Guidelines for Ansible Amazon AWS module development

The Ansible AWS collection (on Galaxy, source code repository) is maintained by the Ansible AWS Working Group. For further information see the AWS working group community page. If you are planning to contribute AWS modules to Ansible then getting in touch with the working group is a good way to start, especially because a similar module may already be under development.

Requirements

Python Compatibility

AWS content in Ansible 2.9 and 1.x collection releases supported Python 2.7 and newer.

Starting with the 2.0 releases of both collections, Python 2.7 support will be ended in accordance with AWS' end of Python 2.7 support. Contributions to both collections that target the 2.0 or later collection releases can be written to support Python 3.6+ syntax.

SDK Version Support

Starting with the 2.0 releases of both collections, it is generally the policy to support the versions of botocore and boto3 that were released 12 months prior to the most recent major collection release, following semantic versioning (for example, 2.0.0, 3.0.0).

Features and functionality that require newer versions of the SDK can be contributed provided they are noted in the module documentation:

```
DOCUMENTATION = '''
---
module: ec2_vol
options:
  throughput:
    description:
      - Volume throughput in MB/s.
      - This parameter is only valid for gp3 volumes.
      - Valid range is from 125 to 1000.
      - Requires at least botocore version 1.19.27.
    type: int
    version_added: 1.4.0
```

And handled using the `botocore_at_least` helper method:

```
if module.params.get('throughput'):
```

```
    if not module.botocore_at_least("1.19.27"):
        module.fail_json(msg="botocore >= 1.19.27 is required to set the throughput for a volume")
```

Maintaining existing modules

Fixing bugs

Bug fixes to code that relies on boto will still be accepted. When possible, the code should be ported to use boto3.

Adding new features

Try to keep backward compatibility with relatively recent versions of boto3. That means that if you want to implement some functionality that uses a new feature of boto3, it should only fail if that feature actually needs to be run, with a message stating the missing feature and minimum required version of boto3.

Use feature testing (for example, `hasattr('boto3.module', 'shiny_new_method')`) to check whether boto3 supports a feature rather than version checking. For example, from the `ec2` module:

```
if boto_supports_profile_name_arg(ec2):
    params['instance_profile_name'] = instance_profile_name
else:
    if instance_profile_name is not None:
        module.fail_json(msg="instance_profile_name parameter requires boto version 2.5.0 or higher")
```

Migrating to boto3

Prior to Ansible 2.0, modules were written in either boto3 or boto. We are still porting some modules to boto3. Modules that still require boto should be ported to use boto3 rather than using both libraries (boto and boto3). We would like to remove the boto dependency from all modules.

Porting code to AnsibleAWSModule

Some old AWS modules use the generic `AnsibleModule` as a base rather than the more efficient `AnsibleAWSModule`. To port an old module to `AnsibleAWSModule`, change:

```
from ansible.module_utils.basic import AnsibleModule
...
module = AnsibleModule(...)
```

to:

```
from ansible_collections.amazon.aws.plugins.module_utils.core import AnsibleAWSModule
...
module = AnsibleAWSModule(...)
```

Few other changes are required. AnsibleAWSModule does not inherit methods from AnsibleModule by default, but most useful methods are included. If you do find an issue, please raise a bug report.

When porting, keep in mind that AnsibleAWSModule also will add the default ec2 argument spec by default. In pre-port modules, you should see common arguments specified with:

```
def main():
    argument_spec = ec2_argument_spec()
    argument_spec.update(dict(
        state=dict(default='present', choices=['present', 'absent', 'enabled', 'disabled']),
        name=dict(default='default'),
        # ... and so on ...
    ))
    module = AnsibleModule(argument_spec=argument_spec, supports_check_mode=True,)
```

These can be replaced with:

```
def main():
    argument_spec = dict(
        state=dict(default='present', choices=['present', 'absent', 'enabled', 'disabled']),
        name=dict(default='default'),
        # ... and so on ...
    )
    module = AnsibleAWSModule(argument_spec=argument_spec, supports_check_mode=True,)
```

Creating new AWS modules

Use boto3 and AnsibleAWSModule

All new AWS modules must use boto3 and `AnsibleAWSModule`.

