Document Version: 1.0
Last Updated: Mar 19, 2026
Applies To: Ansible Automation Platform 2.6+, Partner Collections
Intended Audience: Red Hat Partners, Content Developers, Collection Maintainers

Implementing Indirect Node Counting and Resource Taxonomy

Red Hat partners and content teams may use this guide to implement Indirect Node Counting in Ansible Collections. Indirect Node Counting ensures that managed resources are accurately counted, categorized, and reported.

1. Overview

Ansible modules often interact with external systems—such as clouds, hypervisors, network controllers—to manage nodes that are not present in the Ansible inventory. The Indirect Node Counting feature identifies and categorizes these "indirect" nodes from module execution results. To ensure consistency across vendors, this feature uses a Normalized Resource Taxonomy.

1.1 Prerequisites

Before implementing Indirect Node Counting, ensure you have:

Knowledge Requirements:

- Familiarity with Ansible collection development
- Understanding of YAML syntax
- Basic knowledge of `jq` expressions
- Experience with Ansible module return values

Access Requirements:

- Write access to the collection repository
- Ability to run test playbooks against target platforms

Software Requirements:

- Ansible Automation Platform 2.6 or later
- `jq` command-line tool (for testing expressions)
- Access to the target platform (AWS, Azure, VMware, etc.)

2. The Normalized Resource Taxonomy

Map vendor-specific resource types to the standardized taxonomy for accurate reporting and billing. When defining your query files, you must map the vendor-specific resource type to the

corresponding Normalized Device Type.

2.1 Taxonomy Structure

The taxonomy organizes resources into Categories (Level 1) and Device Types (Level 2).

2.2 Standard Device Types

When filling the `facts.device_type` field in your query, use the `snake_case` version of the `device_type` listed below.

Compute

Resource	**Standard** `device_type` **Value**
Virtual Machines	`virtual_machine`
Containers (Managed)	`container`
Hypervisors	`hypervisor`
Bare Metal	`bare_metal`
Serverless Functions	`serverless_function`
Auto Scaling Groups	`auto_scaling_group`

Networking

Resource	**Standard** `device_type` **Value**
Switches	`switch`
Routers	`router`
Firewalls	`firewall`

Load Balancers	`load_balancer`
Virtual Private Clouds	`vpc`
Subnets	`subnet`
Virtual Private Networks (VPNs)	`vpn`
Gateways	`gateway`
Domain Name System (DNS) Services	`dns_service`
Wireless Access Points	`wireless_access_point`
SD-WAN	`sd_wan`

Storage

Resource	**Standard** `device_type` **Value**
Object Storage	`object_storage`
Block Storage	`block_storage`
File Storage	`file_storage`
Archive Storage	`archive_storage`

Database

Resource	**Standard** `device_type` **Value**
Relational (SQL)	`database_relational`

NoSQL	`database_nosql`
Data Warehouse	`data_warehouse`
In-Memory/Cache	`database_cache`

DevOps & App Integration

Resource	**Standard `device_type` Value**
Continuous Integration/Continuous Deployment (CI/CD) Platforms	`ci_cd_platform`
Container Registries	`container_registry`
Message Queues	`message_queue`
API Endpoints	`api_endpoint`

NOTE: If your resource does not fit one of these standard types, consult the full Taxonomy Register or contact the Partner Engineering team.

3. Create Query Files

Query files are YAML documents that define how to extract data from module return values using expressions.

3.1 File Locations & Types

- **Embedded query files** (Recommended for Partners):
  - Ship these inside your collection.
  - Path: `extensions/audit/event_query.yml`
  - These take precedence over external queries.
- **External query files** (Red Hat Internal/Override):
  - Path: `extensions/audit/external_queries/<namespace>.<name>.<version>.y`

`ml`

3.2 File Structure

The file maps the Fully Qualified Collection Name (FQCN) to a query:

```shell
---
# Example: extensions/audit/event_query.yml
community.vmware.vmware_guest:
  query: >-
    {
      name: .instance.hw_name,
      canonical_facts: {
        host_name: .instance.hw_name,
        uuid: .instance.hw_product_uuid
      },
      facts: {
        device_type: "virtual_machine",
        guest_id: .instance.hw_guest_id
      }
    }
```

4. Expression Requirements

The expression processes the module's standard return data (the res field in Ansible events). It must output a JSON object with the following fields:

4.1 Input Data

The input is the module's returned JSON payload. Run your module with `-vvv` or check documentation to see the exact structure of the return values.

