Network Testing: Validating Network State with Automation

Dan Wade

Author, Public Speaker,
Network Automation Nerd

Pearson

Course Agenda

Section 1: Software Testing Concepts / pyATS Overview

Section 2: pyATS - AEtest Infrastructure

Section 3: pyATS Library (Genie)

Section 4: Integrating Testing into a Network Automation Workflow

Poll Question

How would you describe your current job role or experience with network automation? (single response)

- Network engineer
- Network engineer with a little automation experience
- Full-time network automation engineer
- Software Engineer
- Other

Section 1: Software Testing Basics / pyATS Overview

1.1 Software Testing Concepts

1.2 Applying Software Testing to Networking

1.3 What is pyATS and the pyATS Library?

1.4 The pyATS Architecture

Software Testing Concepts

Software Testing History

- 1960’s - Recognized as a discipline
  - Debugging and test case design
  - Introduced the practices of unit and integration testing
- 1980’s - Black-box and white-box testing
- 1990’s - Dedicated roles emerged (QA engineers and analysts)
- 2000s - DevOps era begins
  - Agile
  - DevOps
  - CI/CD Pipelines
  - Test-Driven Development (TDD)

Software Testing Categories

Category	Examples
Functional Testing	Unit, Integration, System, Smoke, Sanity, Regression, UAT
Non-Functional Testing	Performance, Load, Stress, Security, Usability, Compatibility
Maintenance Testing	Regression, Retesting
Change-Related Testing	Smoke, Sanity, Regression

Functional Testing

- Unit - Tests individual units (i.e. functions)
- Integration - Verifies functionality between modules
- System - Tests the entire system
- Sanity - Quick checks after small changes
- Smoke - Basic checks to ensure stability
- Regression - Ensures new changes didn’t break existing features
- User Acceptance Testing (UAT) - Performed by end users

Non-Functional Testing

- Performance - Responsiveness and stability under load
- Load - Performance under expected usage
- Stress - Pushes system to identify current limits
- Security - Identify system vulnerabilities
- Usability - User Experience (UX) testing
- Compatibility - Ensures software works across different devices/platforms/browsers

Applying Software Testing to Networking

Applying Software Testing to Networking

- The network is a distributed system of nodes
- Network engineers naturally perform Smoke and Sanity tests
  - “Let me ping the core router and internal DNS servers to make sure I didn’t break anything.”
- Applicable Testing Categories/Types:
  - Functional Testing
    - Unit testing
    - Integration testing
    - End-to-End testing
    - Regression testing

Unit Testing for Networking

In programming, unit testing tests a single “unit” of function of a program.

Unit testing in networking could be validating a feature or function that’s particular to a network device.

Examples: Environment (Power/CPU/Mem), Layer 2 (STP/MST, Port Security, VTP), Security (ACLs, Protocol Authentication, AAA)

Integration Testing for Networking

Integration testing ensures separate modules interface with one another properly.

Networking devices communicating and transferring information with one another.

Routing is a very popular integration test domain.

Examples: Routing policies, routing table entries, neighbor/adjacency state, routing metrics, routing stats (received routes, sent routes, etc.)

End-to-End Testing for Networking

End-to-end testing is exactly what you think it is… ensuring the whole system works from start to finish.

Popular end-to-end network testing tools include ping, traceroute, and iperf (performance testing)

Regression Testing for Networking

Regression tests run after every change (big or small) to ensure the change doesn't introduce unintended bugs.

Network engineers perform regression testing during every maintenance window.

Examples:

- After applying a routing change, check the routing table for any unintended routing issues.
- After applying an ACL to an interface, check for hits against each ACL entry.

What is pyATS and the pyATS Library?

What is pyATS?

pyATS = Python Automated Testing Systems

Testing framework developed by Cisco, but is vendor-agnostic. (100% developed in Python)

Data-driven and reusable tests, focused on fast and iterative development

Shares characteristics of other Python test frameworks: unittest and pytest

pyATS Concepts

Testbed

TestScripts

Jobs

pyATS Concepts - Testbed

Defines network topology and devices under testing

Testbeds describe the physical devices and connections in a YAML file

Unicon library - Provides unified interface to control device connectivity

Supported Platforms:
https://pubhub.devnetcloud.com/media/unicon/docs/user_guide/supported_platforms.html#supported-platforms

pyATS Concepts - TestScripts

A Python file that contains the logic and tests to execute

Can be executed standalone and test results are printed to standard output (stdout)

pyATS Concepts - Jobs

Executes TestScripts as *tasks* in a standardized runtime environment

Allows for multiple TestScripts to be executed at once, or in parallel

All TestScript test results and logs executed within a Job are aggregated into a standard format for better reporting

pyATS Library (Genie)

What is the pyATS Library (Genie)?

Built on top of pyATS. Provides the “tooling” for network engineers to extract network data and create reusable tests

Parses configuration and operational state data from network devices

Provides a testing harness built on pyATS called Genie Harness

pyATS Library (Genie) - Harness

Provides ability to build and run test cases with ease using YAML datafiles. Built on top of pyATS test framework.

Stage	Name	Description
1	Common setup	Prepare devices for testing: • Connect • Configure (optional) • Generate traffic (optional) • Take a snapshot of configuration, operational state, or both (optional)
2	Triggers and verifications	Perform tests and run show commands on your devices.
3	Common cleanup	Check that device states match the original: • Take a snapshot of configuration, operational state, or both (optional) • Compare snapshots with the common setup snapshots (optional) • Stop generated traffic (optional)

pyATS Library (Genie) - Triggers/Verifications

Triggers - Actions that change a device's state or configuration (like a pyATS testcase)

Verifications - Retrieves current state of a device using "show" commands. Typically runs before and after a trigger to compare state.

```
genie run --testbed-file mock.yaml --
trigger-uids="TriggerShutNoShutBgp" --
verification-uids="Verify_BgpProcessVrfAll"
--devices uut
```

Example: https://github.com/CiscoTestAutomation/examples/tree/master/libraries/harness_triggers

pyATS Library (Genie) - Harness Workflow

https://pubhub.devnetcloud.com/media/genie-docs/docs/cookbooks/harness.html#common-

pyATS Library (Genie) - Datafiles

Allows test execution to be *data-driven* by passing in data at runtime vs having hard-coded values in tests

- Mapping datafile
- Configuration datafile
- Trigger datafile
- Verification datafile

pyATS Library (Genie) - Mapping Datafile

mapping.yaml

```
devices:
    PE1:
     context: cli
     label: uut
     mapping:
        cli: vty
        yang: netconf
```

pyATS Library (Genie) - Configuration Datafile

configs.yaml

```
devices:
    uut:
        1:
          config: /path/to/my/configuration
          sleep: 3
          invalid: ['overlaps', '(.*inval.*)']
        2:
          jinja2_config: routing.j2
          jinja2_arguments:
            lstrip_blocks: true
            trim_blocks: true
            bgp_data:
                bgp_as: 100
                neighbor_ips: [
                    '1.1.1.1', '2.2.2.2'
                ]
```

pyATS Library (Genie) - Other Harness Features

PTS - Profiles device features during the Common Setup and Common Cleanup sections to ensure the operational state hasn't change during testing.

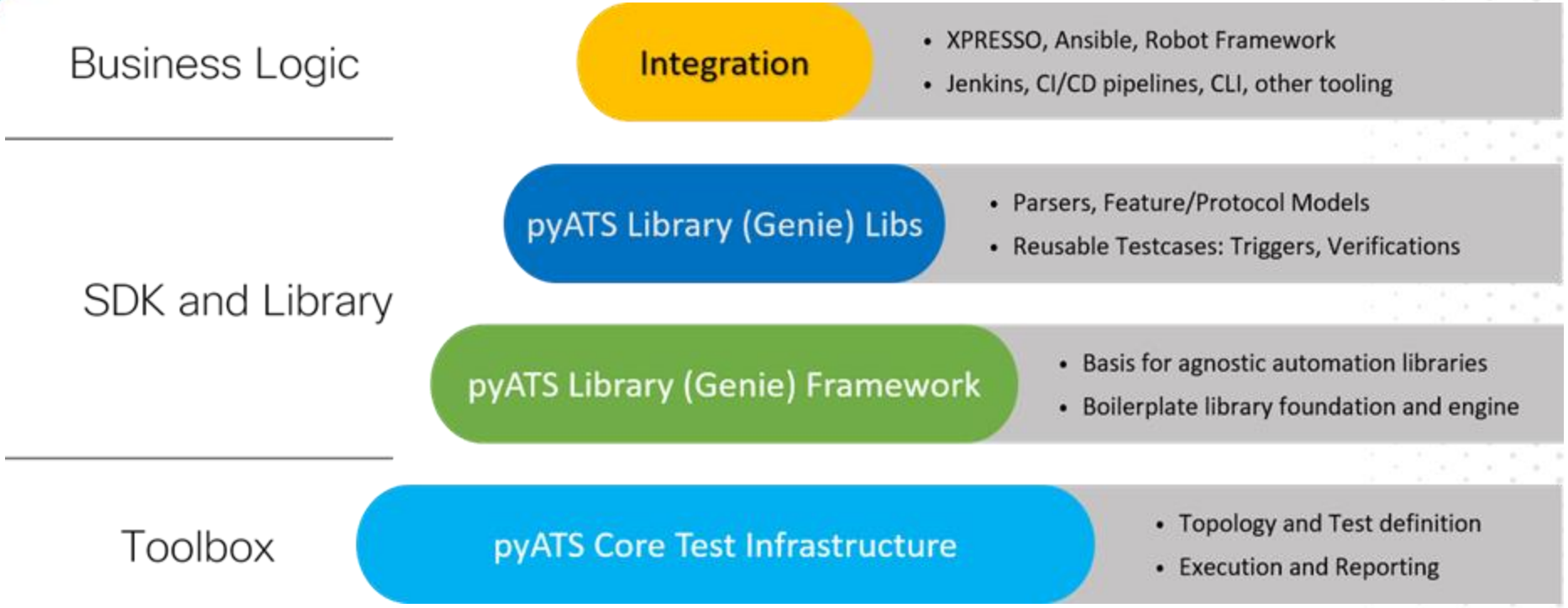
Golden Config - Compares the profiles learned via PTS against a "golden" profile.

