## Assignment: Software Test Plan

### SSDynamics



Apr 4, 2025

Carter K. Chas D. Connor A. Charles D. Savannah Chappus

**Chris Ortiz** Senior Technologist, SSD Validation
 Western Digital Corp., Flash Business Unit

**John Lee**

Senior Director, SSD Validation

Western Digital Corp., Flash Business Unit

#### Table of Contents

[**Assignment: Software Test Plan**](#_mb00m7dt2z0o)

[Introduction](#_7utsxqfdr8s9) 2

[Unit Testing](#_xy0u60beepsf) 3

[Integration Testing](#_u7lhlx1ca6o8) 5

[Usability/End-user Testing](#_k6kk7vlpkog1) 10

[Conclusion](#_b074golya45r) 12

#### Introduction

The NVMe Generative Testing platform is a specialized tool designed for automating the process of testing NVMe storage devices. This application leverages TLA+ specifications, Python modules, and NVMe CLI commands to enable efficient and effective testing for engineers working with NVMe devices. By automating much of the testing workflow, the platform aims to streamline processes and ensure that NVMe devices are tested thoroughly and reliably. The primary goal is to provide a tool that automates the creation and execution of tests, making it easier for engineers to verify the performance and integrity of their devices in a standardized, reproducible way.

Software testing plays a critical role in the development of any application, ensuring that the software works as intended, is free from errors, and meets user requirements. Testing helps identify potential issues early in the development process, improving overall product quality and user satisfaction. For this application, the testing process will cover various levels, from unit testing of individual components to integration testing of the entire system. Additionally, usability testing will be conducted to ensure that the tool is user-friendly and meets the needs of engineers in a real-world setting. The goal is to guarantee the platform performs well under different scenarios and provides a smooth user experience.

The testing approach for this application will involve a comprehensive strategy that includes several types of testing. Unit testing will focus on individual modules within the platform, ensuring that each component behaves as expected. Integration testing will verify that the modules work together seamlessly, with particular attention to the communication between Python, TLA+, and NVMe CLI components. Finally, usability testing will ensure that the tool is intuitive for the target audience, experienced testing engineers. We will also incorporate expert reviews, where real-world users will assess the tool's functionality and provide valuable feedback to enhance its usability. Each type of testing will be detailed in the following sections, and the rationale behind the selected testing strategies will be discussed based on the specific needs of the application.

Given the specialized nature of this application, the testing protocol focuses more intensively on ensuring the correctness and reliability of the platform’s core functionalities—particularly the interaction between the TLA+ specifications and the NVMe CLI commands. Because the application is designed to automate a process that would otherwise be complex and error-prone, it is essential that these interactions are thoroughly tested to prevent issues during real-world use. The testing plan also prioritizes usability, ensuring that engineers can easily incorporate the tool into their existing workflows with minimal disruption.

The subsequent sections will provide a detailed description of the testing activities for each of these areas, starting with unit testing, followed by integration testing, and concluding with usability testing and expert reviews.

#### Unit Testing

Unit testing is an important part of testing software, and this generative testing platform is no exception. Unit testing for our testing platform is a little bit weird, as it is essentially ‘testing the tester’ to make sure that it performs as expected. The unit tests that we are planning to have will test every component of the platform. Using the unittest Python library, the plan is to test each Python file that has been written, and make sure that each chunk is getting the right output. Using unittest for our NVMe Generative Testing platform is the best solution to testing our product, because it is built into Python, and we can make sure that the solution will work in base Python without grabbing any external dependencies except for PlusPy/TLA+. The unittest Python library is quite helpful, allowing for wide test coverage that will allow us to test the majority of our application. Since our application uses a mixture of TLA+, Python, and NVMe-CLI, we need to make sure that the testing covers a large portion of the program. This is a little difficult, since we can technically only check to see if the TLA+ is being parsed correctly after it enters the PlusPy part of the tester. This means that our only way to really check to see if the TLA+ is working is by verifying that the PlusPy library is reading a TLA+ model correctly. Our Python modules that we are writing to ‘translate’ the model into NVMe commands are all going to be tested with the unittest library. This is where the majority of the tests will be written for, since the bulk of the work is turning those models into commands. Lastly, the NVMe-CLI is written in C/C++, so we only need to test that the return from the CLI command did not fail, which should be easy as long as the message is parsed correctly.

