

Cisco pyATS Network Test and Automation Solution

Data-driven and reusable testing for modern networks



ciscopress.com

JOHN CAPOBIANCO DAN WADE



Cisco pyATS—Network Test and Automation Solution

Data-driven and reusable testing for modern networks

John Capobianco Dan Wade

Cisco Press

221 River Street Hoboken, NJ 07030 USA

Cisco pyATS—Network Test and Automation Solution

John Capobianco, Dan Wade

Copyright© 2025 Cisco Systems, Inc.

Cisco Press logo is a trademark of Cisco Systems, Inc.

Published by: Cisco Press

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms, and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/permissions.

No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

Microsoft and/or its respective suppliers make no representations about the suitability of the information contained in the documents and related graphics published as part of the services for any purpose all such documents and related graphics are provided "as is" without warranty of any kind. Microsoft and/or its respective suppliers hereby disclaim all warranties and conditions with regard to this information, including all warranties and conditions of merchantability, whether express, implied or statutory, fitness for a particular purpose, title and non-infringement. In no event shall Microsoft and/or its respective suppliers be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of information available from the services.

The documents and related graphics contained herein could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein. Microsoft and/or its respective suppliers may make improvements and/or changes in the product(s) and/or the program(s) described herein at any time. Partial screen shots may be viewed in full within the software version specified.

Microsoft[®] Windows[®], and Microsoft Office[®] are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.

\$PrintCode

Library of Congress Control Number: 2024907000

ISBN-13: 978-0-13-803167-1

ISBN-10: 0-13-803167-3

Warning and Disclaimer

This book is designed to provide information about all aspects of Cisco pyATS. Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied.

The information is provided on an "as is" basis. The authors, Cisco Press, and Cisco Systems, Inc. shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the discs or programs that may accompany it.

The opinions expressed in this book belong to the authors and are not necessarily those of Cisco Systems, Inc.

Trademark Acknowledgments

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Cisco Press or Cisco Systems, Inc., cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Feedback Information

At Cisco Press, our goal is to create in-depth technical books of the highest quality and value. Each book is crafted with care and precision, undergoing rigorous development that involves the unique expertise of members from the professional technical community.

Readers' feedback is a natural continuation of this process. If you have any comments regarding how we could improve the quality of this book, or otherwise alter it to better suit your needs, you can contact us through email at feedback@ciscopress.com. Please make sure to include the book title and ISBN in your message.

We greatly appreciate your assistance.

Please contact us with concerns about any potential bias at https://www.pearson.com/report-bias.html.

GM K12, Early Career and Professional	Copy Editor: Bart Reed	
Learning: Soo Kang	Technical Editors: Stuart Clark, Charles Greenaway	
Alliances Manager, Cisco Press: Caroline Antonio	Editorial Assistant: Cindy Teeters	
Director, ITP Product Management: Brett Bartow	Designer: Chuti Prasertsith	
Executive Editor: Nancy Davis	Composition: codeMantra	
Managing Editor: Sandra Schroeder	Indexer: Timothy Wright	
Development Editor: Christopher Cleveland	Droofroodon Dorbara Maal	
Senior Project Editor: Mandie Frank	PIOOITeauer: Darbara Mack	

،،|،،،|،، دisco،

Americas Headquarters Cisco Systems, Inc. San Jose, CA Asia Pacific Headquarters Cisco Systems (USA) Pte. Ltd. Singapore Europe Headquarters Cisco Systems International BV Amsterdam, The Netherlands

Cisco has more than 200 offices worldwide. Addresses, phone numbers, and fax numbers are listed on the Cisco Website at www.cisco.com/go/offices.

Cisco and the Cisco logo are trademarks or registered trademarks of Cisco and/or its affiliates in the U.S. and other countries. To view a list of Cisco trademarks, of the property of their respective owners. The use of the word partner on timply a partnership relationship between Cisco and any other company. (1110R)

About the Authors

John Capobianco has a dynamic and multifaceted career in IT and networking, marked by significant contributions to both the public and private sectors. Beginning his journey in the field as an aluminum factory worker, Capobianco's resilience and dedication propelled him through college, earning a diploma as a Computer Programmer Analyst from St. Lawrence College. This initial phase set the foundation for a career underpinned by continuous learning and achievement, evident from his array of certifications, including multiple Cisco certifications as well as Microsoft certification.

Transitioning from his early educational accomplishments, Capobianco's professional life has spanned over two decades, featuring roles that showcased his technical prowess and strategic vision. His work has significantly impacted both the public and private sectors, including notable positions at the Parliament of Canada, where he served as a Senior IT Planner and Integrator, and at Cisco, where he began as a Developer Advocate. These roles have been instrumental in shaping his perspective on network management and security, leading to his recent advancement into a Technical Leader role in Artificial Intelligence for Cisco Secure, reflecting his commitment to integrating AI technologies for enhancing network security solutions.

In addition to his professional and technical achievements, Capobianco is also an accomplished author. His book *Automate Your Network: Introducing the Modern Approach to Enterprise Network Management*, published in March 2019, encapsulates his philosophy toward leveraging automation for efficient and effective network management. He is dedicated to lifelong learning and professional development, supported by a solid foundation in education and a broad spectrum of certifications, and now aims to share his knowledge with others through this book, YouTube videos, and blogs. John can be found on X using @john_capobianco.

Dan Wade is a Network and Infrastructure Automation Practice Lead at BlueAlly. As part of the Solutions Strategy team at BlueAlly, he is responsible for developing network and infrastructure automation solutions and enabling the sales and consulting teams on delivery of the developed solutions. Solutions may include infrastructure provisioning, configuration management, network source of truth, network observability, and, of course, automated testing and validation. Previous to this role, Dan worked as a consulting engineer with a focus on network automation.

Dan has more than seven years of experience in network automation, having worked with automation tooling and frameworks such as Ansible and Terraform, and Python libraries, including Nornir, Netmiko, NAPALM, Scrapli, and Python SDKs. Dan has been working with pyATS and the pyATS library (Genie) for the past four to five years, which has inspired him to embrace automated network testing. In 2021, Dan contributed to the genieparser library with a new IOS XE parser. He also enjoys creating his own open-source projects focused on network automation. Dan holds two professional-level certifications from Cisco: Cisco DevNet Professional and CCNP Enterprise. He is also a member of the Cisco DevNet 500 and Cisco Champions program.

Dan enjoys sharing knowledge and experience on social media with blog posts and YouTube videos as well as participating in podcast episodes. He's passionate about helping others explore network automation and advocating how network automation can empower, not replace, network engineers. You can find him on social media @devnetdan.

About the Technical Reviewers

Stuart Clark is a senior developer advocate, public speaker, author, and DevNet Expert No. 2022005. Stuart is a sought-after speaker, frequently gracing the stages of industry conferences worldwide, presenting on his expertise in programmability and DevOps methodologies. Passionate about fostering knowledge sharing, he actively creates community content and leads developer communities, empowering others to thrive in the ever-evolving tech landscape. In his previous role as a network engineer, he became obsessed with network automation and became a developer advocate for network automation. He contributed to the Cisco DevNet exams and was part of one of the SME teams that created, designed, and built the Cisco Certified DevNet Expert. He lives in Lincoln, England, with his wife, Natalie, and their son, Maddox. He plays guitar and rocks an impressive two-foot beard while drinking coffee. You can find him on social media @bigevilbeard.

Charles Greenaway, CCIE No. 11226 (R&S, Security, Datacenter), is a field CTO for BT (https://www.bt.com/about/bt). With more than 20 years of data networking experience across LAN/WAN/DC in multiple industry sectors across the globe, he ensures that his customers' use of technology is aligned with their business goals while developing and implementing the technology strategy.

His current focus is helping customers transition toward Global Fabric technologies that provide software-defined underlay and overlay networking to underpin secure multicloud connectivity.

As a member of the DevNet 500 and the Cisco Champions program, Charles promotes the use of programmability and automation to make it accessible to engineers at all levels of skill and experience. He has developed technical content through Greencodemedia Limited and in the public domain at events such as Cisco DevNet Create. Charles is a graduate of Loughborough University, holds a BSc in Computer Science, and lives in the United Kingdom.

Dedications

John: This book is dedicated to my wife and partner of more than 25 years, Michelle. Without her support and encouragement, I would likely still be driving a forklift. J'taime le.

Dan: I would like to dedicate this book to my wonderful wife Hailey and my two amazing kids. They are my foundation and have been patient during the entire writing process. I'd also like to dedicate this book to my parents, who have continued to push me to accomplish whatever I wanted in life. I love you all!

Acknowledgments

John: I would first like to acknowledge what a pivotal role St. Lawrence College has played in my life—first as a student, then as a professor. Thank you to Donna Graves, Janis Michael, and, of course, rest in peace, Carl Davis. To everyone I've ever worked with in my career from Empire Life to the Parliament of Canada, thank you. I am very proud of what we accomplished together and for the confidence you had in me to build, support, evolve, and, ultimately, automate your networks. Thanks also to Cisco for embracing me completely as one of your own—I've never had such a supportive culture.

Thanks as well to everyone involved with the publishing of this book, from Nancy Davis and Chris Cleveland and the Pearson team, to our editors Stuart Clark and Charles Greenaway for their dedication to the project, and last but not least, to Dan Wade for co-authoring the book. From joint live streams to collaborating on this book, I am really proud to call you a friend.

Finally, to JB and Siming, for inviting me to a private pyATS crash course. You have both been so giving and have provided me real guidance and direction and turned me onto Python. Thank you both.

Dan: First, I'd like to thank the Art of Network Engineering (AONE) community, specifically AJ Murray, for encouraging me to begin blogging and creating my own brand. I can confidently say there would be no DevNet Dan without AONE! I'd also like to thank NetCraftsmen for taking a chance on me in the beginning of my consulting career. It was my first time working in the consulting space, and they've consistently guided me to success. Thank you Terry, Shaffeel, Robert, Bill, Joel, and John for continuing to encourage me and push me to grow professionally.

I would like to thank Nancy Davis for giving me the opportunity to pursue this project. She continues to encourage and support me to pursue creative opportunities. A big thank you to Chris Cleveland, development editor, for providing the best support developing the book, and to Stuart and Charles for their unbiased and honest technical review of the book and its contents. I'd also like to thank my wonderful co-author John. It has been a pleasure working and writing this phenomenal book with him!

Finally, thanks to all the content creators, trainers, and authors who have influenced my narration style and contributed to my constant learning of networking and software development.

Contents at a Glance

Introduction xxx

- Chapter 1 Foundations of NetDevOps 1 Chapter 2 Installing and Upgrading pyATS 37 Testbeds 49 Chapter 3 AEtest Test Infrastructure 73 Chapter 4 pyATS Parsers 137 Chapter 5 Chapter 6 Test-Driven Development 169 Chapter 7 Automated Network Documentation 189 Chapter 8 Automated Network Testing 233 Chapter 9 pyATS Triggers and Verifications 279 Chapter 10 Automated Configuration Management 303 Chapter 11 Network Snapshots 321 Chapter 12 Recordings, Playbacks, and Mock Devices 337 Chapter 13 Working with Application Programming Interfaces (API) 347 Chapter 14 Parallel Call (pcall) 397
- Chapter 15 pyATS Clean 411
- Chapter 16 pyATS Blitz 459
- Chapter 17 Chatbots with Webex 485
- Chapter 18 Running pyATS as a Container 503
- Chapter 19 pyATS Health Check 515
- Chapter 20 XPRESSO 527
- Chapter 21 CI/CD with pyATS 557
- Chapter 22 Robot Framework 575
- Chapter 23 Leveraging Artificial Intelligence in pyATS 591
- Appendix A Writing Your Own Parser 633
- Appendix B Secret Strings 651

Index 657

Reader Services

Register your copy at www.ciscopress.com/title/ISBN for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to www.ciscopress.com/register and log in or create an account*. Enter the product ISBN 9780138031671 and click Submit. When the process is complete, you will find any available bonus content under Registered Products.

*Be sure to check the box that you would like to hear from us to receive exclusive discounts on future editions of this product.

Contents

Introduction xxx

Chapter 1	Foundations of NetDevOps 1
	Traditional Network Operations 2
	Architecture 2
	High-Level Design 3
	Low-Level Design 3
	Day-1 4
	Offline Initial Configuration 5
	Software Images 5
	Day 0 5
	Layer 1 6
	Initial Configuration 6
	Initial Testing and Validation 7
	Day 1 7
	Incremental Configuration 8
	Provisioning New Endpoints 9
	Provisioning New Services 9
	Day N 9
	Monitoring (and Now Testing) 9
	Responding to Events 10
	Upgrading 10
	Decommissioning 11
	Software Development Methodologies 11
	Waterfall 11
	Lean 12
	Agile 12
	DevOps 13
	Expanding into Networks 13
	Infrastructure as Code 14
	Test-Driven Development 14
	NetDevOps 14
	Plan 16

Code 16 Build 16 Test 16 Release 16 Deploy 17 Operate 17 Monitor 18 Additional Benefits of NetDevOps 18 Single Source of Truth 18 Intent-Based Configuration 18 Version and Source Control 19 GitOps 19 Efficiency 19 Speed 20 Agility 21 Quality 21 Comparing Network Automation Tools 21 The Modern Network Engineer Toolkit 22 Integrated Development Environment 22 "Old School" 22 "New School" 23 Git 23 GitHub 24 GitLab 24 Structured Data 24 JavaScript Object Notation (JSON) 24 eXtensible Markup Language (XML) 25 YAML Ain't Markup Language (YAML) 26 YANG 26 Application Programing Interface (API) 27 Representational State Transfer (REST) 27 GraphQL 28 cURL 28 Postman 29 Python 29

pip 30 Software Development Kits 31 Virtual Environment 31 Virtual Machines 31 Containers 31 Kubernetes 32 CI/CD 32 Jenkins 33 GitLab CI/CD 33 GitHub Actions 34 Drone 34 Summary 35 References 36 Chapter 2 Installing and Upgrading pyATS 37 Installing pyATS 37 Setting Up a Python Virtual Environment 38 Installing pyATS Packages 38 Upgrading pyATS 42 Troubleshooting pyATS 45 Summary 47 Chapter 3 Testbeds 49 What Is YAML? 49 What Is a Testbed? 51 Building a Simple Testbed 53 Edge Cases 54 External Sources of Truth 56 Device Connection Abstractions 60 Testbed Validation 63 Dynamic Testbeds 66 Intent-based Networking with Extended Testbeds 68 Summary 70 Chapter 4 AEtest Test Infrastructure 73 Getting Started with AEtest 74 Installation 74 Design Features 74

Core Concepts 75 Testscript Structure 75 Common Setup 76 Subsection 76 Testcases 77 Setup Section 77 Test Section 77 Cleanup Section 77 Common Cleanup 78 Section Steps 79 AEtest Object Model 82 TestScript Class 82 Container Classes 82 Function Classes 83 Runtime Behavior 85 Self 86 Parent 87 Section Ordering 88 Test Results 88 Result Objects 88 Result Behavior 89 Interaction Results 90 Result Rollup 90 Processors 91 Processor Types 91 Processor Definition and Arguments 92 Context Processors 93 Global Processors 93 Processor Results 94 Data-Driven Testing 94 Test Parameters 95 Parameter Relationships 95 Parameter Properties 96 Parameter Types 96 Parameter Parametrization 98