File Transfer Protocol - Used to transfer a configuration file during Common Setup or to copy core/crash dump files during testing

Connection Pool - Configuration commands can be sent to the same device in parallel instead of sequential.

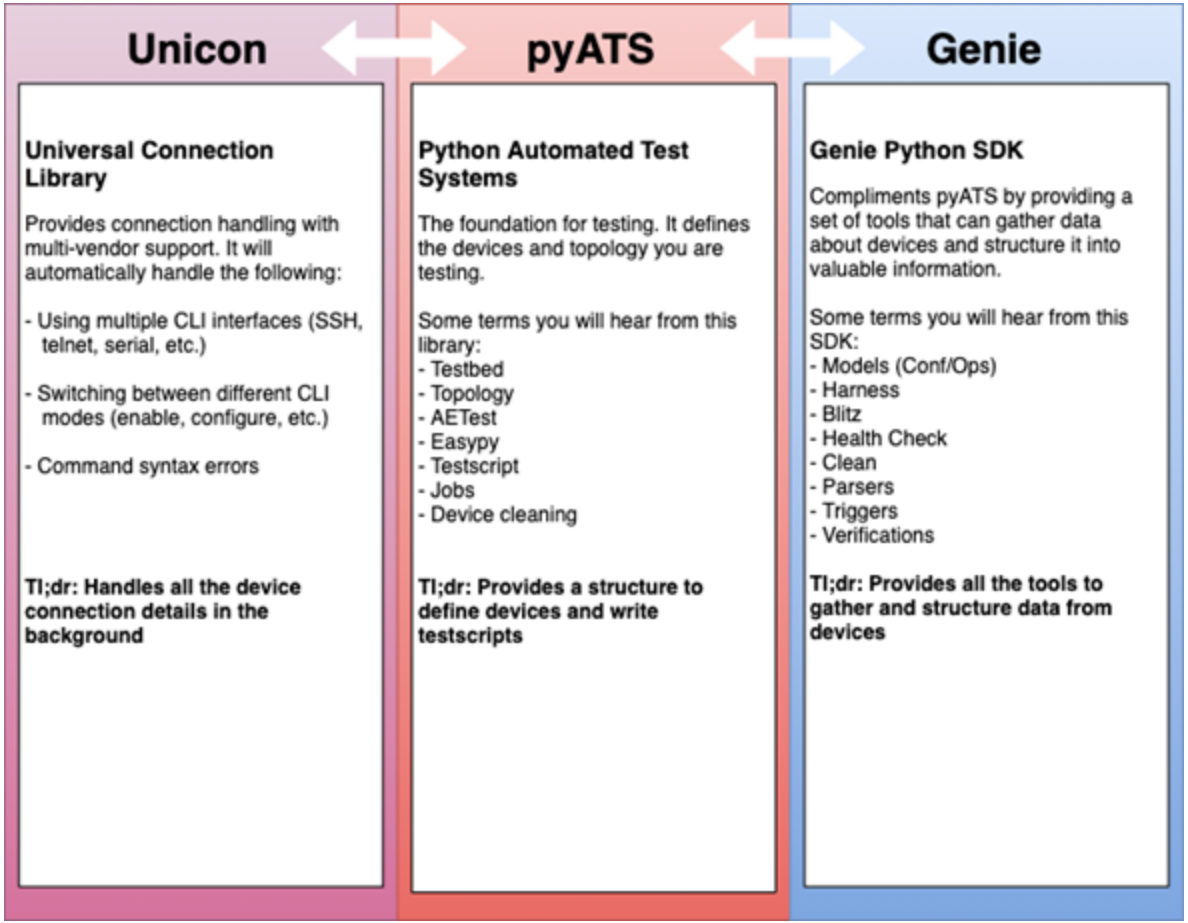
pyATS Architecture

pyATS Architecture



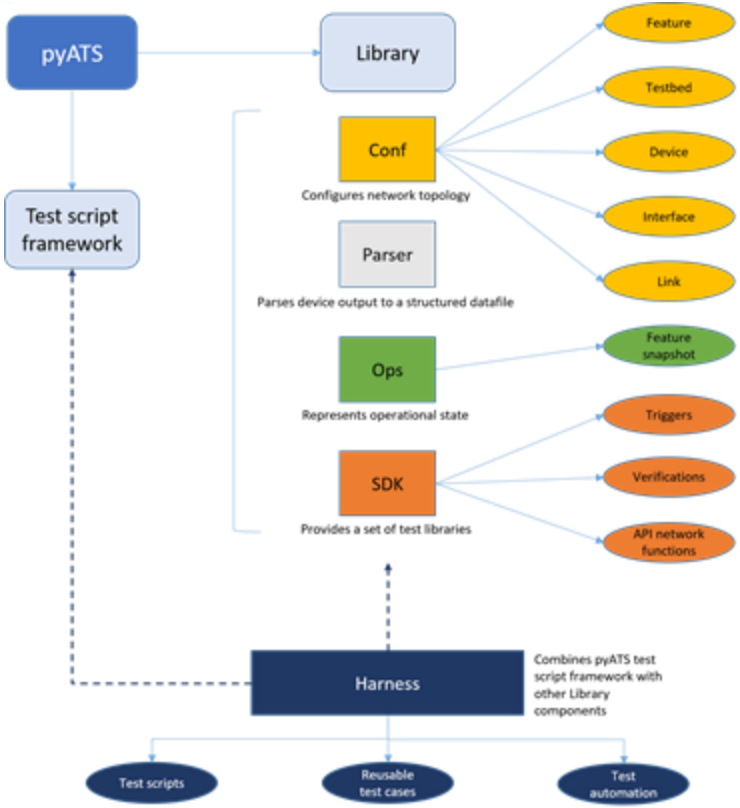
https://developer.cisco.com/docs/pyats-getting-started/

pyATS Architecture - Libraries



https://devnetdan.com/2021/06/14/pyats-and-genie-part-2/

pyATS Architecture - Test Automation Ecosystem



https://pubhub.devnetcloud.com/media/pyats-getting-started/docs/intro/architecture.html#test-ecosystem-architecture

Pearson

Q+A

5-min Break

Section 2: pyATS - AEtest Infrastructure

2.1 TestScript Structure

2.2 Testcase Sections

2.3 Test Parameters

2.4 Test Execution

2.5 Results and Reporting

pyATS - AEtest Infrastructure Overview

AEtest - Automation Easy Testing

Provides framework to build, execute, and debug testscripts and testcases

Serves as base for other test engines (Genie Harness)

Takes advantage of Python's OOP concept

TestScript Structure

AEtest - TestScript Structure

Common Setup

Testcases

Common Cleanup

AEtest - TestScript Structure - Common Setup

Common Setup - Initial configuration and device initialization activities.

- Connects to testbed devices
- Apply initial configuration to devices
- Setup dynamic looping of testcases based on the current environment

AEtest - TestScript Structure - Testcases

Testcases - Contains all the individual tests

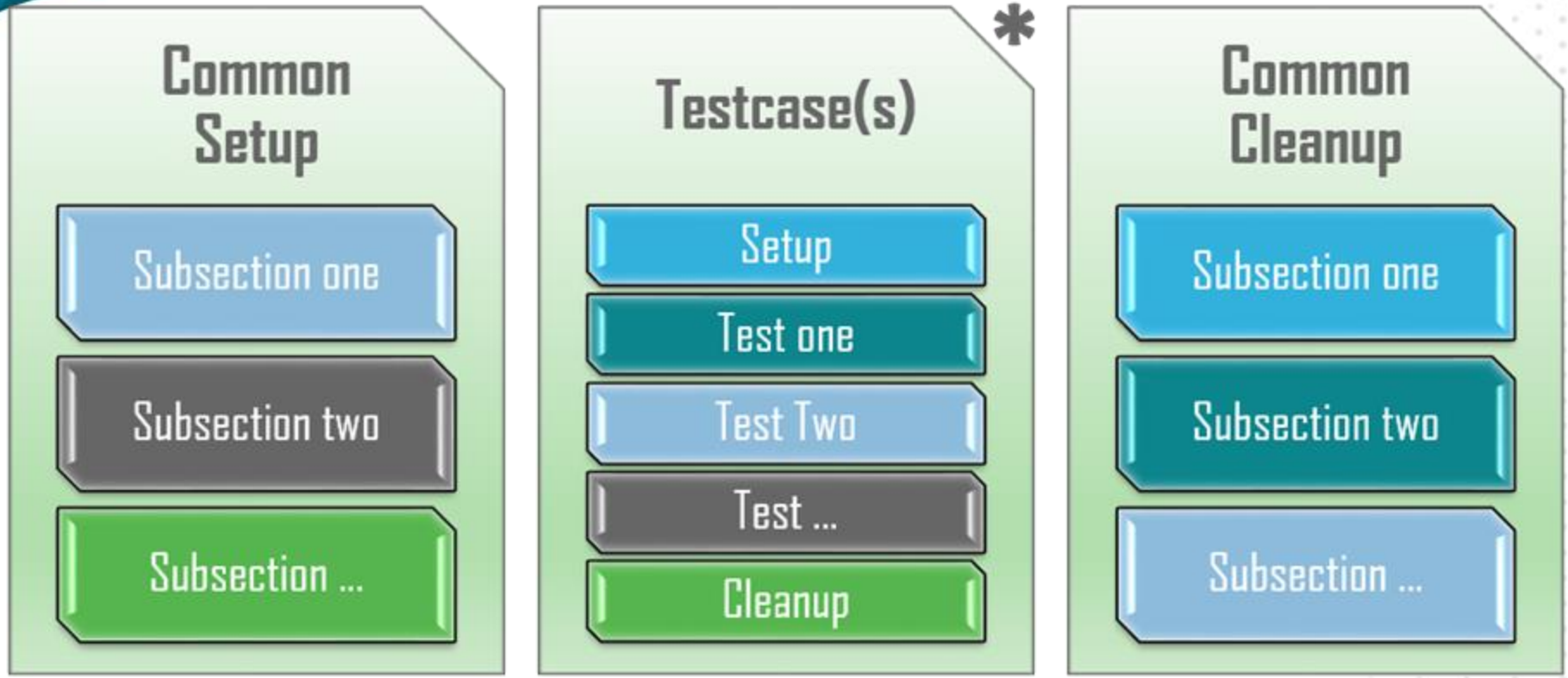
- Testcases can have the following subsections:
  - Setup section
  - Test section
  - Cleanup section
- Testcases must have unique IDs (`testcase.uid`). Defaults to the testcase's name.
- Testcases are independent and should be self-contained

AEtest - TestScript Structure - Common Cleanup

Common Setup - Resets the testing environment. Removes any configuration applied during testing.