Since the focus is on testing the code that we have written (or modified from the PlusPy library), the bulk of our modules/files are where the unit tests are focused. We have had to modify the PlusPy library to be able to fix some ‘quirks’ for the rest of the code to properly work as we intended it to. This has been tested manually, and will continue to do so unless there are far more changes made to the library. The program is currently modular, and is going to be tested separately. This is because of the loose coupling between each Python file, and each one of the modules is there to provide a necessary function to the testing platform as a whole. The intention is to make sure that each module is providing the right details so that if the module is changed in some form or fashion, it will still provide the correct output for the next module that picks the data up.

The ‘Interface.py’ Python module is the ‘terminal application parser.’ Really it’s how the interface is displayed to the user/programmer/tester. This has been written to reflect UNIX style terminal applications, and to conform to a somewhat standard. The main tests here are to make sure that each flag and their combinations are being parsed correctly. We use the argparse library here to grab the user input when the terminal command is invoked.

The ‘Dispatcher.py’ Python module is the communication between the Python code and the NVMe CLI. This is what handles the TLA+ model translation into the corresponding NVMe commands that are within the specification. The test cases here are to make sure that parsing of TLA+ models are working here before being turned into NVMe commands. This is through a standard interface that is defined by the NVMe specification. This includes all of the relevant values or drives and namespaces that need to be passed through to make sure that the program works.

The ‘Executor.py’ Python module is the actual interface between the pre-parsed TLA+ and the NVMe CLI and its return values. This module is really there to execute the NVMe commands from the TLA+ model. It does two things, either run an admin\_passthrough or an IO\_passthrough command. This is how to work with the drive at the lowest level possible without actually writing data to the device in Linux. The switch between admin\_passthrough and IO\_passthrough commands happens at the TLA+ model, when a test is being written, each ‘state’ or command that will be executed on the drive will be labeled as an admin or IO command. The main tests here will make sure that the command that is input matches the output command from the NVMe CLI.

The ‘Logger\_config.py’ Python module is everything related to logging output from the NVMe CLI. This is highly configurable, and the defaults are great for most situations. These tests will focus on making sure that the output files are created, that they have the necessary information that the logging should be spitting out, and that any errors are reported at the end of the file for easier review when the test has concluded.

The ‘Tester.py’ Python module is the environment that kinda puts this all together. It’s the linked modules working together. The tests that are here focus on making sure that the environment is stable, and that the rest of the modules are giving the environment proper data to work with.

There are a few ‘modules’ that are really just helper utilities and aren’t necessarily meant to be exposed to the user/programmer/tester. These don’t have unit tests, since they are meant to be utilities for the main script.

Each of these modules are the building blocks of the nvme\_gentest file, which is the file that actually interacts with the user/programmer/tester. This is currently being tested manually, as a whole. It’s really the part of where you’ll know if anything is working or not, as it functions as a terminal application. Writing unit tests to take some of the burden off of manual testing is the hope.

#### Integration Testing

Integration testing ensures the major components of our NVMe Generative Testing platform interact seamlessly, exchanging data correctly across various layers. Unlike unit testing, which focuses on individual functions or modules in isolation, integration testing examines how well the modules work together, verifying that outputs from one module serve as correct inputs for the next. It also ensures that contract assumptions across different module boundaries are adhered to. Given that our tool includes components developed in different languages (TLA+ and Python), integration testing becomes even more crucial to guarantee that data translation and communication between the layers are consistent.

The primary objectives of our integration testing efforts are as follows:

* To verify that TLA+ models are correctly parsed and interpreted by PlusPy.
* To ensure that Python modules accurately transform interpreted models into NVMe CLI commands.
* To validate that command execution through the NVMe CLI returns expected values, and those values are correctly interpreted by our application.

We have adopted a bottom-up integration strategy for testing, prioritizing the validation of core functionality (such as NVMe CLI command execution) before testing the higher-level integrations with the Tester.py and nvme\_gentest. This approach helps ensure stable lower-level integrations first, which are the foundation for more complex system behavior.The testing approach includes validating key integration points such as:

* TLA+ to Dispatcher Parsing
* Dispatcher to NVMe Command Execution
* Seed Initialization and PlusPy Integration

Through this process, we will ensure that all components interact effectively and that the system behaves as expected under different scenarios, including bug detection and state transitions.