Reserved Parameters 98 Datafile Input 98 Looping Sections 104 Defining Loops 104 Loop Parameters 104 Dynamic Looping 107 Running Testscripts 108 Testing Arguments 108 Standard Arguments 108 Argument Propagation 109 Execution Environments 109 Standalone Execution 109 Easypy Execution 114 Testable 117 Testscript Flow Control 117 Skip Conditions 117 Running Specific Testcases 119 Testcase Grouping 120 Must-Pass Testcases 121 Testcase Randomization 123 Maximum Failures 124 Custom Testcase Discovery 125 Reporting 126 Standalone Reporter 126 AEtest Reporter 127 Debugging 133 Summary 134

Chapter 5 pyATS Parsers 137

Vendor-Agnostic Automation 138 pyATS learn 139 pyATS Parsers 146 Parsing at the CLI 148 Parsing with Python 159 Dictionary Query 160 Differentials 162 Summary 167

Test-Driven Development 169 Chapter 6 Introduction to Test-Driven Development 170 Applying Test-Driven Development to Network Automation 172 Introduction to pyATS 174 The pyATS Framework 175 AEtest 176 Easypy 179 Testbed and Topology 180 Testbed and Device Cleaning with Kleenex 181 Asynchronous Library (Parallel Call) 182 Data Structures 182 TCL Integration 183 Logging 184 Result Objects 185 Reporter 185 Utilities 186 Robot Framework Support 186 Manifest 187 Summary 187 Endnotes 188 Chapter 7 Automated Network Documentation 189 Introduction to pyATS Jobs 190 Running pyATS Jobs from the CLI 196 pyATS Job CLI Logs 199 pyATS Logs HTML Viewer 203 Jinja2 Templating 205 Business-Ready Documents 206 **JSON 207** YAML 207 Comma-Separated Values 208 Markdown: Tables 210 Markdown: Markmap Mind Maps 212 Markdown: Mermaid Flowcharts 215 Markdown: Mermaid Class Diagrams 217 Markdown: Mermaid State Diagrams 219 Markdown: Mermaid Entity Relationship Diagrams 221

	Markdown: Mermaid Mind Maps 223 HTML 225 Datatables 227 Summary 232 References 232	
Chapter 8	 Automated Network Testing 233 An Approach to Network Testing 234 Software Version Testing 235 Interface Testing 243 Communicating with Devices: The Role of SSH in Testing Using RESTCONF 252 Neighbor Testing 259 Reachability Testing 262 Intent Validation Testing 267 Feature Testing 271 Summary 276 	244
Chapter 9	pyATS Triggers and Verifications 279 Genie Objects 279 Genie Ops 280 Genie Conf 281 Genie Harness 282 gRun 282 Datafiles 283 Device Configuration 284 PTS (Profile the System) 285 PTS Golden Config 286 Verifications 286 Verification Types 287 Verification Datafile 288 Writing a Verification 288 Triggers 290 Trigger Datafile 291 Trigger Cluster 292 Writing a Trigger 293 Trigger and Verification Example 296 Genie Harness (gRun) 297	

pyATS 299 Summary 301

Chapter 10 Automated Configuration Management 303 Intent-Based Network Configuration 303 Generating Configurations with pyATS 304 Data Modeling and Validation 304 Data Templates 305 Genie Conf Objects 308 Configuring Devices with pyATS 309 File Transfer 309 FileUtils Module 309 Embedded pyATS File Transfer Server 311 pyATS Library API 313 Testbed-Wide Configuration 314 Device Configuration 315 Jinja2 Configuration 317 Genie Harness 317 Config Datafile 317 Config Check 319 Summary 320 Chapter 11 Network Snapshots 321

Network Profiling 321 What Is a Network Snapshot? 322 Network Features 322 Comparing Network State 324 Pre-change Snapshots 324 Post-change Snapshots 326 Snapshot Differentials 327 Genie Diff 327 Data Exclusions 331 Default Exclusions 331 Additional Exclusions 332 Custom Exclusions 332 Polling Expected State 332 Robot Framework with Genie 333 Summary 335 References 335

Chapter 12	Recordings, Playbacks, and Mock Devices 337
	Recording pyATS jobs 337
	Playback Recordings 341
	Mock Devices 342
	Use Cases for Mock Devices 342
	Additional Use Cases 343
	Mock Device CLI 344
	Summary 345
Chapter 13	Working with Application Programming Interfaces (API) 347
	pyATS APIs 347
	REST Connector 353
	NXOS 355
	GET API 355
	POST API 356
	DELETE API 357
	PATCH API 358
	PUT API 358
	NSO 359
	Connect API 360
	GET API 361
	POST API 361
	PATCH API 362
	PUT API 362
	DELETE API 363
	Catalyst Center 363
	Connect API 364
	GET API 365
	Disconnect API 365
	IOS XE 366
	Connect API 366
	GET API 367
	POST API 367
	PATCH API 368
	PUT API 368

DELETE API 368 Cisco ACI APIC 369 GET API 370 POST API 370 DELETE API 371 Query 371 Config 372 LookupByDn 372 LookupByClass 372 Exists 372 Get_Model 373 Create 373 Config_And_Commit 373 BIG-IP 373 Connect API 374 Disconnect API 375 GET API 375 POST API 375 PATCH API 376 PUT API 376 DELETE API 376 SD-WAN vManage 377 Connect API 378 GET API 379 POSTAPI 379 PUT API 379 DELETE API 380 DCNM 380 GET API 381 POST API 381 DELETE API 382 PATCH API 382 PUT API 383 Nexus Dashboard 383 GET Command 383

POST Command 384 PUT Command 384 DELETE Command 385 YANG Connector 385 Topology YAML Configuration 386 Connect Function 387 Connected Property 388 Get Function 388 Get Config Function 388 Edit Config Function 388 Request Function 388 Get Schema Function 389 Disconnect and Close Session Functions 389 gNMI 389 Setting Up and Connecting with gNMI Client 389 Fetching Device Capabilities 390 gNMI SetRequest 391 gNMI GetRequest 391 Creating an Instance of Config 392 Creating Config Objects 392 Comparing Configs 393 Creating ConfigDelta Objects with Special Requirements 394 XPath Queries 394 Facilitating XPath Queries with ns help() 395 Important Note on RPCReply and XPath 395 Summary 395

Chapter 14 Parallel Call (pcall) 397

Scaling Performance 397 Asynchronous Programming 398 Threading 398 Multiprocessing 399 *Easypy 400 Logging and Reporting 400* Parallel Call (pcall) 400 Usage 401

Targets 402 Single Target 402 Multiple Targets 403 Error Handling 404 Logging 404 Pcall Object 405 Performance Comparison 406 Summary 409 Chapter 15 pyATS Clean 411 Getting Started 411 Device Cleaning 412 Supported Devices 413 Network Platforms 413 Power Cyclers 413 Clean YAML 423 Devices 424 Clean Stages 424 Device Groups 428 Useful Features 431 Software Image Management 431 Device Recovery 434 Clean Execution 436 Clean Validation 436 Execution Methods 437 Integrated 437 Standalone 437 Clean Logging 438 pyATS Testscript Usage 438 Developing Clean Stages 443 Getting Started 443 Stage Template 443 New Stage 444 Abstracting an Existing Stage 444 Schema and Arguments 445 Clean Stage Steps 447

Execution Order 450 Documentation 452 Abstracted Clean Stages 455 Summary 456

Chapter 16 pyATS Blitz 459

Blitz YAML 459 Actions 460 Action Outputs 461 Filters 461 Variables 463 Saving Outputs 465 Verifying Action Outputs 467 Advanced Actions 470 Parallel 470 Loop 471 Run Condition 475 Blitz Features 476 Negative Testing 477 Script Termination on Failure 477 Prompt Handling 478 Results 478 Timeouts 479 Customizing Log Messages 480 Blitz Usage 481 Blitz Development 481 Custom Blitz Actions 481 Custom Blitz Sections 482 Useful Tips 483 Summary 484 Chapter 17 Chatbots with Webex 485 Integrating pyATS with Webex 486 pyATS Job Integration 487 pyATS Health Check Integration 489 Adaptive Cards 490

Customized Job Notifications 492

Summary 502

Running pyATS as a Container 503 Chapter 18 Introduction to Containers 504 pyATS Official Docker Container 506 pyATS Image Builder 507 Building a pyATS Image from Scratch 510 Summary 513 Chapter 19 pyATS Health Check 515 Health Checks 515 CPU and Memory 515 Logging 516 Core File 516 Custom Health Checks 517 Health YAML File 517 The processor Key 520 The reconnect Key 520 Testcase/Section Selection 520 Health Check Results 522 Health Check Usage 523 Health Check YAML Validation 523 PyATS Job Integration 523 Health Check CLI Arguments 524 Summary 525 Chapter 20 XPRESSO 527 Installing XPRESSO 529 Common Issues, Questions and Answers 534 Unhealthy Services 535 References to S3 535 Error: No Resources Found 535 Cannot Log In Using the Default admin/admin 535 ElasticSearch Failed to Start 535 Cannot Connect to Database 535 General Networking Issues with the XPRESSO Installation 535 Getting Started with XPRESSO 536 Facilitating Quick Adoption 536 Transforming a pyATS Job into XPRESSO 538 Summary 556

Chapter 21 CI/CD with pyATS 557 What Is CI/CD? 557 Demystifying CI/CD 558 CI/CD Pipeline Integration 558 GitHub Actions 558 GitLab CI/CD 559 CI/CD In NetDevOps 560 NetDevOps Scenario 562 Lab Setup 562 CI/CD Stages 563 NetDevOps in Action 565 Configuration Changes 565 Testing Network Changes 568 Feedback Loop 571 What's Next? 572 Summary 573 Chapter 22 Robot Framework 575 What Is the Robot Framework? 575 Getting Started with Robot Framework 576 Test Cases 576 Keywords 577 Variables 578 Importing Libraries 580 Test Execution 580 Test Results and Reporting 581 Robot Integration with pyATS 582 PyATS Keywords 582 Unicon Keywords 584 Genie Keywords 585 Easypy Integration 588 Summary 590 Chapter 23 Leveraging Artificial Intelligence in pyATS 591 OpenAI API 597 Retrieval Augmented Generation with LangChain 612 Rapid Prototyping with Streamlit 621 Summary 631

Appendix A Writing Your Own Parser 633

Contributing to the pyATS Library 633 Parser Structure 635 Schema Class 635 Parser Class 637 Creating Your Parser 638 Development Environment 638 Writing Your Schema Class 640 Writing Your Parser Class 642 Testing Your Parser 644 Contributing to Genieparser 647 References and Recommended Readings 650

Appendix B Secret Strings 651

How to Secure Your Secret Strings 651 Multiple Representers 653 Representer Classes 655

Index 657

Command Syntax Conventions

The conventions used to present command syntax in this book are the same conventions used in the IOS Command Reference. The Command Reference describes these conventions as follows:

- Boldface indicates commands and keywords that are entered literally as shown. In actual configuration examples and output (not general command syntax), boldface indicates commands that are manually input by the user (such as a show command).
- *Italic* indicates arguments for which you supply actual values.
- Vertical bars (I) separate alternative, mutually exclusive elements.
- Square brackets ([]) indicate an optional element.
- Braces ({ }) indicate a required choice.
- Braces within brackets ([{ }]) indicate a required choice within an optional element.

Foreword

In late 2013, I found myself seated at the end of a restaurant table in San Jose, celebrating the success of our latest Tcl language–based test automation feature release. Tibor Fabry-Asztalos, our visionary senior director, raised his glass in a toast: "We need to look at our next goal. It's time to transition to Python-based automation." With that, he gazed to his left, where coincidentally his chain of reports was seated in order—and each person looked further to their left, until there was just me, the final link in the chain, entrusted to shoulder that responsibility.

And so pyATS was born.

After two decades of Tcl/Expect-based automation and testing in Cisco, the call for a more modern, natively object-orienting infrastructure was undeniable—one that could scale forward, lower the barrier for adoption, and attract new talents as Tcl expertise waned.

2024 marks the 10-year anniversary for pyATS. Originally introduced as an internal testing solution, its 2017 public launch through Cisco DevNet marked a definitive, transformative moment. It enabled closer collaboration between Cisco engineering, customers, and their network engineers, unlocking a plethora of opportunities and use cases. Around that time, NetDevOps was in its infancy, and network engineers were seeking for their next career breakthrough. pyATS was ready just around the corner.

Rarely does one find themselves at the helm of opportunity to shape the next decade of network automation, a chance to redefine the landscape of network testing, and influence the careers of countless network engineers. It has been an exciting journey, filled with dedication, perseverance, and innovation. Most importantly, we took pride in what we have created and accomplished.

Looking back, could we have done better? Absolutely. Along the way, mistakes were made, and compromises became necessary. But as someone special to me once said, "every decision you make in life [sic] is always the best that you could, based on the limited knowledge you had at that time." We, the pyATS development team, gave it our best, and the community echoed positively.

It has been a privilege and an honor to be able to stand at the precipice of a new chapter in the history of test automation at Cisco. A heartfelt thank-you goes to our team, our community members, and everyone who supported us along the way. As pyATS continues to evolve, my sincerest wishes for its continued momentum and enduring legacy.

>>> from pyats import awesome

—Siming Yuan, pyATS Founder, Architect and Lead Developer

Reflecting on the inception of pyATS, it's astounding to see the journey from an ambitious project within Cisco engineering to a cornerstone of network automation. Born from the challenges we faced daily, it quickly grew beyond the initial scope, demonstrating the power of innovative solutions in a rapidly evolving field. I am filled with gratitude for the brilliant minds I worked with and the community that has grown around these tools. Your enthusiasm and support have been the driving force behind its success.

Now, looking back, I see the legacy of pyATS not just in the technical achievements, but in the community and collaboration it fostered. It has been a privilege to contribute to this chapter of network engineering, and I am proud of what we accomplished together.

A special thanks to all the pyATS team members who have worked on it. It wouldn't have been possible without you. Thank you to everyone who has joined us on this remarkable journey. Your contributions have made all the difference.

-Jean-Benoit Aubin, Lead Developer and Architect, pyATS

Introduction

This book was written to explore the powerful capabilities of automated network testing with the Cisco pyATS framework. Network testing and validation is a low-risk, yet powerful domain in the network automation space. This book is organized to address the multiple features of pyATS and the pyATS library (Genie). Readers will learn why network testing and validation are important, how pyATS can be leveraged to run tests against network devices, and how to integrate pyATS into larger workflows using CI/CD pipelines and artificial intelligence (AI).

Goals and Objectives

This book touches on many aspects of network automation, including device configuration, data parsing, APIs, parallel programming, artificial intelligence, and, of course, automated network testing. The intended audience for this book is network professionals and software developers wanting to learn more about the pyATS framework and the benefits of automated network testing. The audience should be comfortable with Python, as pyATS is built with the Python programming language.