`AnsibleAWSModule` greatly simplifies exception handling and library management, reducing the amount of boilerplate code. If you cannot use `AnsibleAWSModule` as a base, you must document the reason and request an exception to this rule.

Naming your module

Base the name of the module on the part of AWS that you actually use. (A good rule of thumb is to take whatever module you use with boto as a starting point). Don't further abbreviate names - if something is a well known abbreviation of a major component of AWS (for example, VPC or ELB), that's fine, but don't create new ones independently.

Unless the name of your service is quite unique, please consider using `aws_` as a prefix. For example `aws_lambda`.

Importing botocore and boto3

The `ansible_collections.amazon.aws.plugins.module_utils.ec2` module and `ansible_collections.amazon.aws.plugins.module_utils.core` modules both automatically import boto3 and botocore. If boto3 is missing from the system then the variable `HAS_BOTO3` will be set to false. Normally, this means that modules don't need to import boto3 directly. There is no need to check `HAS_BOTO3` when using AnsibleAWSModule as the module does that check:

```
from ansible_collections.amazon.aws.plugins.module_utils.core import AnsibleAWSModule
try:
    import botocore
except ImportError:
    pass  # handled by AnsibleAWSModule
```

or:

```
from ansible.module_utils.basic import AnsibleModule
from ansible_collections.amazon.aws.plugins.module_utils.ec2 import HAS_BOTO3
try:
    import botocore
except ImportError:
    pass  # handled by imported HAS_BOTO3

def main():

    if not HAS_BOTO3:
        module.fail_json(msg='boto3 and botocore are required for this module')
```

Supporting Module Defaults

The existing AWS modules support using :ref:`module_defaults <module_defaults>` for common authentication parameters. To do the same for your new module, add an entry for it in `meta/runtime.yml`. These entries take the form of:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\platforms\(ansible-devel)(docs)(docsite)(rst)(dev_guide)(platforms)aws_guidelines.rst, line 196); *backlink***
>
> Unknown interpreted text role "ref".

```
aws_module_name:
- aws
```

Connecting to AWS

AnsibleAWSModule provides the `resource` and `client` helper methods for obtaining boto3 connections. These handle some of the more esoteric connection options, such as security tokens and boto profiles.

If using the basic AnsibleModule then you should use `get_aws_connection_info` and then `boto3_conn` to connect to AWS as these handle the same range of connection options.

These helpers also for missing profiles or a region not set when it needs to be, so you don't have to.

An example of connecting to ec2 is shown below. Note that unlike boto there is no `NoAuthHandlerFound` exception handling like in boto. Instead, an `AuthFailure` exception will be thrown when you use the connection. To ensure that authorization, parameter validation and permissions errors are all caught, you should catch `ClientError` and `BotoCoreError` exceptions with every boto3 connection call. See exception handling:

```
module.client('ec2')
```

or for the higher level ec2 resource:

```
module.resource('ec2')
```

An example of the older style connection used for modules based on AnsibleModule rather than AnsibleAWSModule:

```
region, ec2_url, aws_connect_params = get_aws_connection_info(module, boto3=True)
connection = boto3_conn(module, conn_type='client', resource='ec2', region=region, endpoint=ec2_url, **aws_conne
```

```
region, ec2_url, aws_connect_params = get_aws_connection_info(module, boto3=True)
connection = boto3_conn(module, conn_type='client', resource='ec2', region=region, endpoint=ec2_url, **aws_conne
```

Common Documentation Fragments for Connection Parameters

There are two :ref:`common documentation fragments <module_docs_fragments>` that should be included into almost all AWS modules:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\platforms\(ansible-devel)(docs)(docsite)(rst)(dev_guide)(platforms)aws_guidelines.rst, line 249); *backlink***
>
> Unknown interpreted text role "ref".