- Connects to testbed devices
- Apply initial configuration to devices
- Setup dynamic looping of testcases based on the current environment

AEtest - TestScript Structure - Visual Diagram



Pearson

https://pubhub.devnetcloud.com/media/pyats/docs/aetest/structure.html#script-

Test Sections

AEtest - Test Sections

Setup section - Initial configuration specific to the testcase. Uses `@aetest.setup` decorator.

Test section - Runs specific evaluation/check. Uses `@aetest.test` decorator.

Cleanup section - Removes any applied configuration during testing in the testcase. Uses `@aetest.cleanup` decorator.

AEtest - Test Sections - Example

```
from pyats import aetest

# setup/test/cleanup sections within Testcases
class Testcase(aetest.Testcase):

    @aetest.setup
    def testcase_setup(self):
        pass
    @aetest.test
    def test_one(self):
        pass
    @aetest.cleanup
    def testcase_cleanup(self):
        pass
```

AEtest - Subsections

Subsections - Helps break up Common Setup and Common Cleanup into identifiable units. Uses `@aetest.subsection` decorator.

```
from pyats import aetest

class ScriptCommonSetup(aetest.CommonSetup):

    @aetest.subsection
    def common_setup_subsection(self):
        pass

# … Some testing is performed …

class ScriptCommonCleanup(aetest.CommonCleanup):

    @aetest.subsection
    def common_cleanup_subsection(self):
         pass
```

Test Parameters

AEtest - Test Parameters

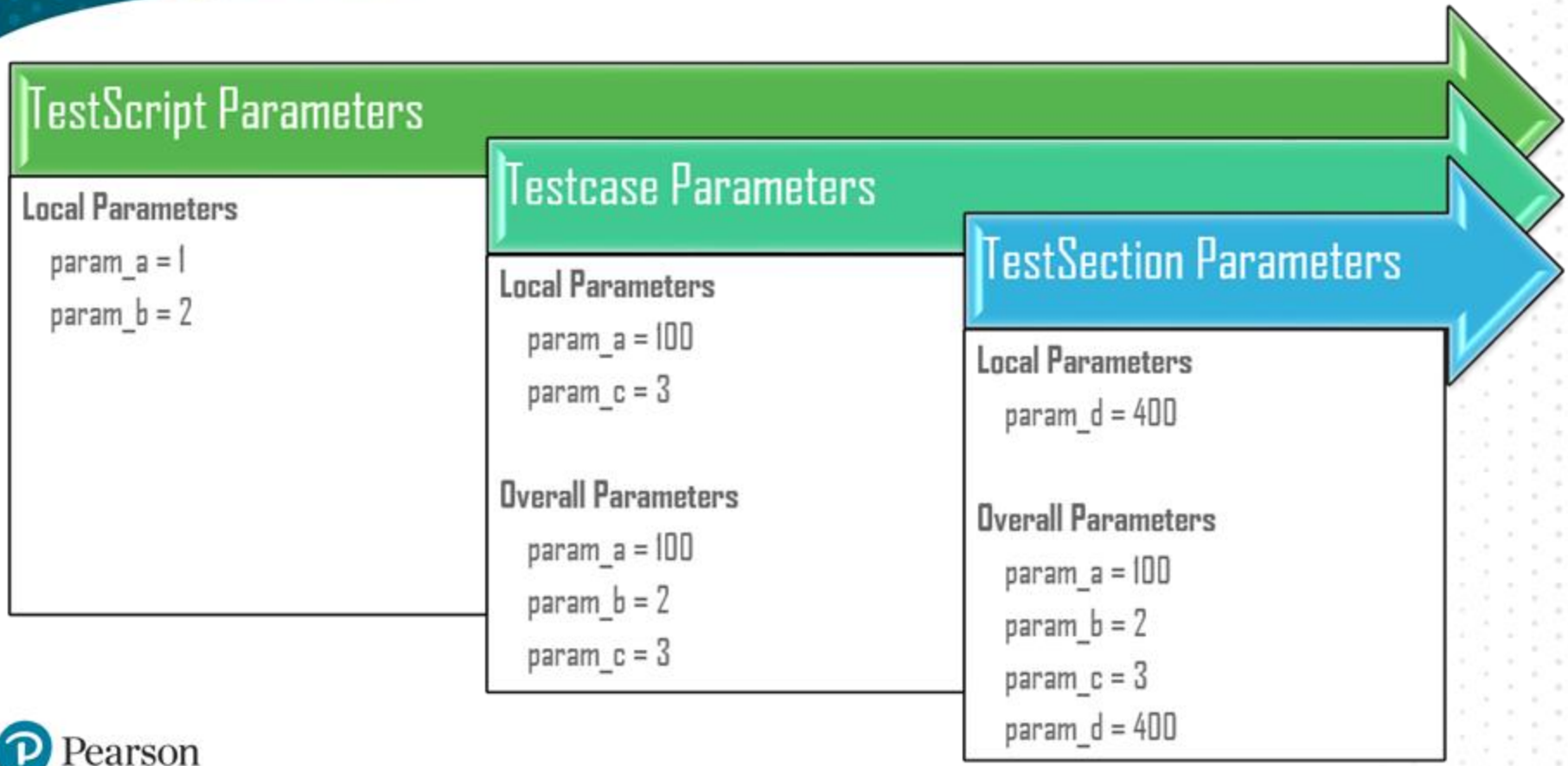
Dynamic data that changes the behavior of testscripts and testcases

TestScripts are designed to be data-driven

Test parameters should be dynamically provided via:

- Input arguments to the TestScript
- Dynamically generated during runtime

AEtest - Test Parameters - Relationship Model



Pearson

https://pubhub.devnetcloud.com/media/pyats/docs/aetest/parameters.html#relationship-model

AEtest - Test Parameters - Predefined Parameters

`parameter` property - Python dictionary that creates baseline parameters *at runtime*

- TestScript-level parameters dictionary
- Testcase-level parameters dictionary

Predefined Parameters Example

```
from pyats import aetest

# testscript level default parameters
parameters = {
    'testscript_param_A': 'some value',
    'testscript_param_B': [],
    'generic_param_A': 100
}

class Testcase(aetest.Testcase):
    # testcase level default parameters
    parameters = {
        'generic_param_A': 200
     }
```

Modifying Predefined Parameters Example

```
class Testcase(aetest.Testcase):
    # testcase parameters defaults, same as above
    parameters = {
        'generic_param_A': 200
    }
    # here we'll do a combination access & updating of parameters
    @aetest.setup
    def setup(self):
        # add to the parameters dict
        self.parameters['new_parameter_from_setup'] = 'new value'

    @aetest.test
    def test(self):
        # access & print all known parameters
        print(self.parameters)
        # {'new_parameter_from_setup': 'new value',
        #  'generic_param_A': 200,
        #  'testscript_param_B': [],
        #  'testscript_param_A': 'some value'}
```

AEtest - Test Parameters - Script Arguments

Arguments passed before testscript execution becomes part of the TestScript parameters

Testscript arguments can be passed via:

- Command-line arguments
- Through Jobfiles during Easypy execution
- Passed to `aetest.main()` in Standalone execution

AEtest - Test Parameters - Callable Arguments

A callable that evaluates to ‘True’

AEtest ‘calls’ the callable and uses the return value as the parameter

Must be passed as function arguments to be evaluated

Callable Arguments Example

```
import random
from pyats import aetest

# random.random() generates a float number between 0 and 1
parameters = {
    'number': random.random,
}

class Testcase(aetest.Testcase):

    @aetest.test
    def test(self, number):
        print(self.parameters['number']) # still an object if viewed from
      self.parameters
        # <built-in method random of Random object at 0x91e2fc4
        # test whether the generated number is greater than 0.5
         assert number > 0.5
```

AEtest - Test Parameters - Parameterizing Functions

Much like callable arguments, but has the following additional features:

- Ability to pass arguments
- The current section object can be passed using an argument named ‘section’
  - Section attributes (i.e. uid, parent, result) can influence the return values

Parameterizing Functions Example

```
import random
from pyats import aetest

# a parametrized function called ‘number’ that
# accepts an upper and lower bound
# random.randint() is used to generate a random number within
the bounds
@aetest.parameters.parametrize(lower_bound=1, upper_bound=100)
def number(lower_bound, upper_bound):
    return random.randint(lower_bound, upper_bound)

# accepts the current section as input, and
# returns 9999 when the section uid is 'expected_to_pass', or
0 otherwise.
@aetest.parameters.parametrize
def expectation(section):
    if section.uid == 'expected_to_pass':
        return 9999
    else
         return 0
```

```
class Testcase(aetest.Testcase):

    # parametrized functions must be passed as function
arguments to be evaluated
    # (just like callable parameters)

    # this section is expected to pass
    # the generated number is between 1 and 100, and the
    # expectation is 9999 (section uid is "expected_to_pass")
    @aetest.test
    def expected_to_pass(self, number, expectation):
        # test whether expectation is > than generated number
        assert expectation > number

    # this section is expected to fail
    # the generated number is still between 1 and 100, but the
    # expectation is 0 (section uid is not "expected_to_pass")
    @aetest.test
    def expected_to_fail(self, number, expectation):
        # test whether expectation is > than generated number
          assert expectation > number
```

AEtest - Test Parameters - Reserved Parameters

Parameters generated at runtime by AEtest

Provides supported method to accessing `aetest` internals

Only available is passed as a keyword argument to test methods

If a normal parameter is created with the same name as a reserved parameter, the normal parameter is only accessible via the parameters property

Reserved Parameters Example

```
from pyats import aetest

# applicable to other TestContainer classes (Testcase and CommonCleanup) as well
class CommonSetup(aetest.CommonSetup):

    # access reserved parameters by providing their names as keyword arguments to methods
    @aetest.subsection
    def subsection_one(self, testscript, section, steps):
        # testscript object has an attribute called module which is this testscript's module
        print(testscript.module)
        # <module 'example_script' from '/path/to/example.py'>

        # current section object is Subsection and subsections have a unique uid
        print(section.uid)
        # subsection_one

        # steps object enables the usages of steps
        with steps.start('a new demo step'):
            pass
```

Test Execution

Test Execution Methods

Standalone - Best suited for development. All logging is sent to standard output (stdout)

Easypy - Runtime environment built within pyATS that allows you to run tests in Jobfiles. It produces logs and archives for regression testing and reporting.