Candidates who are looking to learn pyATS as it relates to the Cisco DevNet Expert Lab exam will find the use cases and examples throughout the book valuable for exam preparation.

How This Book Is Organized

Chapter 1, "Foundations of NetDevOps": This chapter introduces NetDevOps, outlining its benefits and how it merges with software development methodologies to enhance network automation. We compare key automation tools and detail the modern network engineer's toolkit, setting the stage for applying NetDevOps in practice.

Chapter 2, **"Installing and Upgrading pyATS"**: The chapter shows how to install and upgrade pyATS and the pyATS library using Python package management tools and built-in pyATS commands.

Chapter 3, "Testbeds": This chapter covers YAML's basics, explores the concept of a testbed, and examines device connection abstractions. We discuss methods for testbed validation, the creation of dynamic testbeds, and how intent-based networking integrates with extended testbeds, providing a roadmap for their practical application.

Chapter 4, "**AEtest Test Infrastructure**": This chapter is one of the key chapters in this book. It goes in depth and reviews the different components that make up AEtest, the testing infrastructure that is the core of pyATS. Everything from defining testcases and individual test sections to running testscripts is covered in this chapter. After reading this chapter, you'll understand how to introduce test inputs and parameters, define test sections, control the flow of test execution, and review test results with the built-in reporting features.

Chapter 5, "**pyATS Parsers**": This chapter delves into pyATS parsers, emphasizing vendor-neutral automation strategies. It covers the essentials of pyATS learn and parse features, techniques for CLI parsing, and parsing with Python. Additionally, we explore how to perform dictionary queries and analyze differentials, equipping you with the necessary skills for effective network data handling.

Chapter 6, "Test-Driven Development": This chapter introduces test-driven development (TDD), its application in network automation, and an overview of pyATS. It further explores the pyATS framework, setting the foundation for incorporating TDD practices into network management.

Chapter 7, **"Automated Network Documentation":** This chapter explores automated network documentation, beginning with an introduction to pyATS jobs. It details executing pyATS jobs from the command-line interface (CLI), interpreting CLI logs, and utilizing the pyATS logs HTML viewer for enhanced analysis. We also delve into Jinja2 templating for document creation, culminating in the generation of business-ready documents.

Chapter 8, "**Automated Network Testing**": This pivotal chapter delves into automated network testing, the core focus of the book. It outlines a strategic approach to network testing, including software version testing, interface testing, neighbor testing, and reachability testing. Additionally, we explore intent-validation testing and feature testing, essential components for ensuring network reliability and performance.

Chapter 9, "pyATS Triggers and Verifications": This chapter reviews how to use triggers and verifications using the Genie Harness. Triggers and verifications allow you to build dynamic testcases, with a low-code approach, that can change with your network requirements.

Chapter 10, "Automated Configuration Management": In this chapter we will look at how to generate intent-based configuration using data models, Jinja2 templates, and Genie Conf objects. In addition to generating configurations, we will see how to push configuration to network devices using a file transfer server, Genie Conf objects, and pyATS device APIs.

Chapter 11, "Network Snapshots": This chapter looks at how to profile the network by creating and comparing snapshots of the network. Network snapshots can be helpful when you're troubleshooting a network issue or just learning about the network's operating state at a point in time.

Chapter 12, "Recordings, Playbacks, and Mock Devices": This chapter introduces pyATS recordings, covering the recording of pyATS jobs and the playback of these recordings. It explains how to create mock devices and simulate device interactions through the mock device CLI, offering practical insights into testing without the need for live network equipment.

Chapter 13, "Working with Application Programming Interfaces (API)": This chapter focuses on working with pyATS APIs, detailing the pyATS API framework, REST connector, YANG connector, and gNMI. It provides insights into how these tools and protocols can be utilized for efficient network automation and management through API interactions.

Chapter 14, "Parallel Call (pcall)": Testing in pyATS can be sped up using parallel processing (parallelism). In this chapter, we review the differences between parallelism and concurrency using asynchronous programming. Parallel call (pcall) in pyATS enables parallel execution and is built on the multiprocessing package in the Python standard library.

Chapter 15, "pyATS Clean": In this chapter, you will see how pyATS can reset devices during or after testing using the pyATS Clean feature.

Chapter 16, "pyATS Blitz": In this chapter, we will review pyATS Blitz, which creates a low-code approach to building pyATS testcases using YAML syntax.

Chapter 17, "Chatbots with Webex": This chapter explores integrating pyATS with Webex, including pyATS job and health check integrations. It delves into using Adaptive Cards within Webex for interactive content and outlines methods for setting up customized job notifications, enhancing communication and monitoring in network operations.

Chapter 18, "Running pyATS as a Container": This chapter introduces the concept of containers, focusing on the pyATS official Docker container. It guides you through the pyATS image builder and details the process of building a pyATS image from scratch, offering a comprehensive approach to deploying pyATS as a containerized application.

Chapter 19, "pyATS Health Check": This chapter dives into the different health checks that run to ensure devices under testing are operating correctly. Built-in health checks include checking CPU, memory, logging, and the presence of core dump files to ensure devices haven't malfunctioned or crashed during testing.

Chapter 20, "XPRESSO": This section covers pyATS XPRESSO, starting with installation instructions. It provides a beginner's guide to getting started with XPRESSO and details on running pyATS jobs within the XPRESSO environment, facilitating an easy entry into utilizing this powerful tool.

Chapter 21, "CI/CD with pyATS": The concept of CI/CD is a common practice in software development to build and test code before it's pushed to production. In this chapter, we see how to use multiple network automation tools, including GitLab, Ansible, and pyATS, to apply CI/CD practices when pushing configuration changes to the network.

Chapter 22, "Robot Framework": In this chapter, we review Robot Framework, an opensource test automation framework. Robot Framework allows you to use English-like keywords to define testcases. After we review Robot Framework, we see how the pyATS libraries—Unicon, pyATS, and the pyATS library (Genie)—are integrated into Robot Framework by providing test libraries that include keywords to interact with network devices and define testcases.

Chapter 23, "Leveraging Artificial Intelligence in pyATS": This chapter explores the integration of pyATS with artificial intelligence, focusing on leveraging the OpenAI API for enhanced network automation. It discusses the use of retrieval augmented generation (RAG) with Langchain for intelligent data handling and introduces rapid prototyping with Streamlit, showcasing the potential for AI to revolutionize network management processes.

Appendix A, "Writing Your Own Parser": This appendix covers how to contribute to the genieparser library (https://github.com/CiscoTestAutomation/genieparser) by creating a new parser for a Cisco IOS XE show command.

Appendix B, "Secret Strings": This appendix covers how to protect the sensitive data in your testbed.yaml files through secret strings.

Credits

Figure 1.1: chinnappa/123RF Figures 5.2a, 5.7-5.9, 5.13, 5.14, 13.2a, 21.1, AppA 1, 2, 4-7: GitHub, Inc Figures 5.15-5.17, 7.24-7.34: Microsoft Corporation Figure 20.26a: Jenkins Figures 21.3, 21.4: GitLab B.V. Figures 23.3-23.6: Snowflake Inc This page intentionally left blank

Chapter 9

pyATS Triggers and Verifications

Automated network tests face challenges in achieving futureproofing due to the dynamic nature of networks and their evolving requirements. The pyATS library (Genie) provides the Genie Harness to execute network tests with dynamic and robust capabilities. The Genie Harness is part of the pyATS library (Genie), which is built on the foundation of pyATS. It introduces the ability for engineers to create dynamic, event-driven tests with the use of processors, triggers, and verifications. In this chapter, the following topics are going to be covered:

- Genie objects
- Genie Harness
- Triggers
- Verifications

The Genie Harness can be daunting for new users of the pyATS library (Genie), as there are many configuration options and parameters. The focus of this chapter will be to provide an overview of Genie Harness and its features and wrap up with a focus on triggers and verifications. By the end of the chapter, you'll understand the powerful capabilities of Genie Harness and how to write and execute triggers and verifications.

Genie Objects

One of the hardest parts of network automation is figuring out how to normalize and structure the data returned from multiple network device types. For example, running commands to gather operational data (CPU, memory, interface statistics, routing state, and so on) from a Cisco IOS XE switch, IOS XR router, and an ASA firewall may have different, but similar, output. How do we account for the miniscule differences across
the different outputs? The pyATS library (Genie) abstracts the details of the parsed data returned from devices by creating network OS-agnostic data models. Two types of Genie objects represent these models: Ops and Conf. In the following sections, you'll dive into the details of each object.

Genie Ops

As you may be able to guess from the name, the Genie Ops object learns everything about the operational state of a device. The operational state data is represented as a structured data model, or feature. A *feature* is a network OS-agnostic data model that is created when pyATS "learns" about a device. Multiple commands are executed on a device and the output is parsed and normalized to create the structure of the feature. You can access the learned feature data by accessing **<feature>.info**. To learn more information about the available models, check out the Genie models documentation (https://pubhub.devnetcloud.com/media/genie-feature-browser/docs/#/models). Example 9-1 shows how to instantiate an Ops object for BGP and learn the BGP feature on a Cat8000v device. Figure 9-1 shows the associated output.

Example 9-1 Genie Ops Object

```
from genie import testbed
from genie.libs.ops.bgp.iosxe.bgp import Bgp
import pprint
# Load Genie testbed
testbed = testbed.load(testbed="testbed.yaml")
# Find the device using hostname or alias
uut = testbed.devices["cat8k-rt2"]
uut.connect()
# Instantiate the Ops object
bgp_obj = Bgp(device=uut)
# This will send many show commands to learn the operational state
# of BGP on this device
bgp_obj.learn()
pprint.pprint(bgp_obj.info)
```

Figure 9-1 Genie Ops Object Output

Genie Conf

The Genie Conf object allows you to take advantage of Python's object-oriented programming (OOP) approach by building logical representations of devices and generating configurations based on the logical device and its attributes. Just like Genie Ops, the structure of Genie Conf objects is based on the feature models. The best way to describe the Conf object is by example. Example 9-2 shows how to build a simple network interface object for an IOS XE device. You'll notice that the code is very comprehensive, specifically for a network engineer, and will even generate the configuration!

Example 9-2 Genie Conf Object

```
from genie import testbed
from genie.libs.conf.interface.iosxe.interface import Interface
import pprint
# Load Genie testbed
testbed = testbed.load(testbed="testbed.yaml")
# Find the device using hostname or alias
uut = testbed.devices["cat8k-rt2"]
# Instantiate the Conf object
interface_obj = Interface(device=uut, name="Loopback100")
# Add attributes to the Interface object
interface_obj.description = "Managed by pyATS"
interface_obj.ipv4 = "1.1.1.1"
interface_obj.ipv4.netmask = "255.255.255.255"
interface obj.shutdown = False
```

```
# Build the configuration for the interface
print(interface_obj.build_config(apply=False))
! Code Execution Output
interface Loopback100
description Managed by pyATS
ip address 1.1.1.1 255.255.255.255
no shutdown
exit
```

You may notice the last line of code actually builds the necessary configuration for you! To go one step further, you can add the interface object to a testbed device and push the configuration to a device. The Conf object allows you to drive the configuration of network devices with Python, which takes you one step further into the world of automation.

Genie Harness

The Genie Harness is structured much like a pyATS testscript. There are three main sections: Common Setup, Triggers and Verifications, and Common Cleanup. The Common Setup section will connect to your devices, take a snapshot of current system state, and optionally configure the devices, if necessary. The Triggers and Verifications section, which you will see later in the chapter, will execute the triggers and verifications to perform tests on the devices. This is where the action happens! The Common Cleanup section confirms that the state of the devices is the same as their state in the Common Setup section by taking another snapshot of the current system state and comparing it with the one captured in the Common Setup section.

gRun

The first step to running jobs with Genie Harness is creating a job file. Job files are set up much like a pyATS job file. Within the job file there's a main function that is used as an entry point to run the job(s). However, instead of using **run()** within the main function to run the job, you must use **gRun()**, or the "Genie Run" function. This function is used to execute pyATS testscripts with additional arguments that provide robust and dynamic testing. Datafiles are passed in as arguments, which allows you to run specific triggers and verifications. Example 9-3 shows a Genie job file from the documentation that runs one trigger and one verification.

```
Example 9-3 Genie Harness – Job file
```

```
from genie.harness.main import gRun

def main():
    # Using built-in triggers and verifications
    gRun(trigger_uids=["TriggerSleep"],
    verification_uids=["Verify_IpInterfaceBrief"])
```

The trigger and verification names in Example 9-3 are self-explanatory: sleep for a specified amount of time and parse and verify the output of **show ip interface brief** command. Each trigger and verification is part of the pyATS library. If these were custom-built triggers or if a testbed device did not have the name or alias of "uut" in your testbed file, you would need to create a mapping datafile. A mapping datafile is used to create a relationship, or mapping, between the devices in a testbed file and Genie. It's required if you want to control multiple connections and connection types (CLI, XML, YANG) to testbed devices. By default, Genie Harness will only connect to the device with the name or alias of "uut," which represents "unit under testing." The uut name/alias requirement allows Genie Harness to properly load the correct default trigger and verification datafiles. Otherwise, you must include a list of testbed devices to the triggers/verifications in the respective datafile. If this doesn't make sense, don't worry; we will touch on datafiles and provide examples further in the chapter that should help provide clarity.

To wrap up the Genie Harness example, the Genie job is run with the same Easypy runtime environment used to run pyATS jobs. To run the Genie job from the command line, enter the following:

```
(.venv)dan@linux-pc# pyats run job {job file}.py --testbed-file {/path/to/testbed}
```

Remember to have your Python virtual environment activated! In the next section, we will jump into datafiles and how they are defined.

Datafiles

The purpose of a datafile is to provide additional information about the Genie features (triggers, verifications, and so on) you would like Genie Harness to run during a job. For example, the trigger datafile may specify which testbed devices a trigger should run on in the job. There are many datafiles available in Genie Harness. However, many of them are optional and are only needed if you're planning to modify the default datafiles provided. The default datafiles can be found at the following path:

```
$VIRTUAL_ENV/lib/python<version>/site-packages/genie/libs/sdk/genie_
yamls
```

Within the genie_yamls directory, you'll find default datafiles that apply to all operating systems (OSs) and others that are OS-specific. These default datafiles are only implicitly

activated when a testbed device has either a name or alias of uut. If there isn't a testbed device with that name or alias, the default datafile will not be implicitly passed to that job. I would highly recommend checking out (not editing) the default datafiles. If you'd like to edit one, you may create a new datafile in your local directory and extend the default one—but don't jump too far ahead yet! This topic will be covered later in the chapter. Here's a list of the different datafiles that can be passed to **gRun**:

- Testing datafile
- Mapping datafile
- Verification datafile
- Trigger datafile
- Subsection datafile
- Configuration datafile
- PTS datafile

Each datafile serves a purpose to a specific Genie Harness feature, but one only needs to be specified if you are deviating from the provided default datafile. For example, the Profile the System (PTS) default datafile only specifies to run on the testbed device with the alias uut. If you would like it to run on more devices, you'll need to create a pts_data-file.yaml file that maps devices to the device features you want profiled by PTS and include the pts datafile argument to gRun.

