.get_url`
2. **Keep your Windows inventory correct** (WinRM connection, credentials), e.g.:

```
None
[windows]
WindowsServer

[windows:vars]
ansible_connection=winrm
ansible_user=Administrator
ansible_password=YourPassword
ansible_winrm_transport=basic
```

```
ansible_winrm_server_cert_validation=ignore
```

3. **Avoid `become` on Windows** unless you know you need it; Windows privilege elevation differs from sudo on Unix.

4. **Re-run the play** after switching to the proper Windows module.

Code (Incorrect → Correct):

None
```
# Incorrect: Unix module used on a Windows target
- name: win_get_url module Playbook
  hosts: all
  become: false
  vars:
    myurl:
"https://releases.ansible.com/ansible/ansible-2.9.25.tar.gz"
    mydest: 'C:\Users\vagrant\Desktop\ansible-2.9.25.tar.gz'
  tasks:
    - name: download file
      ansible.builtin.get_url:
        url: "{{ myurl }}"
        dest: "{{ mydest }}"
```

None
```
# Correct: Windows module from ansible.windows collection
- name: win_get_url module Playbook
  hosts: all
  become: false
  vars:
    myurl:
"https://releases.ansible.com/ansible/ansible-2.9.25.tar.gz"
    mydest: 'C:\Users\vagrant\Desktop\ansible-2.9.25.tar.gz'
  tasks:
```

```
    - name: download file
      ansible.windows.win_get_url:
        url: "{{ myurl }}"
        dest: "{{ mydest }}"
```

Why this works:
`ansible.windows.win_get_url` delivers a PowerShell-based module payload that Windows can execute, avoiding Python dependency and parse errors.

Optional checks:

- Confirm you have the Windows collection:

```
Shell
ansible-galaxy collection install ansible.windows
```

- Validate WinRM connectivity before running the play:

```
Shell
ansible windows -i inventory -m ansible.windows.win_ping
```

6.Error: Permission denied [Errno 13]

Description:
The `Permission denied [Errno 13]` error occurs when Ansible attempts to perform an action that requires elevated privileges, but the current user does not have the necessary write permissions. This typically happens when modifying system files (for example, `/etc/environment`) without enabling privilege escalation.

Symptoms:

- Playbook execution fails with messages like:

```
None
PermissionError: [Errno 13] Permission denied:
b'/etc/.ansible_tmp_xxxxx'
```

- The error explicitly states that the **destination directory is not writable by the current user**.
- The task fails even though the syntax is correct, because permissions are insufficient.

Resolution:

1. **Enable privilege escalation with `become: true`.**
    - This allows Ansible to run the task as a privileged user (typically root).
2. **Confirm the target file requires root access.**
    - Files under `/etc/` almost always need elevated privileges.
3. **Test the fix by re-running the playbook.**
    - Ensure the task executes without errors and applies the intended changes.

Code (Incorrect → Correct):

```
None
# Incorrect: Missing privilege escalation
- name: set environment Playbook
  hosts: all
  gather_facts: false
  vars:
    os_environment:
      - key: EDITOR
        value: vi
  tasks:
    - name: customize /etc/environment
      ansible.builtin.lineinline:
```

```
        dest: "/etc/environment"
        state: present
        regexp: "^{{ item.key }}="
        line: "{{ item.key }}={{ item.value }}"
      with_items: "{{ os_environment }}"
```

None

```
# Correct: Added become: true for privilege escalation
- name: set environment Playbook
  hosts: all
  gather_facts: false
  become: true
  vars:
    os_environment:
      - key: EDITOR
        value: vi
  tasks:
    - name: customize /etc/environment
      ansible.builtin.lineinline:
        dest: "/etc/environment"
        state: present
        regexp: "^{{ item.key }}="
        line: "{{ item.key }}={{ item.value }}"
      with_items: "{{ os_environment }}"
```

Benefits of Using Privilege Escalation Correctly:

- Security — ensures only authorized tasks gain elevated privileges.
- Reliability — prevents failures when modifying system files.
- Predictability — tasks behave consistently across environments.
- Simplicity — avoids workarounds such as manual file edits.