Test Execution - Argument Propagation

`aetest` uses the `argparse` module to parse command-line arguments that are stored in `sys.argv`

Allows users to provide additional arguments to the testscript

```
$ python script.py --loglevel INFO --my_arg 1 --your_arg 2
sys.argv = ['python script.py', '-loglevel=INFO', '-my_arg=1', '-your_arg=2']
```

`loglevel` is a known argument to `aetest`

`my_arg=1` and `your_arg=2` are passed in as testscript arguments

Test Execution - Standalone Execution

Directly calling `aetest.main()` in a script

Indirectly calling `aetest.main()` by calling `__main__`

- Keyword arguments passed to `aetest.main()` are used as testscript parameters during execution
- Limited to a single script
- All logging is redirected to stdout and stderr
- No logs or archives created

Standalone Execution Example

```
import logging
from pyats import aetest

# your testscript sections, testscases & etc
# ...

# add the following to the end of your testscript
if __name__ == '__main__':

    # change pyATS log level to debug for testing purposes
    logging.getLogger('pyats.aetest').setLevel(logging.DEBUG)

    # aetest.main() api starts the testscript execution.
    # defaults to aetest.main(testable = '__main__')
     aetest.main()
```

Test Execution - Easypy Execution

Recommended for production testing

Multiple aetest testscripts can be executed at once in a Jobfile. Each testscript is considered a *task* in the job. TaskLogs, result report and archives generated

TaskLog - log file for tasks (testscripts)

Reporter - generates the following result files:

- JSON results file
- Results details XML file
- Results summary XML file

Easypy Execution Example

```
from pyats.easypy import run

# job file needs to have a main() definition
# which is the primary entry point for starting job files
def main():

    # run a testscript
    # ----------------
    # easypy.run() api defaults to using aetest as the test infrastructure
    # to execute the testscript. Eg, this is the exact same as doing:
    #   run(testscript='/path/to/your/script.py',
    #       testinfra = 'pyats.aetest')
     run(testscript='/path/to/your/script.py')
```

Results and Reporting

Standalone Reporting - Example Output

```
+-------------------------------------------------------------------------------+
|                          Starting common setup                                |
+-------------------------------------------------------------------------------+
+-------------------------------------------------------------------------------+
|                     Starting subsection subsection_one                        |
+-------------------------------------------------------------------------------+
              The result of subsection subsection_one is => PASSED
+-------------------------------------------------------------------------------+
|                          Starting testcase Testcase                           |
+-------------------------------------------------------------------------------+
+-------------------------------------------------------------------------------+
|                          Starting section test_one                            |
+-------------------------------------------------------------------------------+
                  The result of section test_one is => PASSED
                  The result of testcase Testcase is => PASSED
+-------------------------------------------------------------------------------+
|                              Detailed Results                                 |
+-------------------------------------------------------------------------------+
  SECTIONS/TESTCASES                                                    RESULT
--------------------------------------------------------------------------------
                                        .
   |-- CommonSetup                                                    PASSED
   |   |-- subsection_one                                             PASSED
   `-- Testcase                                                       PASSED
       |-- test_one                                                   PASSED
+-------------------------------------------------------------------------------+
|                                  Summary                                      |
+-------------------------------------------------------------------------------+
  Number of ABORTED                                                        0
  Number of BLOCKED                                                        0
  Number of ERRORED                                                        0
  Number of FAILED                                                         0
  Number of PASSED                                                         2
  Number of PASSX                                                          0
  Number of SKIPPED                                                        0
--------------------------------------------------------------------------------
```

Pearson

Easypy Reporting - Example Output

```
.
├── JobLog.basic_jobfile
├── OReilly-Live-Training.report
├── ResultsDetails.xml
├── ResultsSummary.xml
├── TaskLog.Task-1
├── basic_jobfile.abstract
├── basic_jobfile.py
├── cat8000v-cli-1752371038.log
├── easypy.configuration.yaml
├── env.txt
├── pyats.configuration.yaml
├── rerun.results
├── results.json
└── testbed.clean.extended.yaml
```

```
+------------------------------------------------------------------------------+
|                                Easypy Report                                 |
+------------------------------------------------------------------------------+
pyATS Instance   : /workspaces/oreilly-live-training-pyats/venv
Python Version   : cpython-3.12.1 (64bit)
CLI Arguments    : /workspaces/oreilly-live-training-pyats/venv/bin/pyats run job
basic_jobfile.py
User             : codespace
Host Server      : codespaces-93f503
Host OS Version  : Ubuntu 24.04 noble (x86_64)

Job Information
    Name         : OReilly-Live-Training
    Result       : FAILED
    Start time   : 2025-07-13 01:43:58.271776+00:00
    Stop time    : 2025-07-13 01:44:15.348357+00:00
    Elapsed time : 0:00:18
    Archive      : /home/codespace/.pyats/archive/25-
07/basic_jobfile_2025Jul13_01_43_52_435239.zip

Total Tasks    : 1

Overall Stats
    Passed     : 2
    Passx      : 0
    Failed     : 1
    Aborted    : 0
    Blocked    : 0
    Skipped    : 0
    Errored    : 0

    TOTAL      : 3

Success Rate   : 66.67 %

Section Stats
    Passed     : 4
    Passx      : 0
    Failed     : 1
    Aborted    : 0
    Blocked    : 0
    Skipped    : 0
    Errored    : 0

    TOTAL      : 5

Section Success Rate   : 80.00 %

+------------------------------------------------------------------------------+
|                            Task Result Summary                               |
+------------------------------------------------------------------------------+
Task-1: basic_testscript                                                 FAILED
Task-1: basic_testscript.common_setup                                    PASSED
Task-1: basic_testscript.Basic_TC                                        FAILED
Task-1: basic_testscript.common_cleanup                                  PASSED

+------------------------------------------------------------------------------+
|                            Task Result Details                               |
+------------------------------------------------------------------------------+
Task-1: basic_testscript                                                 FAILED
|-- common_setup                                                         PASSED
|   `-- connect_to_devices                                               PASSED
|-- Basic_TC                                                             FAILED
|   |-- check_device_os_version                                          FAILED
|   |-- check_routes                                                     PASSED
|   `-- check_interface_status                                           PASSED
`-- common_cleanup                                                       PASSED
    `-- disconnect_from_devices                                          PASSED
```

pyATS AEtest Infrastructure Section Exercise

Install pyATS and the pyATS library (Genie)

Review a pyATS testscript and step through the different types of test sections and test parameters.

We will also execute the testscript and review the test results.

Q+A

5-min Break

Section 3: pyATS Library (Genie)

3.1 Introduction to the pyATS Library (Genie)

3.2 pyATS Parsers, APIs, and Models

3.3 pyATS Clean

3.4 pyATS Blitz

3.5 pyATS Health Check

Introduction to the pyATS library (Genie)

Introduction to the pyATS Library (Genie)

Built on top of pyATS. Provides the “tooling” for network engineers to extract network data and create reusable tests

Provides a command-line interface (CLI) to interact with devices

Provides parsers, APIs, and data models that model configuration and operational data

Other features: Genie Harness, pyATS Clean, pyATS Blitz, pyATS Health Check, Robot Framework integration

pyATS Library (Genie) - Genie CLI

`genie parse` - Execute and parse device output

`genie learn` - Learn device feature and store as a “snapshot”

`genie diff` - Compare snapshots (from genie learn)

`genie run` - Execute commands provided by Genie Harness

`genie shell` - Loads testbed file into Genie testbed object and a Pickle file.

`genie dnac` - Communicate with Cisco Catalyst Center (formerly DNA Center)

`pyats create` - Easy way to create parsers, testbeds, and triggers.

Example: Generate testbed YAML file from CSV/Excel file

pyATS Library (Genie) - Parsers/APIs/Models

Parsers - Structures raw device output

```
Python API: testbed.devices['nx-osv-1'].parse('show version')
Genie CLI: genie parse "show version" --testbed-file
/path/to/testbed.yaml --devices nx-osv-1
```

APIs - Provides “shortcut” to specific action/output

```
testbed.devices['nx-osv-1'].api.shut_interface(‘Loopback0’)
```

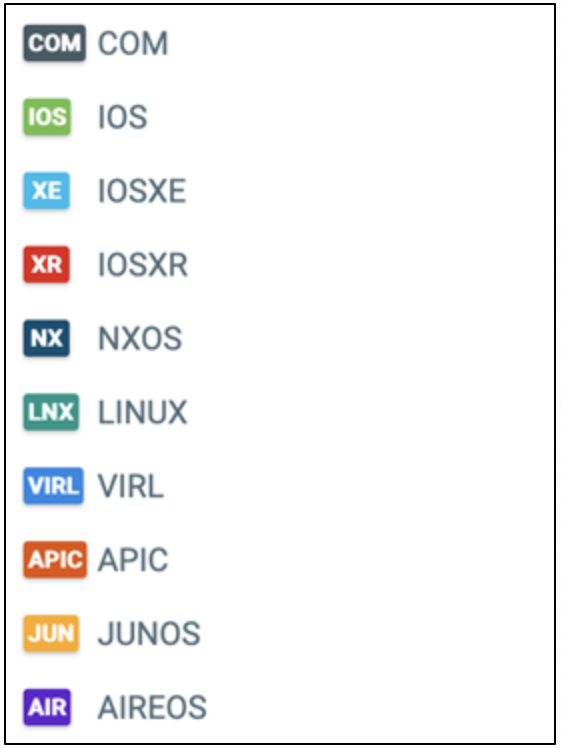
Models - Vendor-agnostic data structure for a given device feature