Device Configuration

Applying device configuration is always a hot topic when it comes to network automation. In the context of pyATS and the pyATS library (Genie), the focus is on applying (and reverting) the configuration during testing. In many cases, you'll want to test a feature by configuring it on a device, testing its functionality, and removing the configuration before the end of testing. The pyATS library (Genie) provides many ways to apply configuration to devices. Here are some of the options you have to configure a network device during testing:

- Manual configuration before testing begins (not recommended).
- Automatically apply the configuration to the device in the Common Setup and Common Cleanup sections with TFTP/SCP/FTP. A config.yaml file can be provided to the **config_datafile** argument of **gRun**, which specifies the configuration to apply.
- Automatically apply the configuration to the device in the Common Setup and Common Cleanup sections using Jinja2 template rendering. The Jinja2 template filename will be passed to gRun using the jinja2_config argument and the device variables will be passed as key-value pairs using the jinja2_arguments argument.

You will dive into Jinja2 templates further and how to use them to generate configurations in Chapter 10, "Automated Configuration Management." For now, just understand that you can standardize the configuration being pushed to the network devices under testing using configuration templates and a template rendering engine (Jinja2) to render the templates with device variables, resulting in complete configuration files.

All configurations should be built using the **show running** style, which means you create your configuration files how they would appear when you view a device's configuration using the **show running-config** command. This differs from how you would configure a device interactively via SSH using the **configure terminal** approach.

After the devices under testing have been configured, the pyATS library learns the configuration of the devices via the check_config subsection. The check_config subsection runs twice: once during Common Setup and another time during Common Cleanup. It collects the **running-config** of each device in the topology and compares the two configuration "snapshots" to ensure the configuration remains the same before and after testing.

PTS (Profile the System)

Earlier in this chapter, you saw examples of device features that can be learned (for example, BGP) during testing. This is made possible by the network OS-agnostic models built into pyATS. These models create the foundation for building reliable data structures and provide the ability to parse data from the network.

PTS provides the ability to "profile" the network during testing. PTS creates profile snapshots of each feature in the Common Setup and Common Cleanup sections. PTS can learn about all the device features, or a specific list of device features can be provided as a list to the **pts_features** argument of **gRun**. Example 9-4 shows how **gRun** is called in a job file with a list of features passed to the **pts features** argument.

Example 9-4 PTS Feature

```
from genie.harness.main import gRun
def main():
    # Profiling built-in features (models) w/o the PTS datafile
    gRun(pts_features=["bgp", "interface"])
```

Along with having the ability to profile a subset of features/device commands, PTS, by default, will only run on the device with the device alias uut. To have more devices profiled by PTS, you'll need to supply a pts_datafile.yaml file. The datafile can provide a list of devices to profile and describe specific attributes to ignore in the output when comparing snapshots (such as timers, uptime, and dates). Example 9-5 shows a PTS datafile, and Example 9-6 shows the updated **gRun** call, with the **pts datafile** argument included. **Example 9-5** *PTS Datafile – ex0906_pts_datafile.yaml*

```
extends: "%CALLABLE{genie.libs.sdk.genie_yamls.datafile(pts)}"
bgp:
    devices: ["cat8k-rt1", "cat8k-rt2"]
    exclude:
        - up_time
interface:
    devices: ["cat8k-rt1", "cat8k-rt2"]
```

Example 9-6 PTS Datafile Argument

```
from genie.harness.main import gRun

def main():
    # Profiling built-in features (models) w/ the PTS datafile
    gRun(pts_features=["bgp", "interface"],
    pts_datafile="ex0906_pts_datafile.yaml")
```

PTS Golden Config

PTS profiles the operational state of the network devices under testing, but how do we know the state is what we expect? PTS provides a "golden config" snapshot feature that compares the profiles learned by PTS to what is considered the golden snapshot. Each job run generates a file named pts that is saved to the pyATS archive directory of the job. Any PTS file can be moved to a fixed location and used as the golden snapshot. Like the **pts_datafile** argument, the **pts_golden_config** argument can be passed to **gRun**, which points to the golden PTS snapshot used to compare against the current test run snapshots.

There's a lot to digest with Genie Harness as well as a lot of different options, and an understanding of these features and when to use them is critical. In the following sections, we will turn our attention to triggers and verifications. Let's take a look at verifications first, as triggers rely on them to perform properly.

Verifications

A verification runs one or multiple commands to retrieve the current state of the device. The main purpose of a verification is to capture and compare the operational state before and after a trigger, or set of triggers, performs an action on a device. The state of the device can be retrieved via the multiple connection types offered by the pyATS library (CLI, YANG, and so on). Verifications typically run in conjunction with triggers to verify the trigger action did what it was supposed to do and to check for unexpected results, such as changes to a feature you didn't initiate.

Verification Types

Verifications can be broken down into two types: global and local. The difference between the two is related to scoping and when each verification type runs within a script.

- Global verifications: Global verifications are used to capture a snapshot of a device before a trigger is executed. Global verifications run immediately after the Common Setup section and before a trigger in a script. If more than one trigger is executed, subsequent snapshots are captured before and after each trigger using the same set of verifications.
- Local verifications: Local verifications are independent of global verifications and run as subsections within a trigger. More specifically, a set of snapshots is taken before a trigger action and a subsequent set is taken after to compare to the first one. Local verifications confirm the trigger action did what it was supposed to do (configure/unconfigure, shut/no shut, and so on).

Figure 9-2 shows where and when the different verification types run in a Genie job.



Figure 9-2 Verification Execution

Verification Datafile

A verification datafile is used to customize the execution of built-in or custom verifications. Like other datafiles, verification_datafile.yaml must be provided to **gRun** using the **verification_datafile** argument. Example 9-7 shows a verification datafile that extends the default verification datafile (via the **extends**: key) and overrides the default setting of connecting to only the "uut" device (via the **devices**: key) for the **Verify_Interfaces** verification. If you wanted to change the list of devices to connect to for another verification, you would need to add that verification in the datafile.

Example 9-7 Verification Datafile – ex0908_verification_datafile.yaml

```
# Extend default verification datafile to inherit the required keys
# (class, source, etc.) per the verification datafile schema
extends: "%CALLABLE{genie.libs.sdk.genie_yamls.datafile(verification)}"
Verify_Interfaces:
    devices: ["cat8k-rt1", "cat8k-rt2"]
```

In order to run the verifications listed in the datafile, or any other built-in verifications, you'll need to include them in the **verification_uids** argument to **gRun**. Example 9-8 shows how to run the Verify_Interfaces verification with the verification datafile from Example 9-7.

Example 9-8 gRun – Verifications

```
from genie.harness.main import gRun

def main():
    gRun(
    verification_uids=["Verify_Interfaces"],
    verification_datafile="ex0908_verification_datafile.yaml"
    )
```

Writing a Verification

The process in which a feature is verified during testing may be different for different use cases. If a built-in verification does not suffice, the pyATS library (Genie) allows you to create your own verification.

There are several ways to create your own verification. You can use a Genie Ops feature (model), a parser, or callable. For the Genie Ops feature and parser options, you can use an existing model or parser, or you can create your own. The last option, using a callable, is discouraged, as it isn't OS-agnostic and does not provide extensibility to use different

management interfaces (CLI, YANG, and so on). Example 9-9 shows a custom verification built with the **show bgp all** parser. Take note of the list of excluded values from the parsed data (found under the **exclude**: key). The reason is because many of these values are dynamic (such as timers, counters, and so on) and are almost guaranteed to be different between snapshots. Remember, if a parsed value is different between snapshots, the verification will fail.

Example 9-9 Custom Verification – ex0910_verification_datafile.yaml

```
# Local verification datafile that already extends the default datafile
extends: verification datafile.yaml
Verify_Bgp:
    cmd:
        class: show bgp.ShowBgpAll
       pkg: genie.libs.parser
    context: cli
    source:
        class: genie.harness.base.Template
    devices: ["cat8k-rt1", "cat8k-rt2"]
    iteration:
        attempt: 5
       interval: 10
    exclude:
    - if_handle
    - keepalives
    - last reset
    - reset_reason
    - foreign port
    - local_port
    - msg_rcvd
    - msg_sent
    - up down
    - bgp_table_version
    - routing_table_version
    - tbl_ver
    - table_version
    - memory_usage
    - updates
    - mss
    - total
    - total bytes
    - up_time
```

```
bgp_negotiated_keepalive_timers
hold_time
keepalive_interval
sent
received
status_codes
holdtime
router_id
connections_dropped
connections_established
advertised
prefixes
routes
state_pfxrcd
```

To run the custom verification, you follow the same process as running any other verification. Pass the verification datafile with the custom verification to **gRun** via the **verification_datafile** argument and add the custom verification name to the list of verifications in the **verification_uids** argument. Example 9-10 shows the updated **gRun** call with the custom verification name and datafile. Remember, the custom verification datafile (Example 9-9) extends the original verification datafile (Example 9-7), which essentially inherits all the built-in verifications from the pyATS library (Genie).

Example 9-10 gRun – Custom Verification

```
from genie.harness.main import gRun

def main():
    gRun(
    verification_uids=["Verify_Interfaces", "Verify_Bgp"],
    verification_datafile="ex0910_verification_datafile.yaml"
    )
```

Triggers

Triggers perform a specific action, or a sequence of actions, on a device to alter its state and/or configuration. As examples, actions may include adding/removing parts of a configuration, flapping protocols/interfaces, or performing high availability (HA) events such as rebooting a device. The important part to understand is that triggers are what alter the device during testing.

The pyATS library (Genie) has many prebuilt triggers available for Cisco IOS/IOS XE, NX-OS, and IOS XR. All prebuilt triggers are documented, describing what happens

when the trigger is initiated and what keys/values to include in the trigger datafile specifically for that trigger. For example, the **TriggerShutNoShutBgpNeighbors** trigger performs the following workflow:

- **1.** Learn BGP Ops object and verify it has "established" neighbors. If there aren't any "established" neighbors, skip the trigger.
- 2. Shut the BGP neighbor that was learned from step 1 with the BGP Conf object.
- 3. Verify the state of the learned neighbor(s) in step 2 is "down."
- 4. Unshut the BGP neighbor(s).
- 5. Learn BGP Ops again and verify it is the same as the BGP Ops snapshot in step 1.

As you might recall from earlier in this chapter, the Genie Ops object represents a device/ feature's operational state via a Python object, and the Genie Conf object represents a feature, as a Python object, that can be configured on a device. The focus of the Conf object is *what* feature you want to apply on the device, not *how* to apply it per device (OS) platform. This allows a network engineer to focus on the network features being tested and not on the low-level details of how the configuration is applied.

Now that there's a general understanding of what triggers do, let's check out how they can be configured using a trigger datafile.

Trigger Datafile

As with other features of the pyATS library (Genie), to run triggers, there needs to be a datafile—more specifically, a trigger datafile (trigger_datafile.yaml). The pyATS library provides a default trigger datafile found in the same location as all the other default datafiles, discussed earlier in the chapter. However, if you want to customize any specific trigger settings, such as what devices or group of devices to run on during testing (any device besides uut), or to run a custom trigger, you'll need to create your own trigger datafile. A complete example can be found at the end of the chapter that includes both triggers and verifications, but let's focus now on just triggers in a brief example. Example 9-11 shows a custom trigger file that flaps the OSPF process on the targeted devices (iosv-0 and iosv01). Example 9-12 shows how to include the appropriate trigger and trigger datafile in the list of arguments to **gRun**.

Example 9-11 *Trigger Datafile – ex0912_trigger_datafile.yaml*

```
extends: "%CALLABLE{genie.libs.sdk.genie_yamls.datafile(trigger)}"
# Custom trigger - created in Example 9-14
TriggerShutNoShutOspf:
# source imports the custom trigger, just as you would any other Python class
source:
    class: ex0915_custom_trigger.ShutNoShutOspf
devices: ["iosv-0", "iosv-1"]
```

Example 9-12 gRun – Triggers and Trigger Datafile

```
from genie.harness.main import gRun

def main():
    gRun(
    trigger_uids=["TriggerShutNoShutOspf"],
    trigger_datafile="ex0912_trigger_datafile.yaml"
    )
```

Trigger Cluster

The last neat trigger feature to cover is the ability to execute a group of multiple triggers and verifications in one cluster trigger. First, a trigger datafile must be created with the list of triggers and verifications, the order in which to run them, and a list of testbed devices to run them against. Example 9-13 shows a trigger datafile configured for a trigger cluster and the accompanying test results if it was run.

```
Example 9-13 Trigger Cluster
```

```
TriggerCombined:
    sub verifications: ['Verify BqpVrfAllAll']
   sub_triggers: [ 'TriggerSleep', 'TriggerShutNoShutBgp']
    sub order: ['TriggerSleep', 'Verify BqpVrfAllAll',
    'TriggerSleep', 'TriggerShutNoShutBgp', 'Verify BgpVrfAllAll']
    devices: ['uut']
-- TriggerCombined.uut
                                                                         PASSED
   |-- TriggerSleep_sleep.1
                                                                         PASSED
   -- TestcaseVerificationOps_verify.2
                                                                         PASSED
   |-- TriggerSleep sleep.3
                                                                         PASSED
   |-- TriggerShutNoShutBgp_verify_prerequisite.4
                                                                         PASSED
      |-- Step 1: Learning 'Bgp' Ops
                                                                         PASSED
       | -- Step 2: Verifying requirements
                                                                         PASSED
       '-- Step 3: Merge requirements
                                                                         PASSED
   |-- TriggerShutNoShutBgp_shut.5
                                                                         PASSED
       '-- Step 1: Configuring 'Bgp'
                                                                         PASSED
   | -- TriggerShutNoShutBgp verify shut.6
                                                                         PASSED
       '-- Step 1: Verifying 'Bgp' state with ops.bgp.bgp.Bgp
                                                                         PASSED
   |-- TriggerShutNoShutBgp_unshut.7
                                                                         PASSED
       '-- Step 1: Unconfiguring 'Bqp'
                                                                         PASSED
   -- TriggerShutNoShutBgp_verify_initial_state.8
                                                                         PASSED
       '-- Step 1: Verifying ops 'Bgp' is back to original state
                                                                         PASSED
   '-- TestcaseVerificationOps verify.9
                                                                         PASSED
```

You may notice that the triggers have accompanying local verifications that run before and after the trigger is run to ensure the action was actually taken against the device. This is the true power of triggers. One of the biggest reasons people are skeptical about network automation is due to the lack of trust. Did this automation script/test really do what it's supposed to do? Triggers provide that verification out of the box through global and local verifications.

What if we wanted to build our own trigger with verifications? In the next section, you'll see how to do just that!

Writing a Trigger

The pyATS library (Genie) provides the ability to write your own triggers. A trigger is simply a Python class that has multiple tests in it that either configure, verify, or unconfigure the configuration or device feature you're trying to test.

To begin, your custom trigger must inherit from a base **Trigger** class. This base class contains common setup and cleanup tasks that help identify any unexpected changes to testbed devices not currently under testing (for example, a device rebooting). For our custom trigger, we are going to shut and unshut OSPF. Yes, this trigger already exists in the library, but it serves as a great example when you're beginning to create custom triggers. The workflow is going to look like this:

- 1. Check that OSPF is configured and running.
- 2. Shut down the OSPF process.
- **3.** Verify that OSPF is shut down.
- 4. Unshut the OSPF process.
- 5. Verify OSPF is up and running.