Integration Points and Testing Strategy

1. TLA+ -> Dispatcher.py

Objective: To ensure that parse\_tla() correctly extracts NVMe command data from TLA+

Test Strategy:

1. Provide well-formed TLA+ output containing NVMe command mappings.
2. Provide malformed or incomplete TLA+ outputs to test robustness.
3. Verify that extracted args, result, ssd, and pattern match expected values.

 Verification:

* Assert correct key-value extraction from TLA+ output
* Validate missing or malformed keys are handled gracefully
1. Dispatcher -> NVMe Command Execution

Objective: Ensure that execute\_parse() correctly formats and submits commands to NVMeExecutor.

Test Strategy:

1. Ensure that NVMe command arguments match expected formats.
2. Validate that execute\_parse() calls nvme\_passthru() with expected args.

 Verification:

* Ensure that NVMe command arguments match expected formats.
* Confirm correct handling of missing or extra arguments.
1. Dispatcher Error Handling & Edge Cases

Objective: Validate how dispatcher.py handles unexpected inputs.

Test Strategy:

1. Provide TLA+ outputs with missing fields
2. Introduce malformed formats

Verification:

* Check for appropriate logging of parsing errors.
* Ensure the system does not crash or execute invalid commands.
1. Initialization -> Seed -> Pluspy

Objective: Ensure that NVMeTester correctly initializes pluspy with proper constants and random seed

Test Strategy:

1. Mock PlusPy and ensure init() and getall() are called with expected inputs.
2. Verify generate\_seed() creates a reproducible seed when one is not provided.
3. Ensure constants are loaded, transformed into tuples, and include discovery data.

 Verification:

* Confirm PlusPy receives correct constants and seed.
* Log output includes seed and initial state.
* Validate discovery data (devInfo) exists in constants.
1. TLA+ State -> Dispatcher Parsing -> NVMe Execution

Objective: Confirm end-to-end pipeline:

TLA+ spec -> state generation -> parse -> CLI simulation -> result verification.

Testing Strategy:

1. Use mock or dry-run mode for NVMeExecutor.nvme\_passthru() to avoid hardware execution.
2. Provide known TLA+ outputs to PlusPy, then check if dispatcher parses and executes them correctly.
3. Test multiple seeds and confirm consistent behavior.

 Verification:

* Dispatcher receives and parses valid TLA+ states.
* Execution output is passed to \_verify\_result() and \_check\_all\_match().
* Result matches expectations when the hex pattern is included.
1. TLA+ State -> Dispatcher Parsing -> NVMe Execution

Objective: Ensure proper detection of bugs and that bug mode logic resamples seeds and retries gracefully.

Testing Strategy:

1. Simulate command mismatch by injecting incorrect stderr values.
2. Set bug\_max and retry\_max to low values to validate stop conditions.
3. Monitor transitions into and out of bug mode.

 Verification:

* Logs indicate when bug mode is activated.
* On hitting the retry limit, the test gracefully exits.
* Resampled seeds are used and counted correctly.

1. Pattern Matching -> Hex Dump Validation

Objective: Ensure \_check\_all\_match() correctly interprets and validates expected hex patterns.

Testing Strategy:

1. Provide known hex dumps and patterns to the function.
2. Test with correct and incorrect patterns, as well as with an empty pattern.

 Verification:

* Returns True when the pattern matches the hex output completely.
* Returns False when mismatched or pattern is malformed.
* Skips validation cleanly when the pattern is empty ("").
1. Graceful Termination & Logging

Objective: Ensure the tester terminates when state count or no-change condition is met.

Testing Strategy:

1. Set a low count limit.
2. Return unchanged state from PlusPy which comes from a specified tla+ file

 Verification:

* Logs reflect early termination with the correct reason.
* \_reset\_bug\_mode() is called upon exit.