7.Error: Syntax Errors

Description:
Ansible syntax errors occur when a playbook is not valid YAML or JSON. These errors typically happen because of **missing quotes, unbalanced brackets, improper indentation, or incomplete strings**. Since Ansible playbooks are written in YAML, even small mistakes can cause the linter or playbook runner to fail.

Symptoms:

- Execution stops immediately with messages like:

```
None
ERROR! We were unable to read either as JSON nor YAML
Syntax Error while loading YAML.
found unexpected end of stream
```

- Ansible points to the offending line but may also note that the actual error could be elsewhere.
- Example indicator:

```
None
src: "{{ source }}"
dest: "{{ destination }}
              ^
Missing closing quote
```

Resolution:

1. **Quote Jinja2 template values** — always use `"{{ variable }}"` instead of bare `{{ variable }}`.
2. **Check for unbalanced or missing quotes.**
    - Correct: `dest: "{{ destination }}"`
    - Incorrect: `dest: "{{ destination }}`

3. **Validate playbooks with `ansible-playbook --syntax-check`.**

4. **Use a YAML linter** (e.g., `yamllint`) to spot common indentation and formatting mistakes.

Code (Incorrect → Correct):

None

```
# Incorrect: Missing closing quote in Jinja2 template
- name: win_copy module Playbook
  hosts: all
  become: false
  gather_facts: false
  vars:
    source: "report.txt"
    destination: "Desktop/report.txt"
  tasks:
    - name: copy report.txt
      ansible.windows.win_copy:
        src: "{{ source }}"
        dest: "{{ destination }}
```

None

```
# Correct: Properly quoted Jinja2 template
- name: win_copy module Playbook
  hosts: all
  become: false
  gather_facts: false
  vars:
    source: "report.txt"
    destination: "Desktop/report.txt"
  tasks:
    - name: copy report.txt
      ansible.windows.win_copy:
        src: "{{ source }}"
        dest: "{{ destination }}"
```

Benefits of Fixing Syntax Errors:

- Predictable playbook execution without crashes.
- Improved readability by following YAML quoting rules.
- Easier troubleshooting when errors occur, since task names and variables remain valid.
- Ensures compatibility with Ansible's YAML parser and prevents early termination of runs.

8.Error: This command has to be run under the root user

Description:
This error occurs when an Ansible module attempts to perform an operation that requires elevated privileges (such as installing packages, modifying system files, or managing services), but the playbook is **not running with root privileges**. By default, Ansible tasks run as the remote user specified in the inventory, unless explicitly escalated.

Symptoms:

- Execution stops with a fatal error:

```
None
FAILED! => {"changed": false, "msg": "This command has to be run
under the root user.", "results": []}
```

- Tasks involving package management, service management, or system-level changes fail.
- Other tasks that do not require privileges may still succeed.

Resolution:

1. Enable privilege escalation by setting `become: true` at the play or task level.
2. Ensure the Ansible control user has sudo access configured on the target system.
3. If necessary, specify the `become_user` (for example, `root`).
4. Confirm that `ansible.cfg` or inventory settings do not override privilege escalation.

Code (Incorrect → Correct):

```
None
# Incorrect: Privilege escalation disabled
- name: Troubleshooting under the root user
  hosts: all
  become: false
  tasks:
    - name: rsync installed
      ansible.builtin.package:
        name: rsync
        state: present
```

```
None
# Correct: Privilege escalation enabled
- name: Troubleshooting under the root user
  hosts: all
  become: true
  tasks:
    - name: rsync installed
      ansible.builtin.package:
        name: rsync
        state: present
```

Benefits of Fixing Root Privilege Errors:

- Ensures package installations, system updates, and service management tasks succeed.
- Prevents inconsistent playbook runs caused by missing privileges.
- Improves security by making privilege escalation explicit and controlled.
- Increases reliability and predictability of automation workflows.

9.Unhandled Exception While Executing Module `win_user`

Description:
When creating or updating a Windows local user with `ansible.windows.win_user`, the task can fail with an *"Unhandled exception while executing module"* message. The most common root cause is that the supplied password **does not satisfy the Windows password policy** (minimum length, complexity, history, etc.). The exception originates from the underlying PowerShell call (`SetPassword`) that enforces local/domain policy.