- `aws` - contains the common boto connection parameters
- `ec2` - contains the common region parameter required for many AWS modules

These fragments should be used rather than re-documenting these properties to ensure consistency and that the more esoteric connection options are documented. For example:

```
DOCUMENTATION = '''
module: my_module
```

```
# some lines omitted here
requirements: [ 'botocore', 'boto3' ]
extends_documentation_fragment:
    - amazon.aws.aws
    - amazon.aws.ec2
'''
```

Handling exceptions

You should wrap any boto3 or botocore call in a try block. If an exception is thrown, then there are a number of possibilities for handling it.

- Catch the general `ClientError` or look for a specific error code with `is_boto3_error_code`.
- Use `aws_module.fail_json_aws()` to report the module failure in a standard way
- Retry using AWSRetry
- Use `fail_json()` to report the failure without using `ansible_collections.amazon.aws.plugins.module_utils.core`
- Do something custom in the case where you know how to handle the exception

For more information on botocore exception handling see the botocore error documentation.

Using is_boto3_error_code

To use `ansible_collections.amazon.aws.plugins.module_utils.core.is_boto3_error_code` to catch a single AWS error code, call it in place of `ClientError` in your except clauses. In this case, *only* the `InvalidGroup.NotFound` error code will be caught here, and any other error will be raised for handling elsewhere in the program.

```
try:
    info = connection.describe_security_groups(**kwargs)
except is_boto3_error_code('InvalidGroup.NotFound'):
    pass
do_something(info)  # do something with the info that was successfully returned
```

Using fail_json_aws()

In the AnsibleAWSModule there is a special method, `module.fail_json_aws()` for nice reporting of exceptions. Call this on your exception and it will report the error together with a traceback for use in Ansible verbose mode.

You should use the AnsibleAWSModule for all new modules, unless not possible. If adding significant amounts of exception handling to existing modules, we recommend migrating the module to use AnsibleAWSModule (there are very few changes required to do this)

```
from ansible_collections.amazon.aws.plugins.module_utils.core import AnsibleAWSModule

# Set up module parameters
# module params code here

# Connect to AWS
# connection code here

# Make a call to AWS
name = module.params.get['name']
try:
    result = connection.describe_frooble(FroobleName=name)
except (botocore.exceptions.BotoCoreError, botocore.exceptions.ClientError) as e:
    module.fail_json_aws(e, msg="Couldn't obtain frooble %s" % name)
```

Note that it should normally be acceptable to catch all normal exceptions here, however if you expect anything other than botocore exceptions you should test everything works as expected.

If you need to perform an action based on the error boto3 returned, use the error code and the `is_boto3_error_code()` helper.

```
# Make a call to AWS
name = module.params.get['name']
try:
    result = connection.describe_frooble(FroobleName=name)
except is_boto3_error_code('FroobleNotFound'):
    workaround_failure()  # This is an error that we can work around
except (botocore.exceptions.BotoCoreError, botocore.exceptions.ClientError) as e:  # pylint: disable=duplicate-e
    module.fail_json_aws(e, msg="Couldn't obtain frooble %s" % name)
```

using fail_json() and avoiding ansible_collections.amazon.aws.plugins.module_utils.core

Boto3 provides lots of useful information when an exception is thrown so pass this to the user along with the message.

```
from ansible.module_utils.ec2 import HAS_BOTO3
try:
    import botocore
except ImportError:
    pass  # caught by imported HAS_BOTO3

# Connect to AWS
# connection code here

# Make a call to AWS
name = module.params.get['name']
try:
    result = connection.describe_frooble(FroobleName=name)
except botocore.exceptions.ClientError as e:
```

```
    module.fail_json(msg="Couldn't obtain frooble %s: %s" % (name, str(e)),
                     exception=traceback.format_exc(),
                     **camel_dict_to_snake_dict(e.response))
```

Note: we use *str(e)* rather than *e.message* as the latter doesn't work with python3

If you need to perform an action based on the error boto3 returned, use the error code.

```
# Make a call to AWS
name = module.params.get['name']
try:
    result = connection.describe_frooble(FroobleName=name)
except botocore.exceptions.ClientError as e:
    if e.response['Error']['Code'] == 'FroobleNotFound':
        workaround_failure()  # This is an error that we can work around
    else:
        module.fail_json(msg="Couldn't obtain frooble %s: %s" % (name, str(e)),
                         exception=traceback.format_exc(),
                         **camel_dict_to_snake_dict(e.response))
except botocore.exceptions.BotoCoreError as e:
    module.fail_json_aws(e, msg="Couldn't obtain frooble %s" % name)
```