In Examples 9-14 and 9-15, you'll see the code to create the custom OSPF trigger and the associated job file, running it with **gRun**. To run the job file, you'll need the following files:

- ex0915_custom_trigger.py
- ex0916_custom_trigger_job.py
- ex0915_custom_trigger_datafile.yaml
- testbed2.yaml

The testbed2.yaml file has two IOSv routers, named "iosv-0" and "iosv-1," running OSPF. The file ex0915_custom_trigger_datafile.yaml is used to map the custom OSPF triggers and the testbed devices:

```
# ex0915_custom_trigger_datafile.yaml
extends: "%CALLABLE{genie.libs.sdk.genie_yamls.datafile(trigger)}"
# Custom trigger
TriggerMyShutNoShutOspf:
    # source imports the custom trigger
    source:
        class: ex0915_custom_trigger.MyShutNoShutOspf
    devices: ["iosv-0", "iosv-1"]
```

Example 9-14 Custom Trigger and Job File

```
import time
import logging
from pyats import aetest
from genie.harness.base import Trigger
from genie.metaparser.util.exceptions import SchemaEmptyParserError
log = logging.getLogger()
class MyShutNoShutOspf(Trigger):
"""Shut and unshut OSPF process. Verify both actions."""
                 @aetest.setup
                 def prerequisites(self, uut):
                 """Check whether OSPF is configured and running."""
                 # Checks if OSPF is configured. If not, skip this trigger
                 try:
                                  output = uut.parse("show ip ospf")
                 except SchemaEmptyParserError:
                     self.failed(f"OSPF is not configured on device {uut.name}")
                 # Extract the OSPF process ID
                 self.ospf_id = list(output["vrf"]["default"]["address_family"] \
                 ["ipv4"] ["instance"].keys()) [0]
```

```
# Checks if the OSPF process is enabled
ospf_enabled = output["vrf"] ["default"] ["address_family"] \
["ipv4"] ["instance"] [self.ospf_id] ["enable"]
if not ospf enabled:
      self.skipped(f"OSPF is not enabled on device {uut.name}")
@aetest.test
def ShutOspf(self, uut):
"""Shutdown the OSPF process"""
uut.configure(f"router ospf {self.ospf_id}\n shutdown")
time.sleep(5)
@aetest.test
def verify ShutOspf(self, uut):
"""Verify ShutOspf worked"""
output = uut.parse("show ip ospf")
ospf enabled = output["vrf"]["default"]["address family"] \
["ipv4"] ["instance"] [self.ospf id] ["enable"]
if ospf_enabled:
          self.failed(f"OSPF is enabled on device {uut.name}")
@aetest.test
def NoShutOspf(self, uut):
"""Unshut the OSPF process"""
uut.configure(f"router ospf {self.ospf id}\n no shutdown")
@aetest.test
def verify NoShutOspf(self, uut):
"""Verify NoShutOspf worked"""
output = uut.parse("show ip ospf")
ospf enabled = output["vrf"]["default"]["address family"] \
["ipv4"] ["instance"] [self.ospf id] ["enable"]
if not ospf_enabled:
           self.failed(f"OSPF is enabled on device {uut.name}")
```





Figure 9-3 shows some sample job output.

Task Result Details	+
Task-1: genie testscript	+
I common setup	PASSED
connect	PASSED
configure	SKIPPED
<pre> configuration snapshot</pre>	PASSED
save_bootvar	PASSED
learn_system_defaults	PASSED
initialize_traffic	SKIPPED
PostProcessor-1	PASSED
<pre> TriggerMyShutNoShutOspf.iosv-0</pre>	PASSED
prerequisites	PASSED
ShutOspf	PASSED
verify_ShutOspf	PASSED
NoShutOspf	PASSED
<pre>` verify_NoShutOspf</pre>	PASSED
<pre> TriggerMyShutNoShutOspf.iosv-1</pre>	PASSED
prerequisites	PASSED
ShutOspf	PASSED
verify_ShutOspf	PASSED
NoShutOspf	PASSED
<pre>` verify_NoShutOspf</pre>	PASSED
` common_cleanup	PASSED
<pre> verify_configuration_snapshot</pre>	PASSED
<pre> stop_traffic</pre>	SKIPPED
` PostProcessor-1	PASSED

Figure 9-3 Job Results

Trigger and Verification Example

Now it's time to combine the triggers and verifications into one complete example. In the example, we will build on previous examples where we flap (shut/unshut) the OSPF process on two IOSv routers. The trigger has local verifications that will confirm OSPF

is indeed shut down and confirm that it comes up after being unshut. In addition to the local verifications defined in the trigger, the job will also introduce a global verification to check the router link states (LSA Type 1) in the OSPF LSDB (link state database). This global verification will run before and after the trigger. This allows us to confirm that LSA Type 1 packets are being exchanged before and after testing and that the router link types have not changed during testing.

Let's see how this example can be implemented and executed using Genie Harness and also within a pyATS testscript.

Genie Harness (gRun)

This whole chapter has been focused on Genie Harness, so let's not rehash the details. Example 9-16 shows an example of the job file, trigger datafile, and verification datafile used to execute the job. Figure 9-4 shows the associated test results.

Example 9-16 *Trigger and Verification Example – ex0917_complete_example.py*

```
# Job file - ex0917_complete_example.py
from genie.harness.main import gRun
def main():
gRun (
        trigger_uids=["TriggerShutNoShutOspf"],
         trigger_datafile="ex0917_trigger_datafile.yaml",
        verification_uids=["Verify_IpOspfDatabaseRouter"],
         verification datafile="ex0917 verification datafile.yaml"
# Trigger datafile - ex0917_trigger_datafile.yaml
extends: "%CALLABLE{genie.libs.sdk.genie yamls.datafile(trigger)}"
# Custom trigger - created before Example 9-14
TriggerShutNoShutOspf:
# source imports the custom trigger, just as you would any other Python class
 source:
   class: ex0915_custom_trigger.MyShutNoShutOspf
 devices: ["iosv-0", "iosv-1"]
```

```
# Verification datafile - ex0917_verification_datafile.yaml
extends: "%CALLABLE{genie.libs.sdk.genie_yamls.datafile(verification)}"
Verify_IpOspfDatabaseRouter:
   devices: ["iosv-0", "iosv-1"]
```

Task Result Summary PASSED Task-1: genie_testscript.common_setup PASSED Task-1: genie_testscript.Verifications.TriggerShutNoShutOspf.iosv-0 Task-1: genie_testscript.TriggerShutNoShutOspf.iosv-0 PASSED Task-1: genie testscript.Verifications.TriggerShutNoShutOspf.iosv-1 PASSED Task-1: genie_testscript.TriggerShutNoShutOspf.iosv-1 PASSED Task-1: genie_testscript.Verifications.post PASSED Task-1: genie_testscript.common_cleanup PASSED Task Result Details Task-1: genie_testscript PASSED -- common setup |-- connect PASSED -- configure SKIPPED -- configuration_snapshot PASSED PASSED -- save_bootvar -- learn_system_defaults PASSED SKIPPED -- initialize_traffic PostProcessor-1 PASSED Verifications.TriggerShutNoShutOspf.iosv-0 PASSED – Verify_IpOspfDatabaseRouter.iosv-0.1 PASSED -- verifv PASSED – Verify_IpOspfDatabaseRouter.iosv-1.1 PASSED -- verify PASSED — TriggerShutNoShutOspf.iosv—0 PASSED |-- prerequisites PASSED -- ShutOspf PASSED -- verify_Shut0spf PASSED PASSED I -- NoShutOspf -- verify_NoShut0spf PASSED -- Verifications.TriggerShutNoShutOspf.iosv-1 PASSED -- Verify IpOspfDatabaseRouter.iosv-0.2 PASSED -- verify PASSED Verify_Ip0spfDatabaseRouter.iosv-1.2 PASSED -- verify PASSED TriggerShutNoShutOspf.iosv-1 PASSED |-- prerequisites PASSED -- ShutOspf PASSED -- verify_Shut0spf PASSED PASSED -- NoShutOspf -- verify_NoShutOspf PASSED -- Verifications.post PASSED Verify IpOspfDatabaseRouter.iosv-0.3 PASSED PASSED -- verify Verify_IpOspfDatabaseRouter.iosv-1.3 PASSED PASSED -- verify - common_cleanup PASSED |-- verify_configuration_snapshot PASSED -- stop_traffic SKIPPED –– PostProcessor–1 PASSED

Figure 9-4 Trigger and Verification Example Results

pyATS

Triggers and verifications can be run in a pyATS testscript as a testcase or within a test section. Custom trigger and verification datafiles can be provided using the --trigger-datafile and --verification-datafile arguments when calling a pyATS job.

To run it as its own testcase, you'll just need to create a class that inherits from **GenieStandalone**, which inherits from the pyATS **Testcase** class. The inherited class you create will provide a list of triggers and verifications. The same trigger and verification datafiles will be included as options when running the pyATS job via the command line.

The second way to include triggers and verifications in a pyATS testscript is by including them in an individual test section, as part of a pyATS testcase. The **run_genie_sdk** function allows you to run triggers or verifications as steps within a section.

Example 9-17 shows how to include triggers and verifications as their own testcase and within an existing subsection in a pyATS testscript. To run the testscript, you'll need to add the **--trigger-datafile** and **--verification-datafile** arguments with the appropriate datafiles to map the custom trigger and verifications to the additional devices. These datafiles are not included in the example. Figure 9-5 shows the testscript results.

Example 9-17 Triggers and Verifications in pyATS Testscript

```
from pyats import aetest
import genie
from genie.harness.standalone import GenieStandalone, run_genie_sdk
class CommonSetup(aetest.CommonSetup):
    """ Common Setup section """
    @aetest.subsection
    def connect(self, testbed):
        """Connect to each device in the testbed."""
        genie_testbed = genie.testbed.load(testbed)
        self.parent.parameters["testbed"] = genie_testbed
        genie_testbed.connect()
# Call Triggers and Verifications as independent pyATS testcase
class GenieOspfTriggerVerification(GenieStandalone):
"""Shut/unshut the OSPF process and verify LSA Type 1 packets are still being
exchanged before and after testing."""
```

```
# Must specify 'uut'
         # If other devices are included in the datafile(s), they will be tested
         uut = "iosv-0"
         triggers = ["TriggerShutNoShutOspf"]
         verifications = ["Verify_IpOspfDatabaseRouter"]
# Calling Triggers and Verifications within a pyATS section
class tc_pyats_genie(aetest.Testcase):
         """Testcase with triggers and verifications."""
         # First test section
         @aetest.test
         def simple_test_1(self, steps):
         """Sample test section."""
         # Run Genie triggers and verifications
         # Note that you must specify the order of each trigger and verification
         run_genie_sdk(self,
                          steps,
                          ["Verify_IpOspfDatabaseRouter", \
                           "TriggerShutNoShutOspf", \
                           "Verify_IpOspfDatabaseRouter"],
                          uut="iosv-0"
                          )
class CommonCleanup(aetest.CommonCleanup):
"""Common Cleanup section"""
@aetest.subsection
def disconnect_from_devices(self, testbed):
         """Disconnect from each device in the testbed."""
         testbed.disconnect()
```

4	+
Task Result Summary	į
Task-1: ex0918_pyats_testscript.common_setup Task-1: ex0918_pyats_testscript.GenieOspfTriggerVerification Task-1: ex0918_pyats_testscript.tc_pyats_genie Task-1: ex0918_pyats_testscript.common_cleanup	PASSED PASSED PASSED PASSED
+ Task Result Details	
<pre>Task-1: ex0918_pyats_testscript common_setup ` connect Genie0spfTriggerVerification Verify_Ip0spfDatabaseBouter.josy-0_verify.1</pre>	PASSED PASSED PASSED PASSED
<pre> Verify_IpOspfDatabaseRouter.iosv-1_verify.2 TriggerShutNoShutOspf.iosv-0_prerequisites.3 TriggerShutNoShutOspf.iosv-0_ShutOspf.4 TriggerShutNoShutOspf.iosv-0_verify_ShutOspf.5</pre>	PASSED PASSED PASSED PASSED
<pre> TriggerShutNoShutOspf.1osv-0_NoShutOspf.6 TriggerShutNoShutOspf.iosv-0_verify_NoShutOspf.7 TriggerShutNoShutOspf.iosv-1_prerequisites.8 TriggerShutNoShutOspf.iosv-1_ShutOspf.9 TriggerShutNoShutOspf.iosv-1_verify_ShutOspf.10</pre>	PASSED PASSED PASSED PASSED PASSED
<pre> TriggerShutNoShutOspf.iosv-1_NoShutOspf.11 TriggerShutNoShutOspf.iosv-1_NoShutOspf.11 TriggerShutNoShutOspf.iosv-1_verify_NoShutOspf.12 Verify_IpOspfDatabaseRouter.iosv-0_verify.13 Verify_IpOspfDatabaseRouter.iosv-1_verify.14</pre>	PASSED PASSED PASSED PASSED PASSED
<pre> tc_pyats_genie ` simple_test_1 STEP 1: Verify_IpOspfDatabaseRouter.iosv-0 STEP 2: TriggerShutNoShutOspf.iosv-0</pre>	PASSED PASSED PASSED PASSED
<pre>` SIEP 3: Verity_IpUsptDatabaseRouter.losv-0 ` common_cleanup ` disconnect_from_devices</pre>	PASSED PASSED PASSED

Figure 9-5 Triggers and Verifications in pyATS Testscript Results

Summary

This chapter covered a lot of information about the pyATS library (Genie), the Genie Harness, with its many different features, along with triggers and verifications. The Genie Harness allows you to take advantage of the pyATS infrastructure without having to dive-deep into code. The goal of the pyATS library (Genie) is to be modular and robust. Triggers and verifications are a perfect example. They make it easy to quickly build dynamic testcases that change with your network requirements. I highly recommend taking a closer at the code examples in this chapter and trying them out for yourself. Within minutes, you'll see how quickly you can test a network feature with speed and accuracy! This page intentionally left blank

Index

A

abstracted clean stage, 455-456 action/s. 460-461 advanced, 470 looping, 471-475 parallel, 470 run condition, 475–476 custom, 481-482 output, 461 appending to an existing file, 466 *Dq filter*, 461–462 List filter, 462-463 RegEx filter, 461–462 saving to a dictionary, 467 saving to a file, 465-466 variables, 463-465 verifying, 467–470 timeouts, 479–480 Adaptive Cards, 490-491 Jinja2 template, 491, 492–500 sending to Webex, 500-501 AEtest, 176-179