4.2 Output Fields

Field	Requirement	Description
`name`	Required	The human-readable display name of the node (for example, VM name, Switch hostname).

`canonical_facts`	Required	A dictionary of facts used to uniquely identify and deduplicate the node across job runs.
`facts`	Optional	Metadata for categorization. Must include `device_type`.

`canonical_facts` (Deduplication)

Select fields that are globally unique and stable for the life of the resource.

- **Recommended**: UUIDs, Serial Numbers, Permanent MAC addresses.
- **Conditional**: Hostnames (if unique in environment), IP addresses (static only).
- **Avoid**: Dynamic IPs, Random session IDs.

```shell
"canonical_facts": {
  "uuid": "550e8400-e29b-41d4-a716-446655440000",
  "serial_number": "XYZ-123"
}
```

`facts` (Categorization)

Apply the Normalized Resource Taxonomy.

- `device_type`: Map your resource to the standard values defined in section 2. The Normalized Resource Taxonomy (for example, `virtual_machine`, `switch`).
- `platform`: (Optional) The underlying platform (for example, `aws`, `vmware`, `azure`).

```shell
"facts": {
  "device_type": "virtual_machine",
  "platform": "vmware",
  "guest_id": "rhel8_64"
}
```

5. Vendor-Specific Strategies & Data Mapping

Different platforms require different data mapping strategies based on their API hierarchy.

5.1 Azure (Hierarchical)

Azure resources are hierarchical and use verbose Resource IDs.

Example ID:

```
Shell
/subscriptions/<sub_id>/resourceGroups/<rg>/providers/Microsoft.Compute/virtual
Machines/myvm
```

Because the full Azure resource ID (`$data.id`) contains the resource type, you can use regex to dynamically capture the type. The regex captures the string immediately after `Microsoft.` and before the next `/`.

Example expression:

```
Shell
($data.id | capture("/providers/[Mm]icrosoft.(?<resourcetype>[^/]+)/")? |
.resourcetype) | ascii_downcase
```

In this example, `Microsoft.Compute -> captures Compute -> lowercase -> "compute"`.

5.2 VMware (Flat)

VMware has a flat structure where the module returns only the top-level type. The Node type (`$node_type`) comes from the top-level key in the returned data: "`vm`", "`host`", "`cluster`", or "`vcsa`".

There is no sub-node type in VMware data. The `device_type` is derived directly using a mapping array, for example: "`vm`" -> "`virtual_machine`".

- Mapping Level: Single level (Top-level key -> `device_type`).

5.3 AWS (Implied / Catch-All)

AWS does not include a canonical "node type" in the resource ID (for example, `i-0123456789abcdef0`, `vpc-1234abcd`). The type is implied by the module you queried.

Example:

- `ec2_instance_info` implies -> `ec2_instance` (Mapped to Compute taxonomy)
- `rds_instance_info` implies -> `rds_instance` (Mapped to Database taxonomy)

To manage this, define an internal mapping dictionary similar to Azure:

```shell
{
  "ec2_instance": "virtual_machine",
  "rds_instance": "database_relational",
  "ec2_vpc_net": "vpc"
}
```

5.4 AWS Data Extraction Methodology (ACA-4453)

Use the following table to map AWS info modules to their respective resource types and keys. For all AWS queries, extract the resource name, ID, status, and tags for canonical facts.

AWS Module Resource Mapping Table:

Resource Category	AWS Info Module	NodeQuery Resource Type	Returned Data (Key)	Key Fields Extracted
Compute	`ec2_instance_info`	`ec2_instance`	`.instances` (list)	`instance_id -> id; state -> status; name, tags`
Database	`rds_instance_info`	`rds_instance`	`.instances` (list)	`db_instance_identifier -> id;`