- **Conf** - Configuration data
- **Ops** - Device operational state data

pyATS Library (Genie) - Parsers and APIs

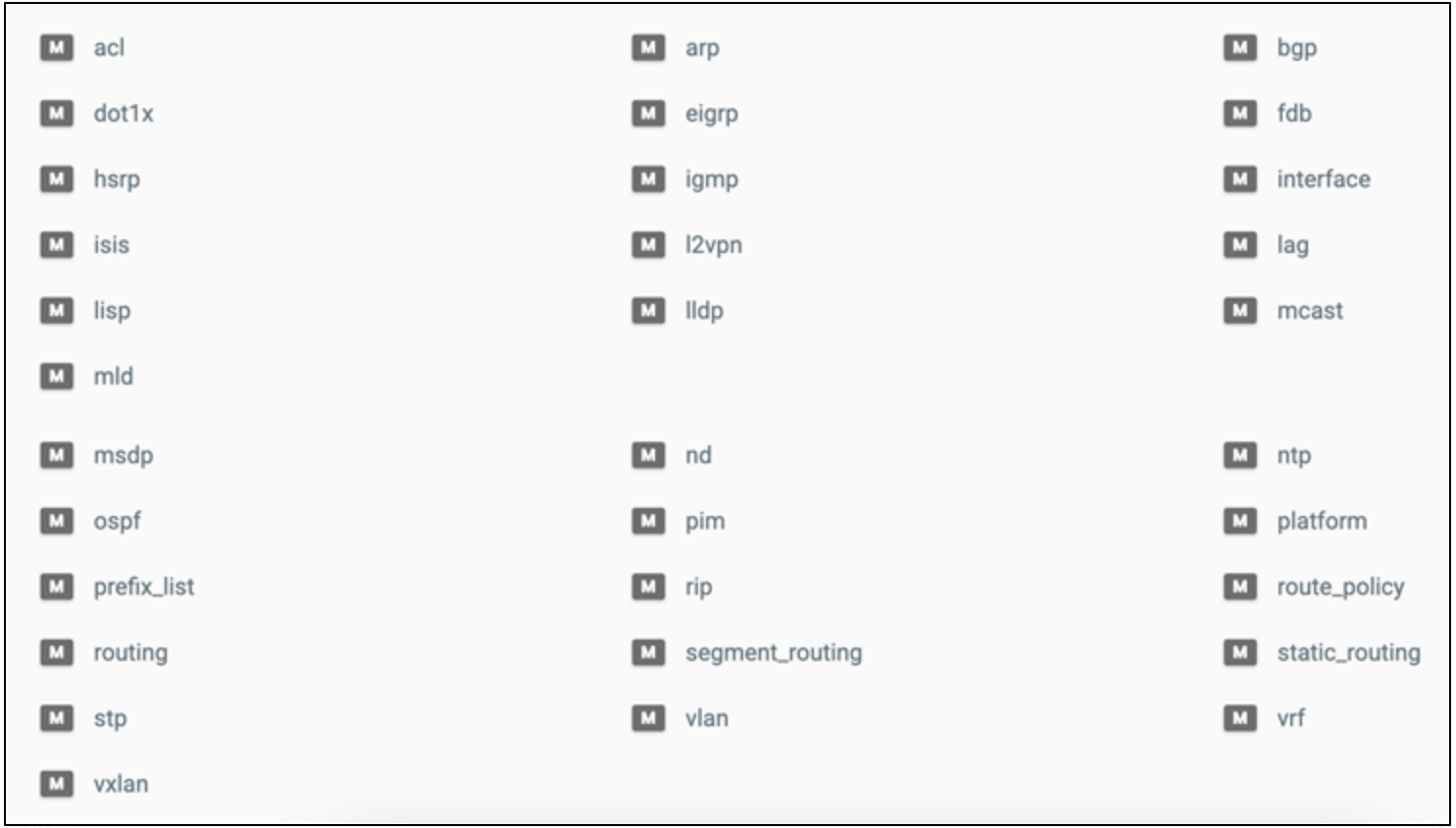
Device Parser Platforms

Device API Platforms



Pearson

pyATS Library (Genie) - Available Models



Pearson

*As of May 2025

Conf Object - Cisco NXOS Interface

```
# Create an NXOS interface
nxos_interface = Interface(device=uut, name='Ethernet4/3')
# Add some configuration
nxos_interface.ipv4 = '200.1.1.2'
nxos_interface.ipv4.netmask ='255.255.255.0'
nxos_interface.switchport_enable = False
nxos_interface.shutdown = False

# Verify configuration generated
print(nxos_interface.build_config(apply=False))
# interface Ethernet4/3
#  no shutdown
#  no switchport
#  ip address 200.1.1.2 255.255.255.0
#  exit
```

Conf Object - Cisco IOSXE Interface

```
iosxe_interface = Interface(device=iosxe_device, name='GigabitEthernet1/0/4')
# Add some configuration
iosxe_interface.ipv4 = '200.1.1.2'
iosxe_interface.ipv4.netmask ='255.255.255.0'
iosxe_interface.switchport_enable = False
iosxe_interface.shutdown = False

# Verify configuration generated
print(iosxe_interface.build_config(apply=False))
# interface Ethernet4/3
#  ip address 200.1.1.2 255.255.255.1
#  no shutdown
#  exit
```

Conf Object - Configuring Multiple Devices

```
# Assumes configuration is already built for multiple devices

# Verify what will applied on the devices
testbed.build_config(apply=False)

# Apply config to devices
testbed.build_config()
```

Ops Object - Learning Device Features

Learn (Python API)

```
# Connect to testbed device
uut = testbed.devices['uut']
uut.connect()

# Instantiate the OPS object
interface = Interface(device=uut)

# This will send many show command to learn the
operational state of interfaces for this device
interface.learn()

print(interface.info)
```

Learn (Genie CLI)

```
# From CLI
genie learn interface --testbed-file testbed.yaml --
devices uut

genie learn <feature1> <feature2> <featureN> --
testbed-file testbed.yaml --output
features_snapshots/
```

Ops Object - Finding Specific Information

Find - Reduces the amount of looping when searching data

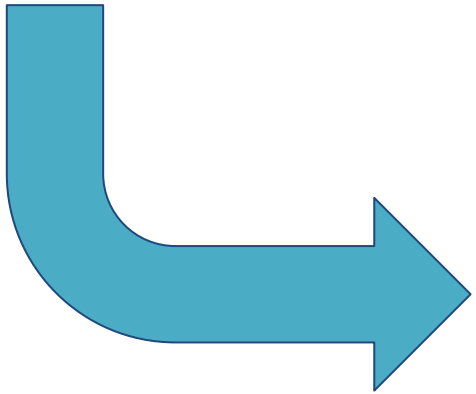
```
# Assumes device is connected
# Learn about the operational state of the device's interfaces
interface.learn()

# Let's get all the up interfaces
from pyats.utils.objects import R, find
req1 = R(['info', '(.*)', 'oper_status', 'up'])

find(interface, req1, filter_=False)
# Output:
# [('up', ['info', 'loopback1', 'oper_status']),
#  ('up', ['info', 'mgmt0', 'oper_status']),
#  ('up', ['info', 'loopback0', 'oper_status']),
#  ('up', ['info', 'Ethernet2/1', 'oper_status']),
#  ('up', ['info', 'Ethernet2/2', 'oper_status'])]
```

Ops Object - Comparing Snapshots with Diff

```
# Initial pre-change snapshot
genie learn interface --testbed-file testbed.yaml --devices uut --output snapshot1/

… making changes to device interface(s) …

# Post-change snapshot
genie learn interface --testbed-file testbed.yaml --devices uut --output snapshot2/

# Compare pre- and post-change snapshots
genie diff snapshot1 snapshot2
```

Comparing Snapshots - Example Output

```
genie diff dir1 dir2 --output diff1
1it [00:00,  8.44it/s]
+==============================================================================+
| Genie Diff Summary between directories dir1/ and dir2/                       |
+==============================================================================+
|  File: bgp_nxos_nxos-osv-1_ops.txt                                           |
|   - Diff can be found at diff1/diff_bgp_nxos_nxos-osv-1_ops.txt              |
|------------------------------------------------------------------------------|
```



diff1/diff_bgp_nxos_nxos-osv-1_ops.txt

```
--- dir1/bgp_nxos_nxos-osv-1_ops.txt
+++ dir2/bgp_nxos_nxos-osv-1_ops.txt
info:
 instance:
  default:
   vrf:
    default:
     neighbor:
      50.1.1.101:
       address_family:
        ipv4 multicast:
+         session_state: active
-         session_state: idle
        ipv4 unicast:
+         session_state: active
-         session_state: idle
      bgp_session_transport:
        connection:
+         state: active
-         state: idle
+       session_state: active
-       session_state: idle
```

pyATS Clean

What is pyATS Clean?

"Resets" a network device before/after testing

Initialize a device with a specific software image and/or configuration

Removes old, unwanted configuration from a device

pyATS Clean Scenarios

Recovering from a bad state during testing

Re-apply baseline configuration after testing

Write erase and reloading a device

Bring up a device with a new software image

Verify connectivity to a device/server before testing

pyATS Clean Framework

Define a “Clean” workflow using ***Stages*** to define individual steps

pyATS Clean workflows are built using YAML files

Device APIs and CLI commands are used to execute Clean commands

To Execute: Testbed YAML + Clean YAML

pyATS Clean Stages

Allows you to break down Clean workflows into modular steps

Abstractions for common tasks (i.e. connect, apply_configuration, reload)

Clean Stages with pyATS: https://pubhub.devnetcloud.com/media/genie-feature-browser/docs/#/clean

pyATS Clean - Power Cyclers

Power cyclers - PDUs, UPS devices, ESXi

Allows you to force reset/power off/on devices when you lose access to a device under testing

Break the boot sequence to bring device to ROMMON mode to boot device to “golden” software image

pyATS Clean - Power Cyclers - Testbed Example

```
Testbed Example
---------------
devices:
  PE1:
    os: nxos
    platform: n9k
    custom:
      abstraction:
        order: [platform, os]
    peripherals:
      power_cycler:
        - type: apc
          connection_type: snmp
          host: 127.0.0.1
           outlets: [20]
```

pyATS Clean - YAML - Cleaner Class

```
cleaners:
    # Cleaner class `PyatsDeviceClean`
    PyatsDeviceClean:
      # The module is where the cleaner class above can be found
       module: genie.libs.clean
```

pyATS Clean - YAML - Adding Devices

```
cleaners:
    PyatsDeviceClean:
      module: genie.libs.clean
      # You must include devices in the list below for them to be cleaned
      devices: [PE1]

# Devices must be defined in the testbed
devices:
    PE1:
```

pyATS Clean - YAML - Defining a Clean Stage

```
cleaners:
    PyatsDeviceClean:
      module: genie.libs.clean
      devices: [PE1]

devices:
    PE1:
        connect:
            timeout: 100

        apply_configuration:
            configuration: logging host 10.1.1.1

        order:
        - connect
        - apply_configuration
```

pyATS Clean - Execution

Both methods below require Clean YAML and Testbed YAML files

Running before a pyATS script

```
pyats run job </path/to/job.py> --testbed-file </path/to/testbed.yaml>
--clean-file </path/to/clean.yaml> --invoke-clean
```

Running without a pyATS script

```
pyats clean --testbed-file </path/to/testbed.yaml> --clean-file
</path/to/clean.yaml>
```

pyATS Clean - Viewing Logs

Best method is using pyATS log viewer

After the device cleaning is finished, run the command:

```
pyats logs view
```

pyATS Clean - Running within a pyATS TestScript

Utilizing the Device's API attribute:
device.api.clean.<stage>(<stage_arguments>)

```
from pyats import aetest

  class MyTestcase(aetest.Testcase):

    @aetest.test
      def my_test(self, steps, testbed):
        device = testbed.devices['PE1']
        device.connect()
        device.api.clean.install_image(images=['bootflash:/image.bin'],
          save_system_config=True)
```

pyATS Blitz

What is pyATS Blitz?

