ChatGPT

pyATS and Network Validation

Overview of pyATS

pyATS (Python Automated Test System) is a network automation testing framework developed by Cisco. It provides a "batteries-included" environment for writing and running test cases against network devices. A key component of pyATS is the **Genie** parsing library, which includes over 1,500 parsers to translate raw device output (from Cisco IOS, IOS-XR, NX-OS, Juniper JunOS, F5 BIG-IP, etc.) into structured data [1]. This means engineers can programmatically inspect device configurations and operational state without having to manually parse text output. PyATS was originally used internally at Cisco (released publicly in 2014) and is designed to support both CLI and API interactions for robust, scalable testing [2] [3]. By combining pyATS with Genie, you get a powerful test suite that can verify your network's health across multi-vendor environments and gather **stateful validation** of device status [4].

Core concepts in pyATS include the **testbed**, **testcases**, and **testscripts**. The **testbed** is a YAML inventory describing your devices, their connection details (IP, protocol, credentials), OS type (to select appropriate Genie parsers), and even how devices are linked in the topology [5] [6]. Once the testbed is defined, you can write a **testcase** focusing on a specific network feature or function. For example, you might create a testcase to verify that **BGP is up and operational** across certain routers. Within that testcase, you would include smaller checks – e.g. confirming all BGP neighbor sessions are established and that expected BGP routes appear in the routing table (RIB) of each device [7]. These individual checks roll up into an overall result for the testcase. A collection of testcases can be organized in a **testscript** (a Python file) which can be run standalone or as part of a larger **job**. PyATS jobs allow grouping multiple testscripts and provide reporting and logging for all tests executed [8]. The framework produces detailed logs and can archive results (in XML/JSON) along with device logs, making it easier to troubleshoot failures [9]. In summary, pyATS offers a structured, modular approach to network testing – from simple pings to complex multi-device verifications – with the ability to parse device output into machine-checkable data.

Using pyATS for Network Validation

Network validation is about ensuring that the network's **operational state** matches expectations, especially after changes or deployments. pyATS excels in this area by automating what would traditionally be manual verification steps. For instance, if your manager asks *"Can we confirm feature X is configured and working on every device?"*, doing this by hand is tedious and error-prone [10]. With pyATS, you can automate such a validation across the network. The Genie library provides feature-specific models and APIs (for BGP, OSPF, interfaces, etc.) to **"learn"** the current network state and compare it to a desired state or baseline. PyATS can capture a snapshot of both **configuration and operational data** (like interface statuses, routing tables, protocol states) and serve as a known baseline [11] [12]. After a change, it can gather a new snapshot and automatically highlight any deviations or failures in meeting the expected outcomes [13] [4]. This before/after comparison capability is extremely useful when rolling out updates or new network links – it ensures nothing was broken or left unconfigured during the change.

Crucially, pyATS doesn't just check that configurations exist; it checks **operational status** and performance. It validates that the network is *actually functioning* as intended. For example, pyATS can automatically run ping tests or traceroutes to verify **reachability** between devices or to specific endpoints. In fact, the Genie parsers include a **ping parser** that makes it easy to assess connectivity and latency metrics. Instead of manually reading ping output, you can have pyATS run a ping and then assert that you got 100% success rate and that the round-trip latency is within acceptable bounds. The parser provides structured results including success rate and min/max/average response times in milliseconds [14], so your script can programmatically check if latency stays below a threshold (critical in environments like high-frequency trading). Similarly, there are parsers for many routing and BGP commands, allowing your tests to directly interpret outputs like `show ip bgp summary` or `show ip route`. This means your pyATS test can, for example, confirm that a BGP neighbor's state is "Established" and that a certain number of prefixes have been received, all via Python data structures rather than text scraping. PyATS' **stateful** nature also allows writing more complex tests – such as bringing down an interface or BGP process and verifying the network reacts as expected (as part of resilience or failover testing) [15]. In non-disruptive scenarios, you typically use pyATS in a read-only manner: connect to devices, gather data, and validate it against your criteria. The results can be output to console, logs, or even nice HTML reports for easy review. PyATS thus helps embed **test-driven practices** into network operations – every change can be accompanied by an automated test to ensure the network remains healthy and policy-compliant.