Common Cleanup, 78-79, 179 Common Setup, 76, 178 decorator, 75 design features, 74-75 installation, 74 logic module, 86 loops, 74-75, 104 defining, 104 dynamic, 107-108 parameters, 104–106 object model, 82 container classes, 82-83 function classes, 83-85 TestScript class, 82 pause on phrase, 133–134 processor/s, 91 context, 93 definition and arguments, 92-93 global, 93-94 results, 94 types, 91–92 Reporter, 127 event data, 132-133

report structure, 128–129 results.yaml file, 129-132 runtime behavior, 85-86 parent, 87 section ordering, 88 self keyword, 86–87 Standalone Reporter, 126–127 step/s, 79-80 attributes, 80-81 nested, 81-82 test results, 88 arguments, 90 interaction results, 90 result behavior, 89-90 result objects, 88-89 result rollup, 90–91 testable, 117 testcase, 75, 77-78 cleanup section, 77 setup section, 77 test section, 77 testscript, 75, 78-79 aetest.main() function, 110, 113 agent-based tools, 21-22 agentless tools, 22 Agile, 12–13, 18 Agile Manifesto, 170-171 agility, NetDevOps, 21 AI (artificial intelligence), 591. See also NLP (natural language processing); OpenAI; RAG (retrieval augmented generation) data analysis routing table, 606–609 running configuration, 609-610 embeddings, 593, 595 generative, 591

models and providers, 597–598 OpenAI API, 597 RAG (retrieval augmented generation), 592, 612 textual segment, 593–594 alert, Webex, 10 Ansible, 562-563, 566 **APIC** (Application Policy Infrastructure Controller), 369 API/s (application programming interface/s), 2, 6-7, 27, 192-193. 347-351. See also REST Connector Chat Completions, 597–599 ISON mode, 599 messages parameter, 599 clean, 438-440 configure by jinja, 317 Connect, 360-361, 364-365, 366-367, 374-375, 378 copy to device, 312 DELETE, 357-358, 363, 368, 371, 376-377, 380, 382 device, 313-314 Disconnect, 365, 375 GET, 355-356, 361, 365, 367, 370, 375. 379. 381 get software version(), 351–352 GraphQL, 28 Jinja2, 206 load jinja template, 307 network management, 352-353 NX-, 355 OpenAI, 591, 597 PATCH, 358, 362, 368, 376, 382 pcall, 182, 400-401 error bandling, 404 logging, 404

object, 405-406 performance comparison, 406-409 targets, 402-404 usage, 401–402 POST, 356-357, 361-362, 367-368, 370-371, 375-376, 379, 381 PUT, 358–359, 362–363, 368, 376, 379-380, 383 REST, 27-28 testbed, 61 approval, code, 16 architect, network, 2-3 argument/s clean stage, 445–447 function, 97 processor, 93 propagation, 109 pyATS job, 197–199 script, 96–97 section, 98 standard, 108-109 template, 306 test result, 90 testing, 108 Unicon configure service, 315–316 asynchronous programming, 398 asyncio library, 398 attribute/s dictionaries, 182-183 function class, 84 interface object, 59 link object, 59 parent, 87 step, 80-81 testbed object, 52-53 Aubin, Jean-Benoit, 1

automated network testing, 234-235 feature testing, 271-276 intent validation testing, 267–271 interface testing, 243-244 keys, 247-248 neighbor testing, 259–262 reachability testing, 262-267 **RESTCONF. 252–258** role of SSH in. 244–251 software version testing, 235-243 testing for input errors, 248–251 automation, 6 agent-based tools, 21-22 agentless tools, 22 builds, 16 day-1 offline initial configuration, 5 software image management, 5 learn models, 144 network. 1 parsing, 137 testing, 14 traditional network, 2 velocity, 20 vendor-agnostic, 138-139 awaitables, 398

В

bare metal, 31
base class, 82
BaseStage class, 443
"batteries-included" programming language, 29–30
Beck, Kent, 170
BGP (Border Gateway Protocol) configuration, 566–567

feature testing, 271–276 BIG-IP, 373–374 Connect API, 374–375 DELETE API, 376–377 Disconnect API. 375 **GET API. 375** PATCH API, 376 POST API, 375–376 PUT API, 376 Blitz, 459, 483-484. See also action/s action output, 461 appending to an existing file, 466 *Dq filter*, 461–462 List filter, 462–463 *RegEx filter*, 462 saving to a dictionary, 467 saving to a file, 465–466 variables, 463-465 verifying, 467–470 advanced actions, 470 loop keywords, 471–475 parallel, 470 run condition, 475–476 altering results, 478–479 custom actions, 481–482 custom test sections, 482–483 customizing log messages, 480 development, 481 negative testing, 477 prompt handling, 478 script termination on failure, 477-478 timeouts, 479–480 usage, 481 YAML files, 459, 460-461 breakpoints, 133

brownfield, 17, 173-174 builds, 16, 53-54 business-ready documents, 206-207 CSV file, 208–210 datatable, 227–232 HTML, 225–227 **JSON**, 207 markdown, 210–211 markmap mind map, 212–215 Mermaid class diagram, 217-219 Mermaid entity relationship diagram, 221 - 223Mermaid flowchart, 215-217 Mermaid mind map, 223–225 Mermaid state diagram, 219–221 YAML, 207-208

С

callables, 83, 98 Catalyst Center, 363-364 Connect API, 364–365 Disconnect API. 365 GET API, 365 CD (continuous deployment), 32 CDP (Cisco Discovery Protocol) enabling with pyATS, 260 neighbor ping test, 265 testing with pyATS, 260–262 Cerebrus, 304-305 change management, 337-338 change plans, 561 ChangeBootVariable class, 444–445 Chat Completions API, 597–599 JSON mode, 599 messages parameter, 599 ChatGPT, 591, 612

child process, 399 ChromaDB, 594, 614, 621 CI/CD (continuous integration/ continuous development), 2-3, 7-8, 16, 16–17, 32–33, 64, 557 Docker containers, 505 Drone. 34–36 GitHub Actions, 34 GitLab, 33-34, 559-560, 563-565 Jenkins, 33 in NetDevOps, 560–562 pipeline, 17, 18, 19, 172, 505, 558, 561-562 configuration changes, 565-568 GitHub Actions, 558–559 testing network changes, 568-570 Cisco ACI APIC, 369-370, 371-373 DELETE API, 371 **GET API. 370** POST API, 370–371 Cisco DevNet Always-On IOS-XE Sandbox testbed, 235 class/es base, 82 BaseStage, 443 callables, 83, 98 ChangeBootVariable, 444–445 container. 82–83 context processor, 93 diagram, 217–219 function, 83–85 MoDirectory, 371–372 parser, 637–638 Pcall, 405–406 representer, 655

REST Connector, 354 schema, 635-636 ScriptDiscovery, 125–126 Steps, 80 TestScript, 82 WebInteraction, 90 Clean, 443–445 clean stage template, 443–445 device recovery, 434–436 file validation, 436-437 including a clean stage in testscripts, 440-443 installing, 411 Integrated execution, 437–440 logging, 438–439 software image management, 431-433 Standalone execution, 437–438 supported OS/platforms, 413 supported power cyclers, 414–422 YAML, 423–424 clean stages, 424–427 device groups, 428–431 devices block, 424 clean API, 438-440 clean stage abstracted, 455-456 creating execution order, 450–452 schema and arguments, 445-447 stage steps, 447–450 template, 443-445 docstrings, 452–454 cleanup section, testcase, 77 CLI (command-line interface), 24 job log, 199–202

mock device, 344–345 network features, learning, 322-324 parsing, 148–150 connection file, 150–151 console output, 151–152 *ops output*, 153–155 Python, 30 running pyATS jobs, 196–199 SSH options, 54-55 using RESTCONF, 56 cluster, trigger, 292–293 CML (Cisco Modeling Labs), 562-563 code. 16 VS Code, 23 code approval, 16 asynchronous, 398 builds, 16 coverage, 73 debugging, 133–134 IDE (integrated development environment) new school. 23 old school, 22–23 infrastructure as, 14, 16-17, 19, 23, 32 learn model, 142 linting, 16 quality, 64 regression testing, 73 static analysis, 64 testing, 73 version control. 23 command/s. See also API/s (application programming interface/s); CLI

copy, 312–313 DELETE, 385 genie diff, 327-329, 331-332 genie learn, 325–327 GET. 383-384 learn, 150, 155–156 connection file, 150–151 console belp, 156 console output, 151–152 *ops output*, 153–155 no logging console, 55 parse, 157–158 ping, 260–262 parsing, 262–264 reachability testing, 265–267 pip install pyats, 38–39, 176 pip list, 39–41 POST, 384 PUT, 384–385 pyats create testbed, 56 pyats logs view, 201–202 pyats run job, 115 pyats version check, 42 pyats version update, 43-44, 45-46 Python *import this*, 30 install. 30 show, 139-145, 189 show ip interface brief, 25, 26, 191, 207, 210, 214-215, 625-626 show running-config, 285, 319, 431 show version, 236-243 terminal width 511, 55 Common Cleanup, 78–79, 179 Common Setup container, 76, 178 concurrency, 398

Conf object, Genie, 281–282, 291, 308 - 309config datafile, Genie Harness, 317-319 configuration/s BGP. 566–567 checking, 319 day-1, 4-5 device, 284-285, 309 config datafile, 317–319 embedded file transfer server, 311-313 FileUtils module, 309–311 drift. 8. 18 generating with pyATS, 304 data modeling and validation, 304-305 data templates, 305-307 Genie Conf objects, 308–309 incremental, 8-9 intent-based, 6-7, 18-19, 303-304 pyATS request and push using ChatGPT, 612 re-. 8 snapshot, 319, 322 testbed-wide, 314-315 Unicon, 315-316 untracked, 303 validation, 7 configure by jinja2 API, 317 Connect API, 360-361, 364-365, 366-367, 374-375, 378 connection file, 150-151 container/s, 31, 504-506 class. 82-83 Common Cleanup, 78–79 Common Setup, 76

subsections, 76 Docker, 503, 505-506 step/s, 79-80 attributes, 80-81 nested, 81-82 testcase, 77-78 cleanup section, 77 must-pass, 121–123 randomization, 123-124 setup section, 77 test section, 77 testscript, 75 context processor, 93 template, 306 continuous improvement, 14 contributing to genieparser, 647–650 copy command, 312–313 copy to device API, 312 core dump file, 516 core framework, pyATS, installing, 38-39 coroutines, 398 cosine similarity, 595–596 CPOC (Center of Proof of Concept), 169 CPU, health check, 515-516 creating clean stage execution order, 450–452 schema and arguments, 445 - 447stage steps, 447–450 template, 443–445 Dockerfile, 508, 623–625 mock devices, 343-344 schema, 635-636

testbed, 66–67 virtual environment, 60–61 credentials device, 54 YAML file, 56–57 CSV file, 208–211 cURL (Client URL), 28–29 customizing log messages, 480

D

data center, failure, 21 data model/s, 3-4 Pydantic, 304–305 YANG, 26 database. See vector stores data-driven testing, 94-95 datafile/s, 98-104 Genie Harness, 283–284 trigger, 291–292 verification, 288 Datatables.net. 227–232 day 0 activities, 5–6 initial configuration, 6–7 initial testing and validation, 7 Layer 1, 6 day 1 activities, 7-8 incremental configuration, 8-9 provisioning new endpoints, 9 provisioning new services, 9 day N activities, 9 decommissioning, 11 monitoring, 9–10 responding to events, 10 upgrading, 10-11 day-1 activities

device configuration, 4–5 offline initial configuration, 5 software images, 5 DCNM (Cisco Data Center Network Manager), 380 DELETE API. 382 GET API, 381 PATCH API, 382 POST API, 381 PUT API, 383 debugging, 133-134 declarative Jenkinsfiles, 33 decommissioning, 1 decorator, 75 DELETE API, 357-358, 363, 368, 371, 376–377, 380, 382 **DELETE command**, 385 deployment, automated, 17 design features, AEtest, 74–75 development. See also software development; TDD (test-driven development) Blitz, 481 test-driven. 14 device/s. See also API/s (application programming interface/s); mock device/s API. 307. 313–314 cleaning, 181, 412-413, 438-445. See also Clean configuration, 284-285, 309 checking, 319 config datafile, 317–319 configure_by_jinja2 API, 317 embedded file transfer server, 311-313 FileUtils module, 309–311 testbed-wide, 314-315

Unicon, 315-316 connection abstractions, 60-63 core dump file, 516 credentials, 54 day-1 configuration, 4-5 decommissioning, 11 groups, 428-431 mock, 342 neighbor testing, 259–262 network features, learning, 322-324 object attribute list, 57-58 output, 279-280 power cyclers, 413-414 public Internet access, 266 recovery, 434-436 **REST Connector**, 354 **RESTCONF** validation, 253 terminal settings, 55 DevOps, 1, 13, 557, 560-561. See also CI/CD (continuous integration/continuous development) builds, 16 lifecycle, 14-15 dictionary/ies attribute, 182-183 orderable, 183 saving action output to, 467 Diff library, 162-167 Disconnect API, 365, 375 discovery, testcase, 125-126 Django, 67, 622 DMI (data model interface), 385 do until keyword, 473 Docker, 17, 31, 504. See also container/s containers, 503, 505-506

image, 504, 624 Dockerfile building a Docker image from scratch. 510-513 creating, 508, 623-625 docstrings, clean stage, 452-454 documentation business-ready, 206-207 CSV file, 208-210 datatable, 227–232 HTML. 225–227 ISON. 207 markdown, 210-211 markmap mind map, 212–215 Mermaid class diagram, 217– 219 Mermaid entity relationship diagram, 221–223 Mermaid flowchart, 215-217 Mermaid mind map, 223–225 Mermaid state diagram, 219-221 YAML, 207-208 high-level design, 3 domain name, intent validation testing, 268-269 Dq (dictionary query), 160–162 Dg filter, 461–462 Drone. 34-36 dynamic looping, 107-108 dynamic testbed, 66-68

Ε

Easypy, 179–180, 190 job file, 116–117 multiprocessing, 400

Reporter package, 400 testscript execution, 114–117 .zip folder, 180 efficiency, NetDevOps, 19-20 ElasticSearch, 595 ElasticVue, 595 element managers, 2 ELK (Elastic, Logstash, Kibana), 595 embedded file transfer server, device configuration, 311-313 embeddings, 593, 595, 621 endpoints, provisioning, 9 end-to-end BGP feature testing, 272-276 enforcing intent, 70 event/s GitHub Actions, 34 response, 10 every seconds keyword, 473-474 exception handling, 89 exclude keyword, 467 ExecuteCommand clean stage, 425 extended testbed, 68-70 external sources of truth. 56-60 extreme programming, 170