Create testcases using YAML syntax vs Python code

Abstraction layer for non-programmers

Testcases are developed in **trigger datafiles** (YAML)

pyATS Blitz - Trigger Datafiles

Testcases are made up **test sections**

Actions - tasks in test sections to perform testing.

Map to backend code for execution

Accept keyword arguments, much like a Python function

pyATS Blitz - Example Test Section with Action

```
# Section name
  - apply_configuration:
      # List of actions
      - configure:
          device: R3_nx
          command: |
            router bgp 65000
            shutdown
      - sleep:
           sleep_time: 5
```

pyATS Blitz - Available Actions

List of blitz actions

- execute
- configure
- configure_dual
- parse
- api
- tgn
- rest
- sleep
- learn
- print
- bash_console
- configure_replace
- save_config_snapshot
- restore_config_snapshot
- yang_snapshot
- yang_snapshot_restore
- run_genie_sdk
- diff
- compare
- dialog
- yang

pyATS Blitz - Action Output

Save the entire output

Save part of the output using filters:

- Dq (Dictionary query)
- Regex
- Regex findall
- List

pyATS Blitz - Variables

Saving a variable

```
- api:
    device: PE1
    function: get_interface_mtu_size
    save:
      - variable_name: api_output
    arguments:
      interface: GigabitEthernet1
```

Using a variable

```
- configure:
    device: PE1
    command: |
      router bgp '%VARIABLES{api_output}'
```

pyATS Blitz - Execution

CLI

```
pyats run genie --trigger-datafile <path_to_blitz_datafile> --trigger-uids 'test1' --testbed-
file testbed.yaml
```

Easypy Job

```
pyats run job <path_to_job_file>
```

pyATS Blitz - Easypy Jobfile Example

```
import os
from genie.harness.main import gRun
from pyats.datastructures.logic import And, Not, Or


def main():

    gRun(
        trigger_datafile=<path_to_blitz_datafile>,
        trigger_uids = ['test1', 'test2'],  # name of the tests you wish to run
        testbed=<path_to_testbed>,
     )
```

Pearson

pyATS Health Check

Pearson

pyATS Health Check

Checks to ensure device is healthy:

- Check CPU utilization
- Memory usage
- Search log messages
- Detect core dump file

Ran as post-processors after each testcase

Ability to create your own health checks

pyATS Health Check - Usage

Use **`--health-checks`** with pyATS command

List the health checks you want to execute

```
pyats run job <job file> --testbed-file /path/to/testbed.yaml --health-checks cpu
memory logging core
```

pyATS Health Check - Core Dump File

Core dump file detection is enabled by default

Copy file from device to a remote server using **`--health-remote-device`**

Protocols supported: **`http`**, **`scp`**, **`tftp`**, **`ftp`**

```
pyats run job <job file> --testbed-file /path/to/testbed.yaml --health-checks cpu
memory logging core --health-remote-device name:myserver path:/tmp/ protocol:http
--health-mgmt-vrf iosxe:None nxos:mgmt
```

*Must have '**myserver**' specified in testbed*

pyATS Health Check - Additional Settings

Change CPU/Memory thresholds: `--health-threshold cpu:75 memory:8`

Change logging keywords (only looks for *traceback* by default):

`--health-show-logging-keywords "nxos:['Crash', 'CRASH']"`

Location of core dump files: `--health-core-default-dir "iosxe:['harddisk0:/core']"`

pyATS Health Check - Additional Settings Cont.

Run health checks against certain devices: `--health-devices R1 R2 R3`

Send Webex notification (only sent when health check(s) fails): `--health-webex --webex-token <webex token> --webex-space <webex space id>`

*--webex-email can also be provided

pyATS Library (Genie) Section Exercise

Review the different Genie features using the Genie CLI.

We will also take a look at an example of a pyATS Blitz trigger datafile.

Q+A

5-min Break

Section 4: Integrating Testing into Network Automation Workflow

4.1 Overview of Continuous Integration/Continuous Delivery (CI/CD)

4.2 Explain how CI/CD pipelines can be used for the network

4.3 Example CI/CD Pipeline includes pyATS for network testing

CI/CD Overview

What is CI/CD?

Continuous **I**ntegration**/C**ontinuous **D**eployment/Delivery

Source: https://www.redhat.com/en/topics/devops/what-is-ci-cd

Why CI/CD?

Identify and fix bugs quicker

Enforce code standards and testing

Push new features/functionality at a faster pace

Source: https://www.redhat.com/en/topics/devops/what-is-ci-cd

CI/CD Services

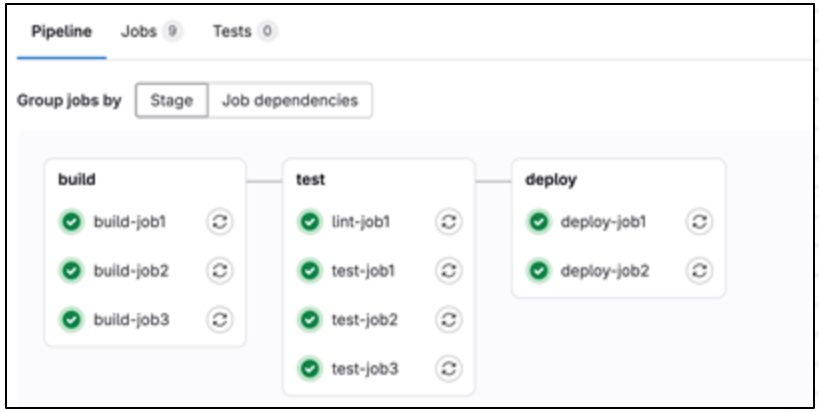
- GitHub Actions
- GitLab CI/CD
- CircleCI
- Azure DevOps
- Jenkins

CI/CD Pipeline Definitions

- GitHub Actions
  - Events
  - Workflows
  - Jobs
    - Steps
  - Actions
  - Runners

GitHub Actions Example

- GitLab CI/CD
  - Stages
  - Jobs
  - Scripts
  - Runners

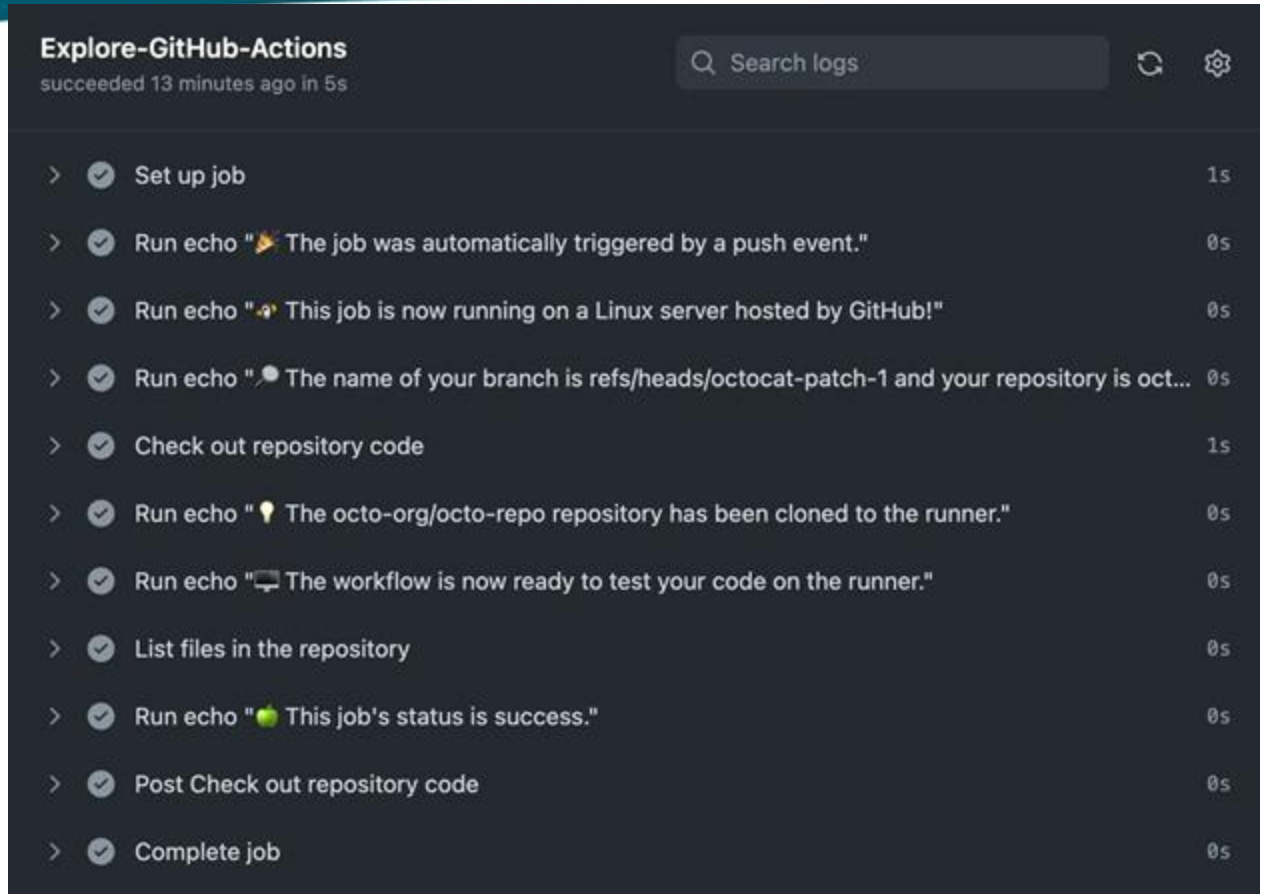


GitLab CI/CD Example

GitHub Actions Pipeline Definition (.github/workflows)