				`db_name -> name; db_instance_status -> status; tags`
Database	`rds_cluster_info`	`rds_cluster`	`.clusters` (list)	`db_cluster_resource_id -> id; db_cluster_identifier -> name; status, tags`
Storage	`s3_bucket_info`	`s3_bucket`	`.buckets` (list)	`name; bucket_tagging -> tags`
Storage	`s3_object_info`	`s3_object`	`.s3_keys` (list) > `.object_info`	`object_name -> name; object_tagging -> tags`
Networking	`elb_application_lb_info`	`elb_application_lb`	`.load_balancers` (list)	`load_balancer_name -> name; state -> status; tags`
Networking	`elb_classic_lb_info`	`elb_classic_lb`	`.elbs` (list)	`load_balancer_name -> name; instances_inservice; tags`

Networking	`ec2_vpc_net_info`	`ec2_vpc_net`	`.vpcs` (list)	`vpc_id -> id; state -> status; tags`
Networking	`ec2_vpc_subnet_info`	`ec2_vpc_subnet`	`.subnets` (list)	`id; state -> status; tags`
Networking	`ec2_vpc_nat_gateway_info`	`ec2_vpc_nat_gateway`	`.result` (list)	`nat_gateway_id -> id; state -> status; tags`
Networking	`ec2_vpc_igw_info`	`ec2_vpc_igw`	`.internet_gateways` (list)	`internet_gateway_id -> id; attachments/state -> status; tags`
Networking	`ec2_vpc_vgw_info`	`ec2_vpc_vgw`	`.virtual_gateways` (list)	`vpn_gateway_id -> id; state -> status; tags`
Networking	`ec2_vpc_route_table_info`	`ec2_vpc_route_table`	`.route_tables` (list)	`id; tags`
Networking	`ec2_vpc_peering_info`	`ec2_vpc_peering`	`.vpc_peering_connections` (list)	`vpc_peering_connection_id -> id; status; tags`

Networking	`ec2_vpc_vpn_info`	`ec2_vpc_vpn`	`.vpn_connections` (list)	`vpn_connection_id -> id; state -> status; tags`

6. Module Matching Rules

6.1 Exact Match

Target a specific module. This is the recommended method.

```
Shell
cisco.ios.ios_facts:
  query: >-
    ...
```

6.2 Wildcard Match

Target all modules in a collection. Use carefully, as different modules return different data structures.

```
Shell
community.vmware.*:
  query: >-
    ...
```

7. External Query Versioning (Reference)

NOTE: This section applies primarily to the `redhat.indirect_accounting` collection.

External query files follow the naming convention:

```
Shell
<namespace>.<collection_name>.<version>.yml
```

Example: `amazon.aws.7.2.1.yml`
Fallback Logic:

If an exact version match isn't found, the system selects:

1. Same Major version.
2. Highest version less than or equal to the installed collection version.

8. Testing Your Query

Before publishing, verify your expression against real module output.

1. **Generate Output**: Run a playbook with your module and capture the JSON output (register the variable and debug: `var=result`).
2. **Test**: Use the command line to verify the transformation.

```shell
# Example Module Output
echo '{"instance": {"hw_name": "prod-db-01", "hw_uuid": "abc-123",
"hw_power_status": "on"}}' > module_output.json

# Test your query
cat module_output.json | jq '{
  name: .instance.hw_name,
  canonical_facts: {
    uuid: .instance.hw_uuid
  },
  facts: {
    device_type: "virtual_machine",
    power_state: .instance.hw_power_status
  }
}'
```

3. **Validate**: Ensure the output JSON contains valid name and `canonical_facts` keys.

9. Troubleshooting

Issue	Check
Query not applied	Verify file path is

	`extensions/audit/event_query.yml`. Check if the module FQCN matches exactly.
Node count is zero	Check output for syntax errors. Ensure `canonical_facts` are not `null/empty`.
Incorrect Category	Verify `facts.device_type` matches the standard taxonomy list.

Ansible partner announcements

Subscribe to the Ansible partner announcements list for periodic updates from the Ansible partner team. Email your subscription request to ansiblepartners@redhat.com.

Exceptions and revisions

Exceptions to this policy can be requested by emailing ansiblepartners@redhat.com. Exceptions are granted at Red Hat's discretion. Red Hat may change the requirements for validating content at any time. Red Hat will give partners advance notification of policy changes.

Feedback

Email ansiblepartners@redhat.com with feedback or questions regarding this document.