F

failure data center, 21 network, 321 feature testing, 271–276 Fielding, Roy, "Architectural Styles and Design of a Network-based Software Architecture", 27 FileUtils module, 309–311 filter Dg, 461-462 List, 462-463 RegEx, 461-462 flow control, testscript, 117 running specific testcases, 119-120 skip conditions, 117–119 testcase grouping, 120–121 flowchart, Mermaid, 215-217 for loop, 243, 473 forensics and security analysis, pyATS job recordings, 338 forking, 399, 400 framework. See also Streamlit.io Django, 622 pyATS, 175-176 AEtest, 176–179 Easypy, 179-180 pyats Clean. See pyATS (Python Automated Test Systems), clean Robot, 333–335, 575–576 keywords, 577–578 test cases, 576-577 test libraries, 580 variables, 578-579 function/s.75 .parse(), 159 aetest.main(), 110, 113 arguments, 97 callables, 83, 98 classes. 83–85 coroutines, 398 gRun(), 282–283 learn poll(), 332–333

loop.mark(), 107–108

parametrized, 98 show version, 406–407 testbed.connect(), 196 verify, 332

G

generating configurations with pyATS, 304 data modeling and validation, 304-305 data templates, 305-307 Genie Conf objects, 308–309 generative AI, 591 Genie, 138 Conf object, 281-282, 291, 308-309 device configuration, 284–285 iob file creating, 282-283 trigger and verification names, 283 keywords, 585-588 objects, 279-280 Ops object, 280-281, 291 Robot Framework, 333–335 genie diff command, 327-329, 331-332 Genie Harness, 279, 282, 297–298 config datafile, 317-319 datafiles, 283-284 job file, 283 PTS (profile the system), 285–286 trigger, 290-291, 296-297 cluster, 292–293 *datafile*, 291–292 writing, 293-296 verifications, 286 datafile, 288 writing, 288–290 genie learn command, 325–327

genieparser, 633, 647–650 geospatial analysis, vector data, 595 GET API, 355-356, 361, 365, 367, 370, 375, 379, 381 get software version() API, 351–352 Git, 7-8, 23, 24, 33 GitHub, 24, 34, 142, 558-559 GitLab, 24, 562-563 CI/CD, 33-34, 559-560, 563-565 Webex Teams integration, 571–572 GitOps, 19 global processor, 93-94 gNMI (gRPC Network Management Interface), 389 comparing configs, 393-394 creating an instance of Config, 392 creating Config objects, 392–393 creating ConfigDelta objects with special requirements, 394 fetching device capabilities, 390–391 GetRequest, 391–392 SetRequest, 391 setting up and connecting with clients, 389-390 golden config, PTS, 286 goto statement, 121–122 GraphQL, 28 group, testcase, 120-121 gRPC (Google Remote Procedure Call), 389 gRun() function, 282-283

Η

health check/s core file, 516 CPU and memory, 515–516 custom, creating, 517

integrating with Webex, 489–490 logging, 516 results, 522–523 running as part of pyATS jobs, 523-524 testcase/section selection, 520-522 usage, 523 YAML file, 517–519 processor key, 520 reconnect key, 520 validation. 523 high-level design, 3 hosted hypervisor, 31 HTML logs viewer, 203–205 table, 225–227 Hugging Face, 597 hypervisor, 31

IaC (infrastructure as code), 14, 16–17, 19, 23, 32
IDE (integrated development environment), 22

new school, 23
old school, 22–23

image, software, 5
Image Builder, 507–509, 543–545
import this command, 30
include keyword, 467
incremental configuration, 8–9
inheritance, class, 80
initial configuration, 6–7
input errors, testing for, 248–251
install command, 30

installing AEtest, 74 pyATS, 38–41 pyats Clean, 411 **REST Connector**, 354 XPRESSO, 529–534 Integrated execution, pyATS Clean, 437 - 440intent, 172 -based configuration, 6-7, 18-19, 303-304 -based networking, 68-70 enforcing, 70 validation testing, 267–271 interaction results, 90 interface/s description, intent validation testing, 269 - 271object attribute list, 59 testing, 177–178, 243–244 Internet access, testing on a device, 266 I/O operation, 397, 398 **IOS XE, 366** Connect API, 366–367 DELETE API, 368 **GET API**, 367 PATCH API, 368 POST API, 367-368 PUT API. 368 ios xe version job.py job file, 236 ipaddress module, 160 iterable, 83 IXP (Internet exchange point), 259

J

Jenkins, 33 Jinja2, 189. See also business-ready documents APIs. 206 creating business-ready documents, 206 - 207template, 6-7, 285 Adaptive Card, 491, 492–501 configuration file, 305–307, 317 context, 306 CSV file, 209 HTML, 226 markdown table, 211 markmap mind map, 213 Mermaid class diagram, 218-219 Mermaid entity relationship diagram, 222–223 Mermaid flowchart, 216 Mermaid mind map, 224–225 Mermaid state diagram, 220-221 tags, 205 JIT (just-in-time) manufacturing, 12 job file, 29-30, 190-191 Easypy, 116–117, 179 Genie creating, 282-283 trigger and verification names, 283 Genie Harness, 283 ios xe version job.py, 236 learn all job.py, 339 parsers, 159

job/s GitHub Actions, 34 health checks, 523–524 integrating with Webex, 487–489 logs, 7 pyATS, 179, 190–196 arguments, 197–199 CLI logs, 199–202 running from the CLI, 196–199 recording, 337-341 replaying recorded jobs, 341–342 use cases, 337–339 transforming into XPRESSO, 538-556 Jones, Daniel, 12 JSON (JavaScript Object Notation), 14, 24-25Chat Completions API, 599 loader, 619-620 output, learn(interface) command, 246 - 247saving to a file, 192–196 transforming to CSV, 208–209 transforming to markmap mind map, 212 - 213transforming to YAML, 207-208

Κ

keyword/s, 334, 577–578 do_until, 473 every_seconds, 473–474 Genie, 585–588 include/exclude, 467 loop, 471–472 parallel, 470, 474 pyATS, 582–584
range, 472–473 run_condition, 475–476 self, 86–87 Unicon, 584–585 user, 578 Kibana, 595 Kleenex Clean, 181 known-good state, 324 Krafcik, John, "Triumph of the Lean Production System", 12 Kubernetes, 32, 505

L

LangChain, 592-593, 621 integration with Streamlit.io, 622-623 RAG (retrieval augmented generation), 612-614 *ISON loader*, 619–620 running configuration, questions and responses, 614-618 sourcing JSON or raw text data, 614 vector stores, 614 language models, 593 Layer 1, testing, 6 Lean. 12 learn command, 150, 155–156 connection file, 150–151 console help, 156 console output, 151–152 ops output, 153–155 .learn() method, 244 learn models, 139–140 code, 142 structure, 140–142

learn poll() function, 332–333 learning network features, 322-324 differential results, 327-329 post-change, 326–327 pre-change, 324-326 using Robot Framework, 334–335 library. See also Genie; genieparser; LangChain asyncio, 398 Cerebrus, 304–305 genieparser, 633 LangChain, 592–593 Rich, 234 Unicon, 61–62, 312–313 configure service, 315-316 *copy service*, *312–313* link object attribute list, 59 linting, 16, 63–65 List filter. 462–463 LLM (large language model), 592, 593, 596-597 load jinja template API, 307 logging module, 400 logic testing, 86 logs and logging, 184 AI (artificial intelligence), 603–604 customizing log messages, 480 health check, 516 HTML viewer, 203-205 job, 199–202 pcall, 404 syslog with AI analysis, 604–605 loop.mark() function, 107–108 loop/s AEtest, 74-75, 104 defining, 104 dynamic, 107–108

parameters, 104–106 combining with parallel, 474 do_until keyword, 473 every_seconds keyword, 473–474 for, 243 keywords, 471–472 nested, 475 range keyword, 472–473 value key, 472 low-level design, 3–4 LSA (link-state advertisement), 330

Μ

machine learning, 595 cosine similarity, 595-596 vector data, 594 Manifest files, 187 "Manifesto for Agile Software Development", 12 markdown, 210-211. See also Mermaid markmap mind map, 212-215 Mermaid, 215-217 tables, 210-211 markmap mind map, 212-215 Martin, Robert C., 171 maximum failures, testcase, 124-125 memory, health check, 515–516 Mermaid, 215-217 class diagram, 217-219 entity relationship diagram, 221–223 flowchart, 215-217 mind map, 223-225 state diagram, 219–221 metaparser, 137 method/s, 75

child, 88 decorator, 75 .learn(), 244 MIT (Management Information Tree), 369 mock device/s, 342, 343-344 CLI, 344-345 creating, 343-344 use cases, 342-343 mock testbed, 60-61 MoDirectory class, 371-372 module FileUtils, 309-311 ipaddress, 160 logging, 184, 400 logic, 86 pickle, 399 robot, 582, 585-586 Tcl. 183–184 topology, 180–181 YANG Connector, 385–386 MongoDB, 595 monitoring, network, 9–10, 18 multiprocessing, 178-182, 399-400 must-pass testcases, 121–123

Ν

negative testing, 477 neighbor testing, 259–262 nested loop, 475 nested steps, 81–82 NETCONF, 385, 386–389 NetDevOps, 1, 2, 13, 562. *See also* CI/CD (continuous integration/ continuous development) benefits

agility, 21 efficiency, 19-20 GitOps, 19 intent-based configuration, 18 - 19quality, 21 single source of truth, 18 speed, 20 version and source control, 19 CI/CD (continuous integration/continuous development), 32–33, 560-562, 563-565 deployments, 17 IaC (infrastructure as code), 14 monitoring activities, 18 network architect, 2-3 operations, 17 plan phase, 16 TDD (test-driven development), 14 network/s. See also AEtest: automated network testing architect. 2-3 automation, 1 brownfield, 17 change plans, 561 failure, 321 features, 322-324 high-level design, 3 intent verification, 266-267 intent-based, 68-70 low-level design, 3-4 management, APIs, 352-353 monitoring, 9–10, 18 objects, 74 profiling, 321-322 routing table analysis using AI, 606-609

snapshot, 322 data exclusions, 331-332 differentials, 327 post-change, 326-327 pre-change, 324-326 state comparing, 324 known-good, 324 polling, 332-333 subnet. 3-4 traditional, automation, 2 Nexus Dashboard, 383 DELETE command, 385 GET command, 383–384 POST command, 384 PUT command, 384-385 NLP (natural language processing) cosine similarity, 595–596 embeddings, 593 semantic analysis, 594 textual segment, 593-594 vector data, 594 NMS (network management system), 18 no logging console command, 55 ns help(), 395 **NSO (Cisco Network Services** Orchestrator), 359-360 Connect API. 360–361 DELETE API. 363 GET API, 361 PATCH API, 362 POST API. 361-362 PUT API, 362-363 **NXOS** DELETE API, 357-358 GET API, 355-356

PATCH API, 358 POST API, 356–357 PUT API, 358–359

0

object model, AEtest. See AEtest, object model object/s. See also class/es; method/s attributes, 57–58 Genie Conf, 281–282 Ops, 280-281 interface. 59 iterable, 83 link, 59 Ops, 322 -oriented programming, 74 parent-child relationships, 87 pcall, 405-406 result, 88-89, 185 runtime, 85-86 Steps, 80 testable, 117 testbed, 52–53 variables, 61 offline initial configuration, 5 onboarding, 5-6. See also day 0 OOP (object-oriented programming), 74 OpenAI API. 597 Chat Completions API, 597–599 ISON mode, 599 messages parameter, 599 ChatGPT, 591

OpenAI API, integrating into an interface health check, 600–603 logs, 603–604 syslog with AI analysis, 604–605 Ops object, 280–281, 291, 322 orderable dictionary, 183 OSPF features, learning, 325–326, 334–335 LSA (link-state advertisement), 330 output. See also command/s device, 279–280

Ρ

package, 29-30 packet forwarding, 321-322 parallel keyword, 470, 474 parallelism, 398, 400 parameter/s, 94–95 loop, 104–106 parametrization, 98 properties, 96 relationship model, 95–96 reserved, 98 types, 96–98 parent attribute, 87 parse command, 157-158 .parse() function, 159 parser creating, 638 development environment, 638-640 writing your parser class, 642-644 writing your schema class, 640-642 testing, 644-647

Parser Filter, 146–148 parsing, 137 at the CLI, 148-150 connection file, 150–151 console output, 151–152 ops output, 153–155 Diff library, 162–167 Dq (dictionary query), 160–162 learn models, 139-145 ping command, 262-264 Python script, 159–160 routing table, 607-608 show command, 322–323 show ip interface brief command, 625-626 PATCH API, 358, 362, 368, 376, 382 pause on phrase, 133-134 pcall, 182, 400-401 error handling, 404 logging, 404 object, 405-406 performance comparison, 406–409 targets, 402-404 usage, 401-402 performance, scaling, 397 Peters, Tim, 29-30 pickle module, 399 Pinecone, 594-595 ping, 260-262 parsing, 262-264 reachability testing, 262–267 pip, 30 pip install pyats command, 38–39, 176 pip list command, 39-41

pipeline. See also CI/CD (continuous integration/continuous development), pipeline CI/CD (continuous integration/ continuous development), 172 Drone, 35 pipes, 399 plan phase, NetDevOps, 16 plugin Datatables.net, 227-228 Drone, 35 Easypy, 179 Webex Team Notification, 487-488 POC (proof of concept), 169 polling, expected state, 332–333 POST API, 356-357, 361-362, 367-368, 370-371, 375-376, 379, 381 POST command, 384 post-change snapshot, 326-327 Postman, 29 power cyclers, 413–414 Clean-supported, 414-422 connected to a testbed, 422-423 device recovery, 435-436 pre-change snapshot, 324-326 predictive analysis, 352 Preston, Hank, 13 principles Agile, 12–13, 170–171 Lean, 12 Python, 30 processes child, 399 parallelism, 400 transferring data between, 399 processor/s, 91

context, 93 definition and arguments, 92–93 global, 93-94 results, 94 types, 91-92 profiling, network, 321-322 programming language, "batteriesincluded", 29-30 prompt handling, 478 properties, test parameter, 96 prototyping with Streamlit.io, 621-622 provisioning endpoint, 9 service. 9 pseudocode, 174 PTS (profile the system), 285–286 public Internet access, testing on a device. 266 PUT API, 358-359, 362-363, 368, 376, 379-380, 383 PUT command, 384–385 pyATS (Python Automated Test Systems), 1, 174–175 API/s, 347-351. See also API/s (application programming interface/s) get software version(), 351-352 network management, 352–353 Blitz. 459. See also Blitz Clean, 10-11. See also Clean clean stage template, 443–445 device recovery, 434–436 docstrings, 452-454 file validation, 436–437 including a clean stage in testscripts, 440-443

installing, 411 Integrated execution, 437–440 logging, 438-439 software image management, 431-433 Standalone execution, 437–438 supported OS/platforms, 413 supported power cyclers, 414-422 YAML, 423-424 command line, 41 core framework, installing, 38-39 data structures, 182-183 day 0 activities, 5-6 initial configuration, 6-7 *initial testing and validation*, 7 Layer 1, 6 day 1 activities, 7-8 incremental configuration, 8-9 provisioning new endpoints, 9 provisioning new services, 9 day N activities, 9 decommissioning, 11 monitoring, 9-10 responding to events, 10 upgrading, 10–11 day-1 activities, 4-5 offline initial configuration, 5 software image management, 5 device configuration, 309 embedded file transfer server, 311-313 FileUtils module, 309-311 Diff library, 162–167 Docker container building an image from scratch, 510 - 513

customizing, 506-507 image, downloading, 506 shell mode, 506 files, 190 framework, 175-176 AEtest, 176–179 Easypy, 179-180 Kleenex Clean, 181 testbed and topology, 180-181 generating configurations, 304 data modeling and validation, 304-305 data templates, 305–307 Genie Conf objects, 308–309 Health Check, 517. See also health checks CLI arguments, 524-525 core file, 516 CPU and memory, 515-516 logging, 516 results. 522-523 testcase/section selection, 520-522 usage, 523 HTML logs viewer, 203–205 Image Builder, 507–509 installing, 38-41 integrating with Webex, 486 health checks, 489-490 iobs, 487-489 integration with NetDevOps, 14–15 job/s, 29-30, 179, 190-196 arguments, 197–199 CLI logs, 199-202 ios xe version job.pv, 236 running from the CLI, 196–199 keywords, 582-584

learn command. See learn command learn models, 139–140 log viewer, 438-439 logging, 184 Manifest, 187 metaparser. See metaparser NETCONF client, 386 parse command, 157–158 Parser Filter, 146–148 pcall, 182 Python shell, 159–160 Reporter, 185–186 **REST Connector**, 354 installing, 354 specifying for a device, 354 result objects, 185 Robot Framework, 186 running as a container, 503–504 Tcl module, 183-184 testbed, 16, 51. See also testbed benefits, 51 building, 53–54 device connection abstractions. 60-63 dynamic, 66-68 extended, 68–70 external sources of truth, 56-60 interface object attribute list, 59 keys and values, 53-54 link object attribute list, 59 *object*, *52–53* object attribute list, 57-58 single device, 54 topology module, 51–52 validation, 63-65 troubleshooting, 45–46

upgrading, 42–44 utilities, 186 YANG Connector, 385–386 pyats create testbed command, 56 pyats logs view command, 201–202 pyats run job command, 115 pyats version check command, 42 pyats version update command, 43-44, 45-46 Pydantic, 304–305 pytest, 73, 74 Python, 2, 29-30. See also commands, Python Cerebrus, 304–305 Debugger, 133–134 Django, 67 Dq (dictionary query), 160–162 packages, 29–30 parsing, 159-160 pip, 30 principles, 30 Rich library, 234 SDK (software development kit), 31 settting up a virtual environment, 38 virtual environment, 31, 60-61