API throttling (rate limiting) and pagination

For methods that return a lot of results, boto3 often provides paginators. If the method you're calling has `NextToken` or `Marker` parameters, you should probably check whether a paginator exists (the top of each boto3 service reference page has a link to Paginators, if the service has any). To use paginators, obtain a paginator object, call `paginator.paginate` with the appropriate arguments and then call `build_full_result`.

Any time that you are calling the AWS API a lot, you may experience API throttling, and there is an `AWSRetry` decorator that can be used to ensure backoff. Because exception handling could interfere with the retry working properly (as AWSRetry needs to catch throttling exceptions to work correctly), you'd need to provide a backoff function and then put exception handling around the backoff function.

You can use `exponential_backoff` or `jittered_backoff` strategies - see the cloud `module_utils` ()/lib/ansible/module_utils/cloud.py) and AWS Architecture blog for more details.

The combination of these two approaches is then:

```
@AWSRetry.exponential_backoff(retries=5, delay=5)
def describe_some_resource_with_backoff(client, **kwargs):
     paginator = client.get_paginator('describe_some_resource')
     return paginator.paginate(**kwargs).build_full_result()['SomeResource']

def describe_some_resource(client, module):
    filters = ansible_dict_to_boto3_filter_list(module.params['filters'])
    try:
        return describe_some_resource_with_backoff(client, Filters=filters)
    except botocore.exceptions.ClientError as e:
        module.fail_json_aws(e, msg="Could not describe some resource")
```

Prior to Ansible 2.10 if the underlying `describe_some_resources` API call threw a `ResourceNotFound` exception, `AWSRetry` would take this as a cue to retry until it is not thrown (this is so that when creating a resource, we can just retry until it exists). This default was changed and it is now necessary to explicitly request this behaviour. This can be done by using the `catch_extra_error_codes` argument on the decorator.

```
@AWSRetry.exponential_backoff(retries=5, delay=5, catch_extra_error_codes=['ResourceNotFound'])
def describe_some_resource_retry_missing(client, **kwargs):
     return client.describe_some_resource(ResourceName=kwargs['name'])['Resources']

def describe_some_resource(client, module):
    name = module.params.get['name']
    try:
        return describe_some_resource_with_backoff(client, name=name)
    except (botocore.exceptions.BotoCoreError, botocore.exceptions.ClientError) as e:
        module.fail_json_aws(e, msg="Could not describe resource %s" % name)
```

To make use of AWSRetry easier, it can now be wrapped around a client returned by `AnsibleAWSModule`. any call from a client. To add retries to a client, create a client:

```
module.client('ec2', retry_decorator=AWSRetry.jittered_backoff(retries=10))
```

Any calls from that client can be made to use the decorator passed at call-time using the *aws_retry* argument. By default, no retries are used.

```
ec2 = module.client('ec2', retry_decorator=AWSRetry.jittered_backoff(retries=10))
ec2.describe_instances(InstanceIds=['i-123456789'], aws_retry=True)

# equivalent with normal AWSRetry
@AWSRetry.jittered_backoff(retries=10)
def describe_instances(client, **kwargs):
    return ec2.describe_instances(**kwargs)

describe_instances(module.client('ec2'), InstanceIds=['i-123456789'])
```

The call will be retried the specified number of times, so the calling functions don't need to be wrapped in the backoff decorator.

You can also use customization for `retries`, `delay` and `max_delay` parameters used by `AWSRetry.jittered_backoff` API using module params. You can take a look at the *cloudformation <cloudformation_module>* module for example.

To make all Amazon modules uniform, prefix the module param with `backoff_`, so `retries` becomes `backoff_retries` and likewise with `backoff_delay` and `backoff_max_delay`.

Returning Values

When you make a call using boto3, you will probably get back some useful information that you should return in the module. As well as information related to the call itself, you will also have some response metadata. It is OK to return this to the user as well as they may find it useful.