#### Usability/End-user Testing

Usability testing is a method for determining how easy it is for the end user to use a product to accomplish a task. Usability testing is crucial for delivering a final product that our clients will want to use. Usability testing ensures that the users are satisfied with the state of the product in both functionality and usability. Because we are creating a CLI based tool it is even more important that we ensure it is easy to use. Because while a user can stumble around a bad GUI based project a bad CLI based project will leave the user smashing their keyboard. Usability testing helps by identifying key points of tension and issue with the product. Once these pain points have been identified they can be remedied. This can also be used to improve documentation to assist users in getting help using the product or future developers improve the product later down the road. With these improvements the workflow of the product can be further optimized to improve user productivity. This also eases users into the product by adhering to expected behavior. For this project an example might be displaying a help page when the app is run with the -h flag.

 The main goals of usability testing are to ensure the end product is intuitive for users, the workflow of using the product is unrestrictive and easy, and the documentation is clear, easy to understand and actionable. Users should be able to use the product with full confidence that they know what is going to happen without needing to know how the product works on the inside. With this being said, it shouldn’t be difficult to understand the lower level implementation. This ensures that the product can be improved in the future. In addition to this the testing process should ensure that the simulator notifies the user if an error occurred during execution and what caused that error. There is a lot that can go wrong with testing hardware. It is important that the end user is aware if the error is the result of a misconfigured TLA+ file or if it is a result of faulty hardware. Another key feature of our development is providing detailed, yet easy to understand documentation that can also be easily acted on. The main piece of this is that every part of the system is documented thoroughly. Though almost just as important as through documentation is how easy it is to understand and take action on. There is no point to having pages and pages of instructions if they don’t make sense or are hard to follow. We will have key processes and options for the program documented in a short, detailed, easy to understand, and standardized way. Such as bringing up a help menu when the -h flag is used after the program.

In order to accomplish these goals we will conduct expert reviews. Because of the nature of our product it was not made for most people and therefore they would not be able to use it regardless. With this in mind we will provide the minimum value product to our clients to do their internal testing. This allows them to use the product with real world hardware and data, testing it as how it will be used. This will be our only method for gathering feedback on the product as it will be an internal tool for our client and to do other testing we would have to find competitors and show them the product. This goes against the new and cutting edge product testing that this system will provide.

 Some key considerations for our project include the CLI format, the target user audience being experienced testing engineers looking for a tool to speed up their current workflow. Even though our product is based on the command line and thus no user interface to optimize, CLI based applications need to be thoroughly documented. This is because a blinking cursor is all the hint a user gets without struggling through the interface. Without a help page the user will, at best, stumble through the program and spend way too much time trying to figure out how it works instead of using it to speed up their workflow. The novelty of the product lies in its use case. It will only be used by testing engineers working with NVMe storage devices. These testers already know the workflow for testing a device and simply want a way to automate most of the process.

Our expert reviews will be conducted over several weeks. Our current schedule is already tight and thus testing might get cut short. However we plan to conduct testing with 3-5 experts at Western Digital. These testers will use the tool in their normal testing workflow. This will be conducted over a week to ensure testers have thorough exposure to the product and can take. We will also have them test with working and non-working devices as well as incorrect input to the system to clearly show the difference between a user error and a hardware error. Notes will be taken by the testers as well as a quick questionnaire to help gather both quantitative and qualitative data. After testing has completed we will spend 3 days reviewing and analyzing the data. We will categorize the qualitative data to find recurring issues and rate them on a severity scale of 1-5. Higher severity, more common problems will be addressed first. The changes based on these reviews will take place over ideally one week however we have up to 2 weeks to apply them with testing to ensure that the program functions as expected. With all this included our final product will be user friendly and improve the end user experience.

#### Conclusion

The NVMe Generative Testing platform offers a reliable and efficient way to automate testing for NVMe storage devices. By combining Python scripts, TLA+ models, and NVMe CLI commands, it streamlines the testing process and reduces the chances of human error. This makes it a practical solution for engineers looking to validate device behavior with greater speed and consistency.

Through careful unit testing, we confirmed that individual components of the system perform as expected. Integration testing showed that the various parts of the platform, such as the Python backend, TLA+ modeling, and NVMe command-line interactions, work smoothly together. Usability testing ensured the platform remains accessible and intuitive for engineers, with a command-line interface that fits into their daily workflow.

Together, these testing efforts demonstrate the platform’s strength in functionality, reliability, and user experience. With this solid foundation, the NVMe Generative Testing platform is well-prepared to support the growing demands of storage device validation.