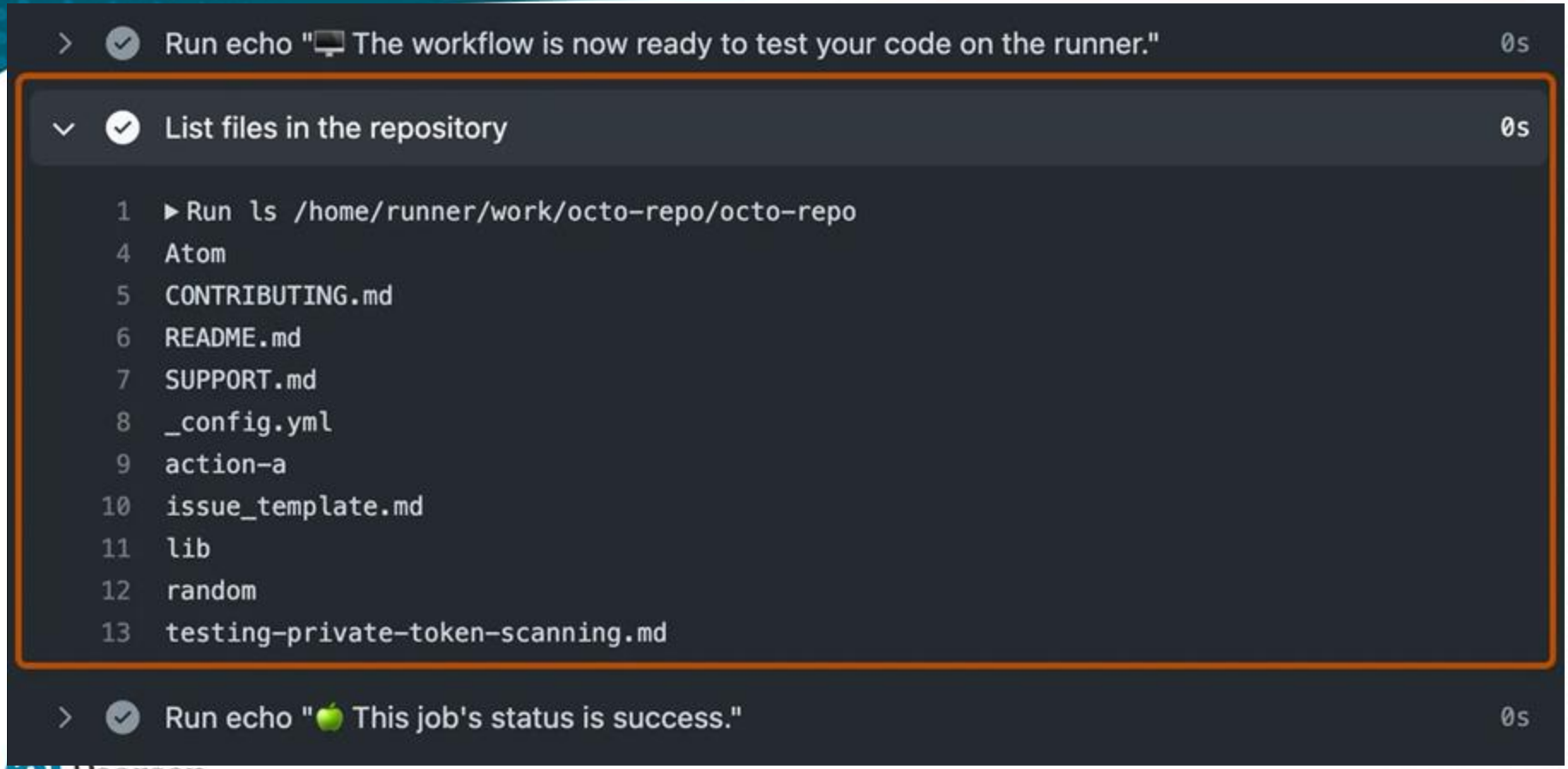
```yaml
# .github/workflows/demo.yml
name: GitHub Actions Demo
run-name: ${{ github.actor }} is testing out GitHub Actions 🚀
on: [push]
jobs:
 Explore-GitHub-Actions:
   runs-on: ubuntu-latest
   steps:
     - run: echo "🎉 The job was automatically triggered by a ${{ github.event_name }} event."
     - run: echo "🐧 This job is now running on a ${{ runner.os }} server hosted by GitHub!"
     - run: echo "🔎 The name of your branch is ${{ github.ref }} and your repository is ${{ github.repository }}."
     - name: Check out repository code
       uses: actions/checkout@v4
     - run: echo "💡 The ${{ github.repository }} repository has been cloned to the runner."
     - run: echo "🖥️ The workflow is now ready to test your code on the runner."
     - name: List files in the repository
       run: |
         ls ${{ github.workspace }}
     - run: echo "🍏 This job's status is ${{ job.status }}."
```

GitHub Actions Pipeline Output



Pearson

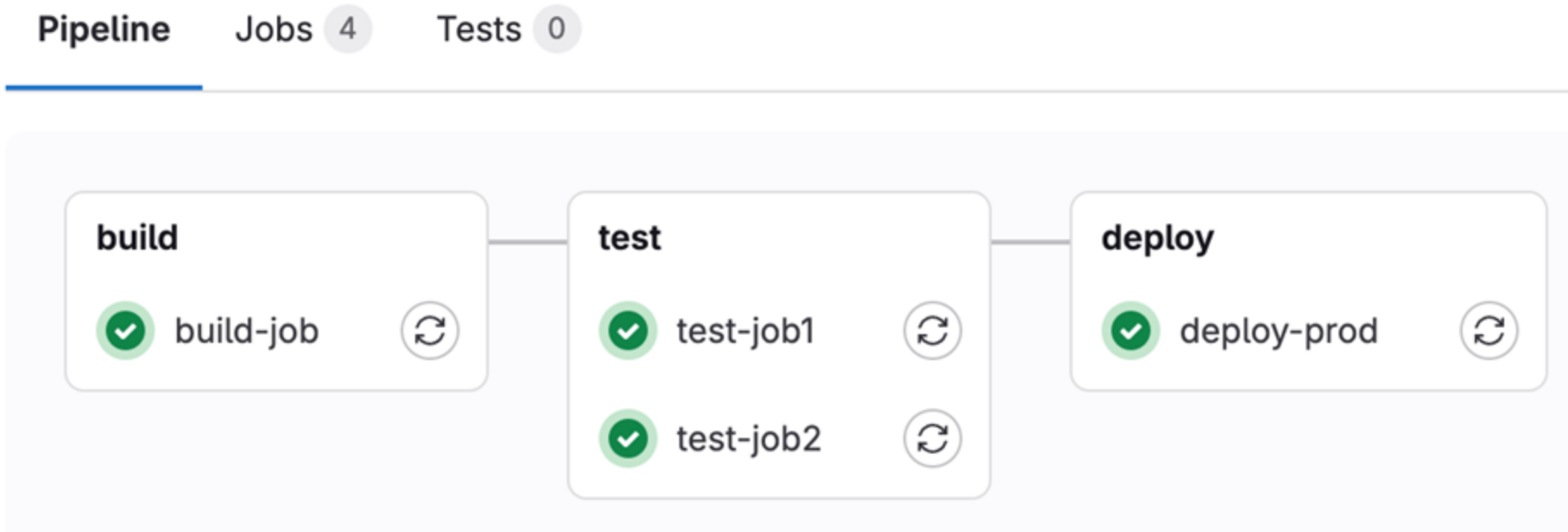
GitHub Actions Pipeline Output Logs



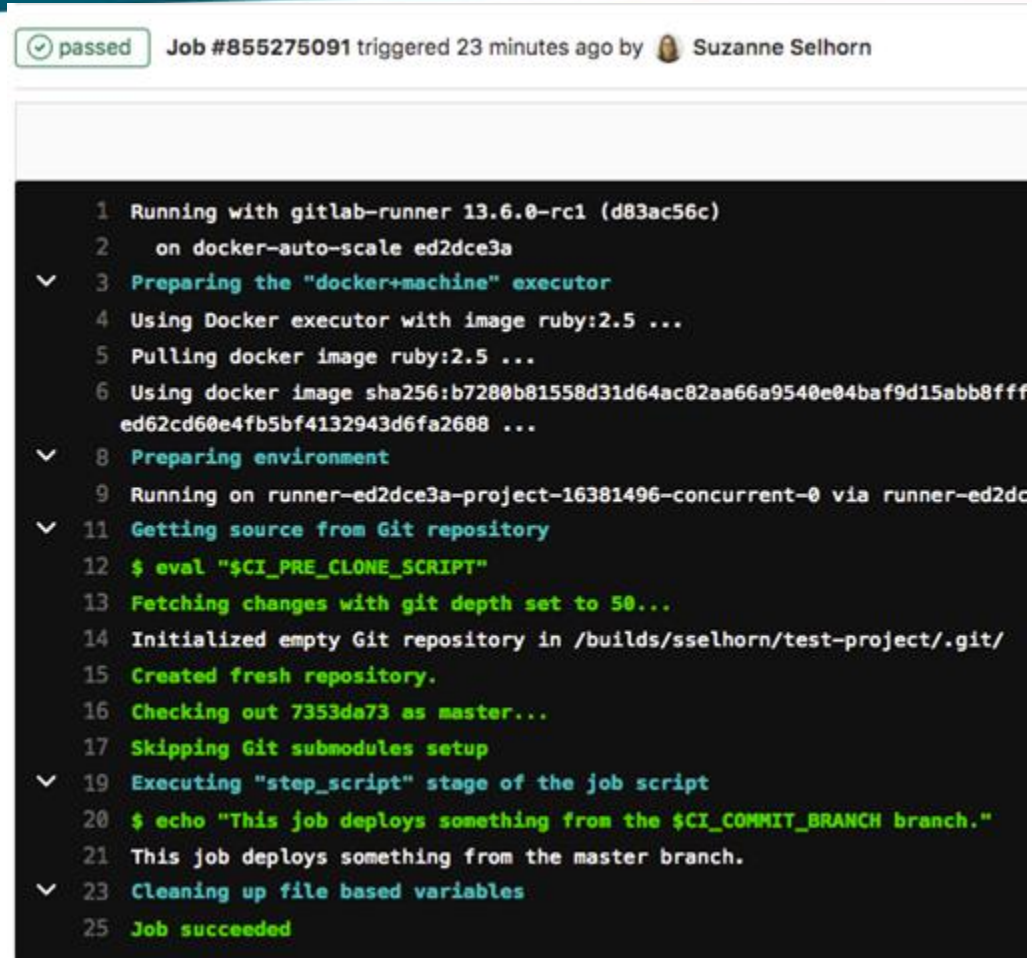
GitLab CI/CD Pipeline Definition (.gitlab-ci.yml)

```yaml
# .gitlab-ci.yml
build-job:
 stage: build
 script:
   - echo "Hello, $GITLAB_USER_LOGIN!"

test-job1:
 stage: test
 script:
   - echo "This job tests something"

test-job2:
 stage: test
 script:
   - echo "This job tests something, but takes more time than test-job1."
   - sleep 20

deploy-prod:
 stage: deploy
 script:
   - echo "This job deploys something from the $CI_COMMIT_BRANCH branch."
 environment: production
```