Boto3 returns all values CamelCased. Ansible follows Python standards for variable names and uses snake_case. There is a helper function in module_utils/ec2.py called *camel_dict_to_snake_dict* that allows you to easily convert the boto3 response to snake_case.

You should use this helper function and avoid changing the names of values returned by Boto3. E.g. if boto3 returns a value called 'SecretAccessKey' do not change it to 'AccessKey'.

```
# Make a call to AWS
result = connection.aws_call()

# Return the result to the user
module.exit_json(changed=True, **camel_dict_to_snake_dict(result))
```

Dealing with IAM JSON policy

If your module accepts IAM JSON policies then set the type to 'json' in the module spec. For example:

```
argument_spec.update(
    dict(
        policy=dict(required=False, default=None, type='json'),
    )
)
```

Note that AWS is unlikely to return the policy in the same order that is was submitted. Therefore, use the *compare_policies* helper function which handles this variance.

compare_policies takes two dictionaries, recursively sorts and makes them hashable for comparison and returns True if they are different.

```
from ansible_collections.amazon.aws.plugins.module_utils.ec2 import compare_policies

import json

# some lines skipped here

# Get the policy from AWS
current_policy = json.loads(aws_object.get_policy())
user_policy = json.loads(module.params.get('policy'))

# Compare the user submitted policy to the current policy ignoring order
if compare_policies(user_policy, current_policy):
    # Update the policy
    aws_object.set_policy(user_policy)
else:
    # Nothing to do
    pass
```

Dealing with tags

AWS has a concept of resource tags. Usually the boto3 API has separate calls for tagging and untagging a resource. For example, the ec2 API has a create_tags and delete_tags call.

It is common practice in Ansible AWS modules to have a *purge_tags* parameter that defaults to true.

The *purge_tags* parameter means that existing tags will be deleted if they are not specified by the Ansible task.

There is a helper function *compare_aws_tags* to ease dealing with tags. It can compare two dicts and return the tags to set and the tags to delete. See the Helper function section below for more detail.

Helper functions

Along with the connection functions in Ansible ec2.py module_utils, there are some other useful functions detailed below.

camel_dict_to_snake_dict

boto3 returns results in a dict. The keys of the dict are in CamelCase format. In keeping with Ansible format, this function will convert the keys to snake_case.

`camel_dict_to_snake_dict` takes an optional parameter called `ignore_list` which is a list of keys not to convert (this is usually useful for the `tags` dict, whose child keys should remain with case preserved)

Another optional parameter is `reversible`. By default, `HTTPEndpoint` is converted to `http_endpoint`, which would then be converted by `snake_dict_to_camel_dict` to `HttpEndpoint`. Passing `reversible=True` converts HTTPEndpoint to `h_t_t_p_endpoint` which converts back to `HTTPEndpoint`.

snake_dict_to_camel_dict

snake_dict_to_camel_dict converts snake cased keys to camel case. By default, because it was first introduced for ECS purposes, this converts to dromedaryCase. An optional parameter called *capitalize_first*, which defaults to *False*, can be used to convert to CamelCase.

ansible_dict_to_boto3_filter_list

Converts a an Ansible list of filters to a boto3 friendly list of dicts. This is useful for any boto3 *_facts* modules.

boto_exception

Pass an exception returned from boto or boto3, and this function will consistently get the message from the exception.

Deprecated: use *AnsibleAWSModule*'s *fail_json_aws* instead.

boto3_tag_list_to_ansible_dict

Converts a boto3 tag list to an Ansible dict. Boto3 returns tags as a list of dicts containing keys called 'Key' and 'Value' by default. This key names can be overridden when calling the function. For example, if you have already camel_cased your list of tags you may want to pass lowercase key names instead, in other words, 'key' and 'value'.