Symptoms:

- Task fails on `win_user` with an exception similar to:

```
None
Unhandled exception while executing module:
Exception calling "SetPassword" with "1" argument(s):
"The password does not meet the password policy requirements.
Check the minimum password length, password complexity and
password history requirements."
```

- Play recap shows `failed=1` for the target Windows host.
- Re-running with `-vvv` shows traceback around a `SetPassword` call.

Resolution:

1. **Provide a compliant password** that meets the machine's policy. Typical complexity requirements include:
    - Minimum length (often 8 characters or more).
    - Mix of character types (uppercase, lowercase, digits, symbols).
    - Not matching recent password history, not containing the username.
2. **Verify the current password policy** on the target host:
    - Local machine (non-domain):

```
None
net accounts
```

- Domain-joined (from a DC or with RSAT):

```
None
Get-ADDefaultDomainPasswordPolicy
```

3. **Adjust the password** in your playbook to conform to policy.
4. (Optional) **Change the policy** if you control the environment and a different policy is desired (domain: `Set-ADDefaultDomainPasswordPolicy`; local: adjust Local Security Policy). Do this with great care and approvals.
5. **Re-run the playbook** after updating the password (or policy).

Code (Incorrect → Correct):

```
None
# Incorrect: password too weak / violates policy
- name: windows user add
  hosts: all
  vars:
    usr_name: 'example'
    usr_password: 'password'
  tasks:
    - name: create local user
      ansible.windows.win_user:
        name: "{{ usr_name }}"
        password: "{{ usr_password }}"
```

```
None
# Correct: password meets typical complexity requirements
```

```
- name: windows user add
  hosts: all
  vars:
    usr_name: 'example'
    usr_password: 'NRns@bOFJNyX'  # Example: length and mixed
character types
  tasks:
    - name: create local user
      ansible.windows.win_user:
        name: "{{ usr_name }}"
        password: "{{ usr_password }}"
```

Optional checks (PowerShell) you can run via Ansible or interactively:

```
None
# Local machine policy overview
net accounts

# For domain environments (requires AD module)
Get-ADDefaultDomainPasswordPolicy
```

Notes and good practices:

- Do not hardcode real secrets in playbooks. Use **Ansible Vault** or a secrets manager to store `usr_password`.
- For repeatable runs, consider additional `win_user` parameters as needed (for example, `password_never_expires`, `update_password`, `user_cannot_change_password`) but keep in mind these do not bypass policy.
- If the password includes the username or common words, Windows may still reject it even if it looks complex.

10. Error: VARIABLE IS NOT DEFINED! ansible_hostname

Description:
This error occurs when a task references the `ansible_hostname` variable, but **facts are not gathered**. Facts are system information (like hostname, OS, interfaces) that Ansible collects automatically during playbook execution. If `gather_facts` is disabled, variables like `ansible_hostname` are unavailable, causing the “VARIABLE IS NOT DEFINED!” error.

Symptoms:

- Execution produces:

```
JSON
"ansible_hostname": "VARIABLE IS NOT DEFINED!"
```

- Task runs but does not print the expected hostname.
- Typically happens when `gather_facts: false` is set in the playbook.

Resolution:

1. **Enable fact gathering** by setting `gather_facts: true` at the play or task level.
2. **If you want facts but disabled global gathering**, use the `setup` module explicitly:

```
None
- name: Collect facts
  ansible.builtin.setup:
```

3. **If facts are unnecessary**, avoid referencing fact-based variables such as `ansible_hostname`.

Code (Incorrect → Correct):

```
None
# Incorrect: Facts disabled, variable unavailable
- name: hostname Playbook
```

```
  hosts: all
  gather_facts: false
  tasks:
    - name: print hostname
      ansible.builtin.debug:
        var: ansible_hostname
```

```
None
# Correct: Facts enabled, variable available
- name: hostname Playbook
  hosts: all
  gather_facts: true
  tasks:
    - name: print hostname
      ansible.builtin.debug:
        var: ansible_hostname
```

Benefits of Fixing Fact-Related Errors:

- Ensures variables like `ansible_hostname`, `ansible_distribution`, and `ansible_memtotal_mb` are available.
- Provides consistent, reliable access to host-specific data.
- Prevents confusion when debugging tasks that depend on Ansible facts.
- Supports richer automation by leveraging system details.