Example Scenario: Validating New Link Deployment in a FinTech Network

Scenario: Imagine a fintech company is deploying multiple new network links between its data centers and trading partners. These could be additional WAN circuits or direct peering links intended to improve redundancy and reduce latency (a common requirement in financial trading networks where milliseconds matter). After the network team provisions the new links (configuring interfaces, BGP neighbors, and routing policies), they want to **validate** that everything is working as intended *before* declaring the deployment successful. Here's how pyATS can be used to automate the validation of such a deployment:

1. **Testbed Definition:** First, update or create the pyATS **testbed** file to include the devices involved (routers, switches at the data centers, etc.) and the details of the new links. This means adding the new interfaces/IPs and ensuring credentials and device OS types are specified. The testbed can also describe the topology by linking interfaces between devices, which helps if the tests need to understand link relationships [5]. For our scenario, the testbed would list all relevant routers and the new link interfaces connecting them (including any BGP peer IPs on those links).

2. **Establish Baseline (Optional):** It's often useful to capture a **baseline** before the change. Using pyATS Genie, you could execute and parse commands like `show ip route`, `show bgp summary`, and latency tests **before** the new links are activated. This provides a point of comparison. For instance, prior to the change you might record that certain remote subnets are reached via an older link with X ms latency. After deploying the new link, you expect some of those routes to move to the new link and latency to improve. PyATS can save this baseline data (perhaps in a JSON file or simply keep in memory) to compare against post-deployment state [4]. (If no baseline is taken, you would at least have predefined expected values or thresholds for the tests below.)

3. **Post-Deployment Connectivity Check:** Once the new links are up and configured, pyATS connects to the devices and verifies basic **reachability**. Using the `ping()` API or executing ping commands via pyATS, the test script sends ICMP pings across the new links. For example, from DataCenter A's router to DataCenter B's router via the new link, and vice versa. The Genie ping parser will give structured results that the script can evaluate. We assert that the ping **success rate is 100%** (no packet loss) and that the **average latency** is within the expected range (e.g. "the average RTT should not exceed 5 ms"). If the new link is meant to be a low-latency path, we might also compare that the latency is lower than the old path's latency. All these checks are done in code – for instance, `ping_result = device.ping('1.2.3.4', count=5)` might return a Python dict with keys like `success_rate_percent` and `rtt_avg` that we compare against thresholds [14]. If any ping test fails or shows unexpected delay, pyATS will flag the test as failed, alerting us that reachability or performance is not as desired.

4. **BGP Peering Verification:** Next, the pyATS script verifies that all **BGP sessions** over the new links come up correctly. Using Genie, the script can execute commands like `show ip bgp summary` or equivalent on each router and parse the output. The structured data will include each BGP neighbor IP, the state (Idle, Active, Established, etc.), and numbers of prefixes exchanged. The test asserts that each new neighbor is in **Established** state and has exchanged the expected number of routes (or at least >0 prefixes received, depending on your criteria). Essentially, we are programmatically confirming the **BGP peerings** are live. If, say, a neighbor is stuck in "Idle" or "Active", pyATS will catch that. In our example, we expect that the data center routers have formed iBGP or eBGP sessions with each other or with the provider/partner over the new link. A pyATS testcase for BGP would include checks for neighbor relationships being up and routes present in the routing table [7]. For instance, one sub-test might iterate through all defined BGP neighbors in the testbed and assert `neighbor.state == 'Established'`. This gives a rapid, consistent way to verify BGP across multiple devices without manually logging into each one.

5. **Routing Table (RIB) Validation:** With BGP up, the **Routing Information Base (RIB)** on each device should now include routes learned over the new links. The pyATS script can retrieve the routing table (e.g. via `show ip route` or using Genie's higher-level **"learn routing"** feature) and ensure the expected prefixes are present and have the correct next-hops. For example, if the new link provides connectivity to certain networks, the test verifies that those network prefixes now appear in the routing table with the new link's interface or next-hop IP. If a route was previously learned via an older link and is now supposed to be learned via the new link (perhaps with a better BGP attribute), the test can confirm that the routing table's entry for that prefix points to the new link's next-hop (this implicitly checks that BGP policy and path selection worked). PyATS/Genie's structured output makes such checks straightforward – you can query the parsed route data structure for a specific prefix or for all routes with certain attributes. In code, it might look like: `routes = device.parse("show ip route")` returning a dictionary of routes, then asserting the presence or attributes of specific routes. In our fintech scenario, you might specifically ensure that the routes to trading partner networks are now learned via the new low-latency link.