This function converts the list in to a single dict where the dict key is the tag key and the dict value is the tag value.

ansible_dict_to_boto3_tag_list

Opposite of above. Converts an Ansible dict to a boto3 tag list of dicts. You can again override the key names used if 'Key' and 'Value' is not suitable.

get_ec2_security_group_ids_from_names

Pass this function a list of security group names or combination of security group names and IDs and this function will return a list of IDs. You should also pass the VPC ID if known because security group names are not necessarily unique across VPCs.

compare_policies

Pass two dicts of policies to check if there are any meaningful differences and returns true if there are. This recursively sorts the dicts and makes them hashable before comparison.

This method should be used any time policies are being compared so that a change in order doesn't result in unnecessary changes.

compare_aws_tags

Pass two dicts of tags and an optional purge parameter and this function will return a dict containing key pairs you need to modify and a list of tag key names that you need to remove. Purge is True by default. If purge is False then any existing tags will not be modified.

This function is useful when using boto3 'add_tags' and 'remove_tags' functions. Be sure to use the other helper function *boto3_tag_list_to_ansible_dict* to get an appropriate tag dict before calling this function. Since the AWS APIs are not uniform (for example, EC2 is different from Lambda) this will work without modification for some (Lambda) and others may need modification before using these values (such as EC2, with requires the tags to unset to be in the form *[{'Key': key1}, {'Key': key2}]*).

Integration Tests for AWS Modules

All new AWS modules should include integration tests to ensure that any changes in AWS APIs that affect the module are detected. At a minimum this should cover the key API calls and check the documented return values are present in the module result.

For general information on running the integration tests see the :ref:`Integration Tests page of the Module Development Guide <testing_integration>`, especially the section on configuration for cloud tests.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\platforms\(ansible-devel)(docs)(docsite)(rst)(dev_guide)(platforms)aws_guidelines.rst`, **line 656**); *backlink*
>
> Unknown interpreted text role "ref".

The integration tests for your module should be added in *test/integration/targets/MODULE_NAME*.

You must also have a aliases file in *test/integration/targets/MODULE_NAME/aliases*. This file serves two purposes. First indicates it's in an AWS test causing the test framework to make AWS credentials available during the test run. Second putting the test in a test group causing it to be run in the continuous integration build.

Tests for new modules should be added to the same group as existing AWS tests. In general just copy an existing aliases file such as the aws_s3 tests aliases file.

AWS Credentials for Integration Tests

The testing framework handles running the test with appropriate AWS credentials, these are made available to your test in the following variables:

- *aws_region*
- *aws_access_key*
- *aws_secret_key*
- *security_token*

So all invocations of AWS modules in the test should set these parameters. To avoid duplicating these for every call, it's preferable to use :ref:`module_defaults <module_defaults>`. For example:

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\platforms\(ansible-devel)(docs)(docsite)(rst)(dev_guide)(platforms)aws_guidelines.rst`, **line 680**); *backlink*
>
> Unknown interpreted text role "ref".

```
- name: set connection information for aws modules and run tasks
  module_defaults:
    group/aws:
      aws_access_key: "{{ aws_access_key }}"
      aws_secret_key: "{{ aws_secret_key }}"
      security_token: "{{ security_token | default(omit) }}"
      region: "{{ aws_region }}"

  block:

  - name: Do Something
    ec2_instance:
      ... params ...

  - name: Do Something Else
    ec2_instance:
      ... params ...
```

AWS Permissions for Integration Tests

As explained in the :ref:`Integration Test guide <testing_integration>` there are defined IAM policies in mattclay/aws-terminator that contain the necessary permissions to run the AWS integration test.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\platforms\(ansible-devel)(docs)(docsite)(rst)(dev_guide)(platforms)aws_guidelines.rst`, **line 706**); ***backlink***
>
> Unknown interpreted text role "ref".

If your module interacts with a new service or otherwise requires new permissions, tests will fail when you submit a pull request and the Ansibullbot will tag your PR as needing revision. We do not automatically grant additional permissions to the roles used by the continuous integration builds. You will need to raise a Pull Request against mattclay/aws-terminator to add them.

If your PR has test failures, check carefully to be certain the failure is only due to the missing permissions. If you've ruled out other sources of failure, add a comment with the *ready_for_review* tag and explain that it's due to missing permissions.