GitLab CI/CD Pipeline Output



GitLab CI/CD Pipeline Output Logs



Pearson

Applying CI/CD to Networking

Applying CI/CD to Networking

1. Lint code/playbooks with network changes
2. Run pre-change tests against the network
    a. Produce artifact
3. Apply proposed network changes to a lab environment automatically
4. Deploy changes to the production network
5. Run post-change tests/checks
    a. Produce artifact
6. Compare pre- and post-changes using artifacts from each job

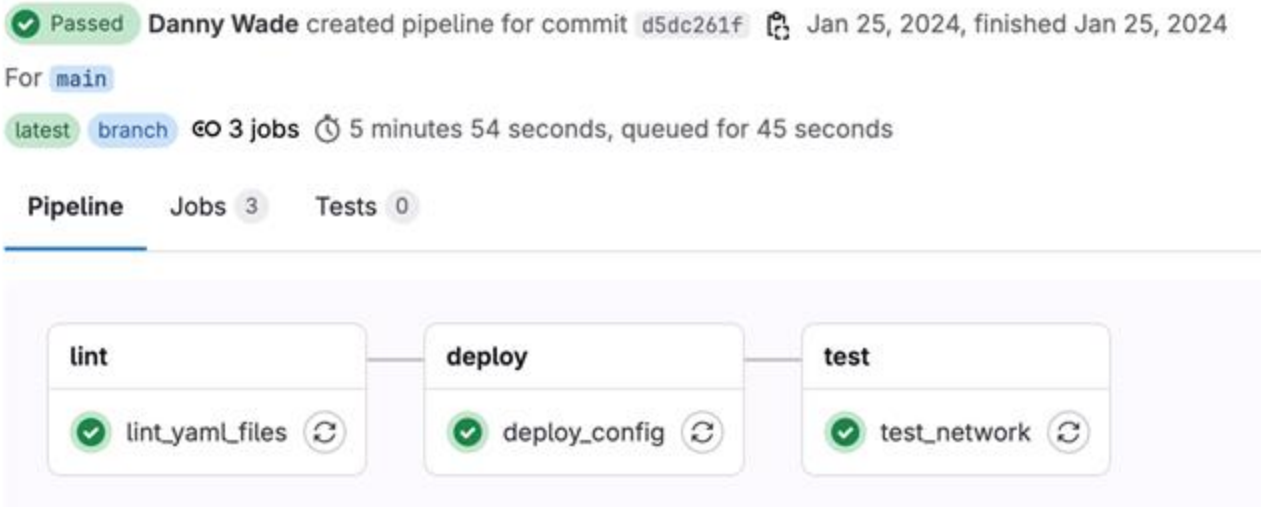
CI/CD Pipeline Example with pyATS

NetDevOps CI/CD Pipeline Definition (GitLab)

```yaml
# Lint Ansible playbooks
lint_yaml_files:
  stage: lint
  script:
    - ansible-lint routers.yaml
    - pyats validate datafile net_testing/blitz.yaml
    - pyats validate testbed net_testing/testbed.yaml

# Deploy configuration to network devices using Ansible
deploy_config:
  stage: deploy
  script:
    - ansible-playbook routers.yaml

# Test network using pyATS Blitz
test_network:
  stage: test
  script:
    - pyats run job pyats_jobfile.py --health-checks cpu memory logging core --html-logs pyats_html_logs --archive-dir pyats_archive_lo
  artifacts:
    name: pyats_test_artifacts
    untracked: false
    when: on_success
    expire_in: 30 days
    paths:
      - $CI_PROJECT_DIR/pyats_html_logs
      - $CI_PROJECT_DIR/pyats_archive_logs
```

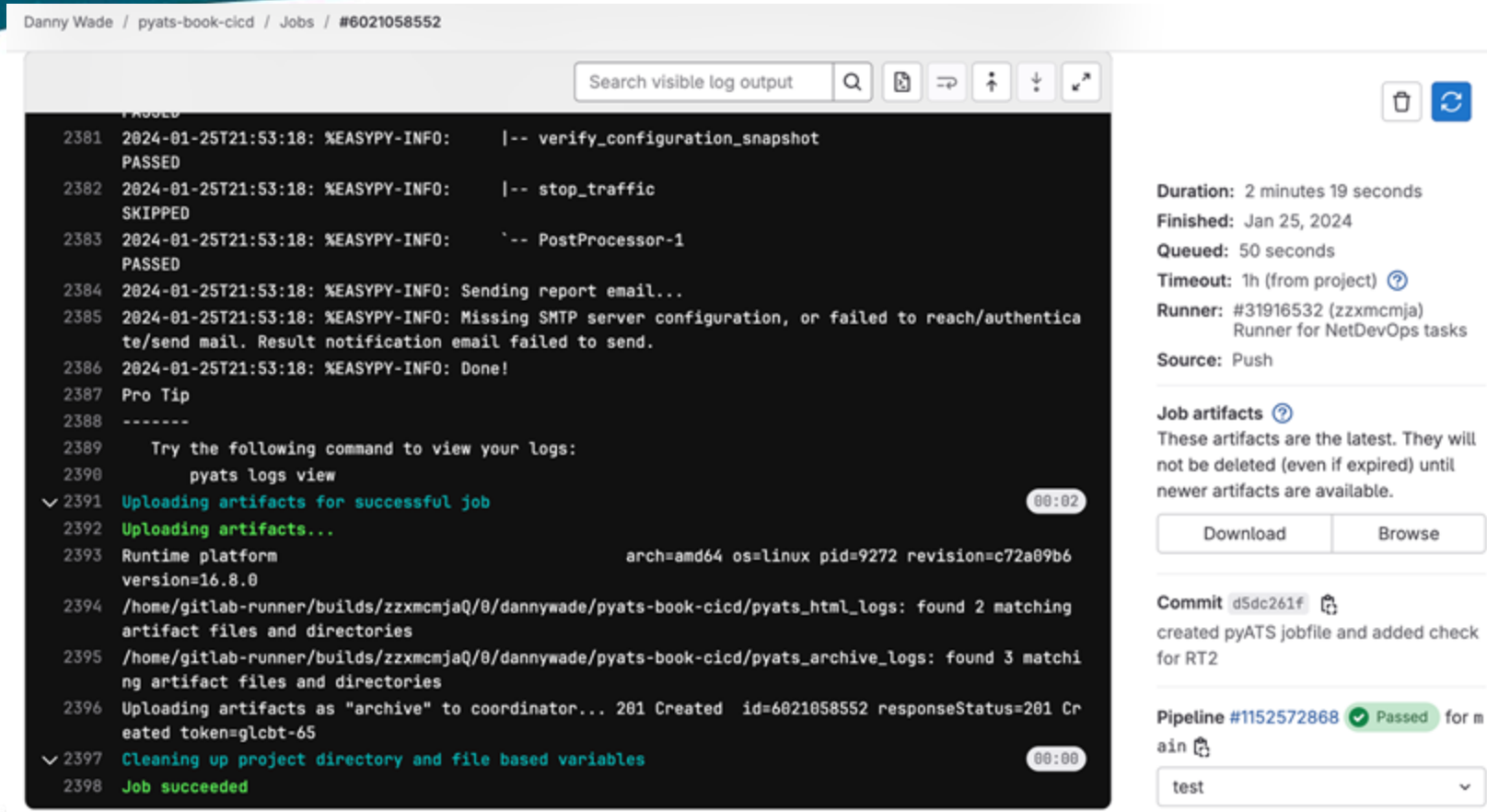
pyATS in a CI/CD Pipeline

- pyATS is used in the ‘test_network’ job that occurs in the ‘test’ stage
- Pipeline runs on any event (commit or pull request)



Pearson

CI/CD Pipeline Output - pyATS Logs



Artifacts - Folder Structure

```
├── pyats_archive_logs
│   └── 24-01
│       └── pyats_jobfile.2024Jan25_21:52:40.988104.zip
└── pyats_html_logs
    └── TaskLog.pyats_jobfile.html
```

Integrating Testing into Network Automation Workflow Section Exercise

Show a CI/CD pipeline that uses pyATS to test network changes.

Wrap Up

- Software Testing Basics
- pyATS Infrastructure
- pyATS Library (Genie)
- Applying pyATS in a CI/CD Pipeline
- Questions and feedback

Please remember to complete the survey!

Next Steps

- **Apply** what you learned today
  - Spin up a sandbox/lab environment and practice!
- Feel free to reach out to me with any follow-up beyond the course!

Additional Resource(s):

- *pyATS documentation*
- *pyATS library (Genie) documentation*
- *Cisco Network Test and Automation Solution: Data-Driven and Reusable Testing for Modern Networks* by John Capobianco and Dan Wade