Q-R

queries
Dictionary, 160–162
XPath, 394–395
race conditions, 399
RAG (retrieval augmented generation), 592, 612, 612–614
embeddings, 593, 595, 621
JSON loader, 619–620
in network automation, 593

running configuration, questions and responses, 614-618 sourcing JSON or raw text data, 614 textual segment, 593–594 vector stores, 594–596, 614 randomization, testcase, 123-124 range keyword, 472–473 reachability testing, 262-267 reconfiguration, 8 recording jobs, 337–341 replaying recorded jobs, 341–342 use cases, 337–339 RegEx (regular expressions), 24 RegEx filter, 461–462 regression testing, 73 replaying recorded jobs, 341–342 reporting AEtest Reporter, 127–133, 185–186, 400 results.yaml file, 129–132 Standalone Reporter, 126–127 representer classes, 655 reserved parameters, 98 **REST** (Representational State Transfer), 27–28 DELETE API, 357–358 GET API, 355–356 PATCH API, 358 POST API, 356–357 PUT API, 358–359 REST Connector, 253–258, 354 BIG-IP. 373–374 Connect API, 374-375 DELETE API, 376–377 Disconnect API, 375 **GET API. 375** PATCH API, 376

POST API, 375–376 PUT API, 376 Catalyst Center, 363-364 Connect API, 364–365 Disconnect API. 365 GET API, 365 Cisco ACI APIC, 369-370 DELETE API, 371 GET API, 370 POST API, 370-371 DCNM, 380 DELETE API, 382 *GET API*, 381 PATCH API, 382 POST API, 381 **PUT API, 383** installing, 354 **IOS XE, 366** Connect API, 366-367 DELETE API. 368 GET API, 367 PATCH API, 368 POST API, 367-368 PUT API, 368 Nexus Dashboard, 383 DELETE command, 385 GET command, 383-384 POST command, 384 PUT command, 384-385 NSO, 359-360 Connect API, 360-361 DELETE API, 363 **GET API, 361** PATCH API, 362 POST API. 361–362 PUT API, 362-363

NXOS. 355 DELETE API, 357-358 GET API, 355-356 PATCH API, 358 POST API. 356-357 PUT API, 358-359 SD-WAN vManage, 377-378 Connect API, 378 DELETE API, 380 **GET API. 379** POST API. 379 PUT API, 379-380 specifying for a device, 354 RESTCONF, 56, 252-258 RESTful verbs. 353–354 result objects, 88-89, 185 result rollup, 90-91 results.yaml file, 129-132 Rich library, 234 risk, mitigation, 20 Robot Framework, 186, 333-335, 575-576 Easypy integration, 588–590 integration with pyATS, 582 keywords, 577-578 Genie. 585-588 pyATS, 582-584 Unicon, 584-585 test cases, 576-577 test execution, 580-581 test libraries, 580 variables, 578-579 rollback, 17 routing table AI analysis, 606–609 creating an application to chat with, 623-631

RPC (remote procedure call), 26 run condition, 475–476 run_condition keyword, 475–476 runner Drone, 35 GitHub Actions, 34 running a testscript, 108 runtime behavior, AEtest, 85–86 parent, 87 section ordering, 88 self keyword, 86–87 runtime.groups variable, 121

S

saving command output to a variable, 463-464 scalar variable syntax, 578–579 scaling performance, 397 schema clean stage, 445-447 creating, 635-636 script arguments, 96–97 Dockerfile, 623 Python, parsing, 159–160 startup.sh, 625 termination on failure, 477–478 ScriptDiscovery class, 125–126 SDK (software development kit), 31, 592 SDN (software-defined networking), 355 SD-WAN vManage, 377–378 Connect API, 378 DELETE API. 380 **GET API. 379**

POST API. 379 PUT API, 379-380 secret strings, 651 multiple representers, 653–655 representer classes, 655 securing, 651–653 section argument, 98 securing secret strings, 651-653 self keyword, 86–87 semantic analysis, 594 semantic search, 595, 596 service, provisioning, 9 setup section, testcase, 77 show commands, 189 learn model, 139–140 code. 142 *structure*, 140–142 parsing, 322–323 show ip interface brief command, 25, 26, 191, 207, 210, 214-215, 625-626 show running-config command, 285, 319.431 show version command, 236–243 show version function, 406–407 simplicity, 170–171 skip conditions, testscript, 117–119 SLA (service-level agreement), 2–3 snapshot, 322 data exclusions, 331, 332 custom, 332 default, 331–332 differentials, 327 post-change, 326–327 pre-change, 324–326 **SNMP** (Simple Network Management Protocol), 9–10

SOA (service-oriented architecture), 27 SOAP (Simple Object Access Protocol), 27 software -defined networking, 355 image management, 5, 10-11, 431-433 linting, 63–65 version testing, 235–243 software development containerization, 503 extreme programming, 170 lifecycle, 234 methodology, 11, 17 Agile, 12–13, 18 DevOps, 13 Lean, 12 TDD (test-driven development), 169 waterfall, 11–12 scaling performance, 397 source control, 16, 18, 19, 23 Sparks, Geoffrey, "Database Modeling in UML", 217 speed, NetDevOps, 20 SSH (Secure Shell) CLI options, 54–55 versus RESTCONF, 252-253 role in testing, 244–251 standalone execution, testscript, 109-114 Standalone execution, pyATS Clean, 437-438 Standalone Reporter, 126–127 standard arguments, 108-109 state, network, 324 known-good, 324

polling, 332–333 static analysis, 64 step/s, 79-80 attributes, 80-81 clean stage, 447-450 Drone, 35 GitHub Actions, 34 nested, 81–82 Steps class, 80 Steps object, 80 Streamlit.io, 621–622 creating an application to query a routing table, 623-631 integration with LangChain, 622-623 structure AEtest report, 128–129 learn model, 140–142 Ops, 323 testscript, 75, 78-79 structured data, 24. See also JSON (JavaScript Object Notation); YAML: YANG JSON (JavaScript Object Notation), 24 - 25XML (eXtensible Markup Language), 25 - 26YAML. 26 YANG, 26 subnet, 3-4 subsections, Common Setup, 76 syslog, alarm, 9-10

tags HTML, 225 Jinja2 template, 205

XML. 25 targets, 402-404 TaskLog, 179 Tcl module, 183–184 TDD (test-driven development), 14, 16, 49, 169, 234 applying to network automation, 172 - 174in brownfield environments, 173-174 rules, 171 three-step approach, 171–172 workflow, 172–173 template, 305-307 Jinja2, 6–7, 285 Adaptive Card, 492–501 arguments, 306 configuration file, 305–307 context, 306 CSV file, 209 HTML. 226 markdown table, 211 markmap mind map, 213 Mermaid class diagram, 218-219 Mermaid entity relationship diagram, 222–223 Mermaid flowchart, 216 Mermaid mind map, 224–225 Mermaid state diagram, 220-221 tags, 205 variables, 305 stage, 443-445 terminal server, device recovery, 435-436 terminal width 511 command, 55 test case, 576-577. See also Robot Framework

execution, 580-581 results and reporting, 581–582 test libraries, 580 test parameter, 94–95 parametrization, 98 properties, 96 relationships, 95–96 reserved. 98 types, 96–98 test results arguments, 90 interaction results, 90 result behavior. 89–90 result objects, 88-89, 185 result rollup, 90–91 testable, 117 testbed, 49, 51, 651. See also device arguments, 306 benefits. 51 building, 53–54 Cisco DevNet Always-On IOS-XE Sandbox, 235 creating, 66-67 device connection abstractions. 60-63 dynamic, 66-68 extended, 68–70 external sources of truth, 56-60 file. 5 intent testbed.yaml, 267 keys and values, 53-54 mock, 60–61 object, 52–53 attributes, 57–58 interface, 59 link. 59 variables, 61

with power cycler, 422–423 single device, 54 topology module, 51–52 validation, 63-65 -wide configuration, 314-315 testbed.connect() function, 196 testcase, 77-78, 178-179, 253. See also Blitz cleanup section, 77 container, 77 discovery, 125–126 maximum failures, 124–125 must-pass, 121–123 randomization, 123–124 script termination on failure, 477-478 self, 86–87 setup section, 77 test section, 77 **UID. 85** testing, 10, 14. See also TDD (testdriven development) arguments, 108 code, 73 feature, 271–276 for input errors, 248–251 intent validation, 267-271 interface, 243-244 interfaces for CRC errors, 177-178 linting, 16 logic, 86 negative, 477 neighbor relationshiops, 259-262 network, 234–243 pause on phrase, 133–134 reachability, 262-267 regression, 73

SSH role in, 244–251 unit, 172, 173 testscript, 78-79, 177 Common Cleanup, 78–79 datafile input, 98–104 debugging, 133-134 device cleaning, 438–443 Easypy execution, 114–117 flow control, 117 must-pass testcases, 121–123 running specific testcases, 119 - 120skip conditions, 117–119 testcase grouping, 120–121 ios xe interface.py, 245 ios xe version job.py, 236-243 learn all.py, 340–341 running, 108 standalone execution, 109–114 structure, 75 triggers and verifications, 299–301 TestScript class, 82 text editor, 22–23 textual segment, 593-594 threading, 398–399 tool/s. 352–353 AL 591 automation agent-based, 21–22 agentless, 22 cURL. 28–29 element managers, 2 GitLab CI/CD, 33–34 image builder, 505 Image Builder, 507–509, 543–545 **Jenkins**. 33

LangChain, 592–593 Mermaid. 215–217 network monitoring, 561 Postman, 29 topology module, 51-52, 180-181 topology YAML configuration, 386-389 Torvalds, Linus, 23 TPS (Toyota Production System), 12 traditional network, automation, 2 training, role of pyATS job recordings, 338 trigger, 290-291, 296-297 cluster, 292–293 datafile. 291-292 Drone, 35 job file, 283 in pyATS testscript, 299–301 writing, 293-296 troubleshooting, 330 pyATS, 45-46 role of job recordings, 338 XPRESSO, 534-536

U

UID, testcase, 85 UML, class diagram, 217–219 Unicon, 61–62 configure service, 315–316 copy service, 312–313 creating mock devices, 343–344 keywords, 584–585 unified diff format, 327–330 unit testing, 173 unittest, 73, 74 unstructured data, parsing, 137 untracked configuration, 303 upgrading, 10–11, 42–44 use cases job recording, 337–339 mock device, 342–343 user keywords, 578 utilities, 186. *See also* tool/s

V

validation, 304-305 Clean file, 436–437 configuration, 7 intent. 267-271 testbed, 63–65 value keyword, 472 variables, 16 action output, 463-465 Robot Framework, 578–579 runtime.groups, 121 template, 305 testbed object, 61 testscript-level, 464-465 version threshold, 241 vector data, 594 cosine similarity, 595–596 geospatial analysis, 595 vector stores, 594-596, 614, 621 velocity, 20 vendor-agnostic automation, 138-139 venv module, 31, 38 verification/s, 296-297 action output, 467-470 datafile, 288

job file, 283 in pyATS testscript, 299–301 types, 287 writing, 288–290 version control, 16, 18, 19, 23, 23 version_threshold variable, 241 virtual environment, creating, 60–61 VM (virtual machine), 31, 504 VXLAN (Virtual eXtensible LAN), 3–4

W

waterfall methodology, 11-12, 17 Webex, 485 Adaptive Cards, 490–492, 500–501 alert, 10 integrating with pyATS, 486 health checks, 489–490 iobs, 487–489 sending a notification when a health check fails, 525 Webex Team Notification plugin, 487-488 webhooks, 571 WebInteraction class, 90 while loop, 473 Womack, James, 12 workflow GitHub Actions, 34 TDD (test-driven development), 172 - 173writing triggers, 293-296 verifications, 288-290 WSL (Windows Subsystem for Linux), 37

<u>X</u>

XML (eXtensible Markup Language), 25 - 26XPath queries, 394–395 XPRESSO, 172 common issues, questions and answers. 534 cannot connect to database, 535 cannot log in using default admin. 535 *ElasticSearch failed to start,* 535 Error: No Resources Found, 535 general networking issues with the installation, 535-536 references to S3, 535 unhealthy services, 535 features and benefits, 527–529 getting started facilitating quick adoption, 536-538 transforming a pyATS job into XPRESSO, 538-556 installing, 529-534

Y

YAML/YAML files, 5, 14, 26, 181, 306–307. *See also* topology YAML configuration applications, 51 Blitz, 459, 460–461 business-ready documents, 207–208 characteristics, 49–50 Clean, 423–424 *clean stages, 424–427* device groups, 428–431 devices block, 424 creating, 56–57 datafiles, 98–104 health check, 517–520 testbed, 49. See also testbed building, 53–54 Cisco DevNet Always-On IOS-XE Sandbox, 235 dynamic, 66–68 extended, 68–70 external sources of truth, 56–60 interface object attribute list, 59 keys and values, 53–54 object attribute list, 57–58 validation, 63–65 YANG, 26 YANG Connector, 385–386 Yuan, Siming, 1

Ζ

"The Zen of Python", 30 .zip folder, 180 ZTP (zero-touch provisioning), 6