6. **BGP Policy Enforcement:** If there are BGP policies (route filters, communities, local preference rules) associated with these new links, the validation should confirm those are in effect. For instance, suppose the network has a policy that routes learned from a partner over the new link must have a certain community tag, or that only specific prefixes are allowed in. The pyATS script can check the **BGP RIB** or BGP table for the presence or absence of routes accordingly. Using Genie parsers like

`show bgp ...` or device APIs, the test could verify that all received routes over the new link have the expected community value, or that no unauthorized prefixes are present in the routing table. If the policy was to prefer the new link for certain traffic (e.g. via local-preference), the test can ensure that those routes now have a higher preference or lower metric on this link. Essentially, this step uses pyATS to programmatically assert that your **business policies** (which might be encoded in route-maps or prefix-lists on the devices) are being fulfilled by observing the outcome in the device's operational data. Any deviation – e.g., a route that should have been filtered is found, or a community is missing – would cause the test to fail, prompting further investigation.

7. **Reporting and Integration:** After running all the above checks, pyATS will produce a **report** of results. Each sub-test (latency, reachability, BGP neighbor, route table, policy) will be marked **PASS** or **FAIL**. PyATS has built-in reporting that can output to the console or generate HTML/XML reports summarizing the test outcomes [9]. In a CI/CD pipeline, these tests could be triggered automatically right after the network change (deployment of new links) is done. For example, using Jenkins or GitLab CI, you could kick off the pyATS job as a validation stage. The results could even be integrated with chatops – notifying the team on Slack/Teams if the deployment validation passed or if there are issues. In our fintech environment scenario, this means right after the network team enables the new links, they can run the pyATS test suite and within minutes get a comprehensive confirmation that **latency is within limits, all links are reachable, BGP adjacencies are established, routes are correct, and policies are enforced**. This provides high confidence that the new links are functioning properly and the network is ready for live financial traffic. If any test fails (say one BGP session didn't come up due to a config error, or latency is higher than expected indicating a potential circuit issue), the team can immediately take corrective action *before* traders or applications notice any problem.

In summary, **pyATS automates network validation** tasks that would otherwise be manual and error-prone. It is particularly valuable in scenarios like the above where multiple new links are introduced in a critical environment. By programmatically checking connectivity, protocol states, routing entries, and compliance with policies, pyATS ensures the network change achieves the intended outcomes. This not only saves time but also reduces risk in network operations. As noted in real-world use cases, incorporating such automated testing can drastically de-risk configuration changes – for example, engineers have used pyATS to validate BGP routing and reachability (ping/traceroute) as part of large network upgrades [16]. In a fintech context, where downtime or latency issues directly translate to financial loss, having this level of automated validation is invaluable. PyATS gives network engineers a way to deliver consistent, repeatable tests and quickly generate evidence that the network is healthy after each change. This way, deploying multiple new links becomes a more **scientific and reliable process**, with each step validated by code. The end result is a network that meets performance expectations (low latency, high availability) and a team that can confidently deploy changes knowing that any deviation from expected behavior will be caught immediately by the automated tests.

Sources: The capabilities and examples above are drawn from Cisco's documentation and industry use cases of pyATS. PyATS with Genie enables parsing device outputs into structured data, supporting hundreds of show commands across platforms [1]. It has been used to validate features like BGP neighbor uptime and route presence in routing tables via automated testcases [7]. PyATS makes it easy to verify network state **before and after changes**, providing reports on which checks passed or failed [9]. For connectivity testing, the pyATS ping parser is an example of how even performance metrics (min/max/avg latency) can be automatically checked against requirements [14]. By automating such validations, pyATS helps ensure that new deployments (like our multi-link rollout) meet the intended design and service levels [13] [4].

1 3 5 6 7 8 9 10 15 Network Validation with pyATS - BlueAlly
https://www.blueally.com/network-validation-with-pyats/

2 13 Introduction to Cisco pyATS: Revolutionizing Network Testing | Orhan Ergun
https://orhanergun.net/introduction-to-cisco-pyats-revolutionizing-network-testing

4 pyATS Demos Introduction and Setup - NetDevOps Series, Part 12
https://blogs.cisco.com/developer/pyats-demo-netdevops-12

11 12 16 Cisco pyATS — Network Test and Automation Solution: Data-driven and... | Daniel Wade
https://www.linkedin.com/posts/danielcwade_cisco-pyats-network-test-and-automation-activity-7265488393579446272-dYGO

14 automateyournetwork/pyats_ping_tests - Cisco Code Exchange
https://developer.cisco.com/codeexchange/github/repo/automateyournetwork/pyats_ping_tests/