Your pull request cannot be merged until the tests are passing. If your pull request is failing due to missing permissions, you must collect the minimum IAM permissions required to run the tests.

There are two ways to figure out which IAM permissions you need for your PR to pass:

- Start with the most permissive IAM policy, run the tests to collect information about which resources your tests actually use, then construct a policy based on that output. This approach only works on modules that use *AnsibleAWSModule*.
- Start with the least permissive IAM policy, run the tests to discover a failure, add permissions for the resource that addresses that failure, then repeat. If your module uses *AnsibleModule* instead of *AnsibleAWSModule*, you must use this approach.

To start with the most permissive IAM policy:

1. Create an IAM policy that allows all actions (set `Action` and `Resource` to `*`).
2. Run your tests locally with this policy. On AnsibleAWSModule-based modules, the `debug_botocore_endpoint_logs` option is automatically set to `yes`, so you should see a list of AWS ACTIONS after the PLAY RECAP showing all the permissions used. If your tests use a boto/AnsibleModule module, you must start with the least permissive policy (see below).
3. Modify your policy to allow only the actions your tests use. Restrict account, region, and prefix where possible. Wait a few minutes for your policy to update.
4. Run the tests again with a user or role that allows only the new policy.
5. If the tests fail, troubleshoot (see tips below), modify the policy, run the tests again, and repeat the process until the tests pass with a restrictive policy.
6. Open a pull request proposing the minimum required policy to the CI policies.

To start from the least permissive IAM policy:

1. Run the integration tests locally with no IAM permissions.
2. Examine the error when the tests reach a failure.
    a. If the error message indicates the action used in the request, add the action to your policy.
    b. If the error message does not indicate the action used in the request:
        - Usually the action is a CamelCase version of the method name - for example, for an ec2 client the method *describe_security_groups* correlates to the action *ec2:DescribeSecurityGroups*.
        - Refer to the documentation to identify the action.
    c. If the error message indicates the resource ARN used in the request, limit the action to that resource.
    d. If the error message does not indicate the resource ARN used:
        - Determine if the action can be restricted to a resource by examining the documentation.
        - If the action can be restricted, use the documentation to construct the ARN and add it to the policy.
3. Add the action or resource that caused the failure to an IAM policy. Wait a few minutes for your policy to update.
4. Run the tests again with this policy attached to your user or role.
5. If the tests still fail at the same place with the same error you will need to troubleshoot (see tips below). If the first test passes, repeat steps 2 and 3 for the next error. Repeat the process until the tests pass with a restrictive policy.
6. Open a pull request proposing the minimum required policy to the CI policies.

Troubleshooting IAM policies

- When you make changes to a policy, wait a few minutes for the policy to update before re-running the tests.

- Use the policy simulator to verify that each action (limited by resource when applicable) in your policy is allowed.
- If you're restricting actions to certain resources, replace resources temporarily with *. If the tests pass with wildcard resources, there is a problem with the resource definition in your policy.
- If the initial troubleshooting above doesn't provide any more insight, AWS may be using additional undisclosed resources and actions.
- Examine the AWS FullAccess policy for the service for clues.
- Re-read the AWS documentation, especially the list of Actions, Resources and Condition Keys for the various AWS services.
- Look at the cloudonaut documentation as a troubleshooting cross-reference.
- Use a search engine.
- Ask in the #ansible-aws chat channel (using Matrix at ansible.im or using IRC at irc.libera.chat).

Unsupported Integration tests

There are a limited number of reasons why it may not be practical to run integration tests for a module within CI. Where these apply you should add the keyword *unsupported* to the aliases file in *test/integration/targets/MODULE_NAME/aliases*.

Some cases where tests should be marked as unsupported: 1) The tests take longer than 10 or 15 minutes to complete 2) The tests create expensive resources 3) The tests create inline policies 4) The tests require the existence of external resources 5) The tests manage Account level security policies such as the password policy or AWS Organizations.

Where one of these reasons apply you should open a pull request proposing the minimum required policy to the unsupported test policies.

Unsupported integration tests will not be automatically run by CI. However, the necessary policies should be available so that the tests can be manually run by someone performing a PR review or writing a patch.