ReLiANT

Software Testing Plan

March 20, 2026

For

SanDisk

Faculty Mentor: Ogonna Eli
Team Members: Xavier Graham, Fermin Valenzuela, Brody England, Isaac Schwarz

Version 1.0

Table Of Contents

Introduction

ConTiNGENT, which stands for Convert TLA+ into NVMe Generative Test, is a capstone project sponsored by SanDisk that aims to reimplement and improve a prior proof-of-concept for specification-driven NVMe testing. The original effort, developed during an earlier collaboration with SSDynamics under Western Digital, explored the idea of generating test sequences directly from a TLA+ specification rather than relying only on hand-written tests created by test engineers. That earlier project demonstrated the promise of model-driven test generation, including improved bug discovery, broader test coverage through nondeterministic execution, better replayability than fuzzing, and reduced wasted device-under-test time. However, parts of the proof-of-concept were unreliable, and this was later traced to the use of PlusPy, a Python-based TLA+ interpreter. SanDisk's new direction for this project is to rebuild the idea from scratch in a more robust, scalable, and production-aligned way using TLC, Rust, and a clearer instrumentation strategy between model state and physical NVMe device state.

The purpose of ConTiNGENT is to create a framework that connects a formal TLA+ specification of NVMe-related behavior to real executable test actions on SSD hardware. In the current project, the system includes a TLA+ model (SampleSpec), an execution-oriented TLA+ module (GenIOExec) that maps abstract actions to concrete command execution, and a Rust passthrough tool that invokes nvme-cli commands and normalizes their outputs into a format the model can reason about. The intended users are technical users: test engineers, validation engineers, researchers, and advanced developers working with NVMe SSD behavior and formal methods. The main quality goals of the system are correctness, repeatability, reliability, traceability, and safe interaction with hardware.

In this project, software testing is broader than ordinary application testing. The product is not just a command-line tool or a standalone codebase. It is a verification pipeline with multiple layers: a formal specification layer, an executable orchestration layer, a Rust systems layer, an external command-line dependency layer, and a real hardware execution layer. Major components include the TLA+ state machine, the TLC-driven generation logic, the Rust passthrough binary, the nvme-cli toolchain, device payload files, and the NVMe device under test. External dependencies include TLC, OpenJDK, Linux, sudo, Ubuntu-compatible nvme-cli, and compliant NVMe hardware. The test environment also includes the required extensions and development tooling described in the proposal, such as the TLA+ VSCode extension, Rust-analyzer, and a Linux system with a secondary NVMe SSD.

The scope of testing includes the correctness of state transitions in the TLA+ model, the validity of invariants, the accuracy of parsing and transformation logic in the Rust passthrough utility, the integrity of command construction, the correctness of interaction between model and hardware, and the usability of the workflow for technical users. The project will also test repeatability of generated sequences and the handling of device state changes such as identify, write, read, format, and restore. Out of scope are the correctness of third-party tools themselves, the correctness of the NVMe specification authored by outside bodies, firmware internals inside the SSD, hardware defects unrelated to software operation, and operating systems or environments outside the supported Linux-based setup.

The overall testing strategy for ConTiNGENT combines unit testing, integration testing, and usability testing, all of which are required by the assignment, along with additional project-specific testing categories that are especially important in a formal verification context. Unit testing will validate isolated Rust logic such as parsing, bit extraction, payload preparation, and helper behavior. Integration testing will focus heavily on the boundaries between the TLA+ model, TLC execution, Rust command orchestration, nvme-cli, and the physical NVMe device. Usability testing will evaluate whether the intended technical users can successfully understand, run, and interpret the toolchain. In addition, this project requires formal model and invariant validation because the system's core claim depends on the model remaining sound and meaningful. Reliability and performance testing will also be included because repeated execution on real devices must be stable enough for practical use within a sponsor's test process.

This strategy is appropriate because not all parts of the system carry the same risk. The highest-risk areas are the translation points between abstract model state and physical hardware behavior, since a defect there could make the framework report false agreement or false disagreement between the model and the device. Parsing errors, command construction errors, and state restoration errors also present significant risk because they can invalidate test evidence or leave the DUT in an unexpected state. Usability matters as well because even a technically correct framework will fail in practice if engineers cannot run it confidently or interpret its outputs. The sections that follow explain in detail how the team will test each of these areas.

Unit Testing

Unit testing in ConTiNGENT is used to verify the correctness of individual software units, especially within the Rust passthrough utility, before those units are combined with TLC, nvme-cli, and actual NVMe hardware. In this project, unit testing means checking that small functions, parsers, helper routines, and data transformation logic behave correctly in isolation. This matters because the passthrough layer acts as a bridge between raw system output and the formal model. If this layer contains defects, the system may incorrectly interpret hardware state even when the device itself is behaving correctly.

The main goals of unit testing are to verify parsing logic, validate helper computations, confirm command preparation behavior, and ensure that edge cases are handled safely. The project will use Rust's built-in testing framework through cargo test. These tests will be run locally during development and as part of continuous integration whenever possible. The team will track pass/fail status for all unit test suites and will pay special attention to critical-module coverage rather than relying only on a raw overall percentage. In this project, testing the highest-risk logic is more important than maximizing simple coverage metrics.

The most important units under test are the Rust helper functions and parsers shown in the passthrough code. These include get_fna_bit, get_fnvms_bit, parse_hex_u64, maybe_pad_write_payload, print_ok_passthru_stderr, parse_id_ctrl, parse_id_ns, and the internal logic that supports run_read_ok_csv_first_byte_per_lba. These functions are logic-heavy and directly affect whether the framework produces correct modeled values from command outputs. Additional units under test may include argument handling in main, command routing in run_cli, and any future helper functions

extracted for cleaner testability. The team will not attempt to unit test third-party command execution itself, direct sudo behavior, or real device interactions in isolation, because those belong to integration testing rather than unit testing.

The team's unit test design will include valid inputs, boundary conditions, and invalid inputs. For example, get_fna_bit and get_fnvms_bit will be tested with known hexadecimal controller values to verify that the correct bits are extracted. Boundary conditions will include minimum and maximum expected bit positions or unusual but valid hexadecimal strings. Invalid cases will include malformed hex values and missing prefixes. For parse_hex_u64, tests will include hex strings with and without 0x, decimal strings, zero values, and malformed input that should safely fall back or fail in a predictable way. For maybe_pad_write_payload, tests will verify that when the file is smaller than the requested -z size, the content is repeated to the exact required length; when the file already meets or exceeds the requested size, no unnecessary temp file is created; and when the file is empty or missing, the function fails safely. For read normalization logic, tests will verify that the expected number of first-byte values are emitted per LBA and that non-success command results are represented as "0".

A sample unit-testing breakdown for this project is shown below.

Unit Under Test: parse_id_ctrl(output)
Purpose: Extract controller capability bits from nvme id-ctrl output and print normalized CSV output.

Test Case Categories:
Valid inputs: correctly formatted oacs and fna lines
Boundary cases: additional unrelated fields, whitespace variations, lowercase/uppercase hex
Invalid inputs: missing required fields, malformed hex strings, empty output

Sample Tests:

Output containing valid oacs and fna values returns the expected fnvms, fna bit 0, and fna bit 3 values
Output with extra irrelevant lines still parses correctly
Missing fna line causes the function to fail in a controlled and detectable way

Unit Under Test: maybe_pad_write_payload(args)
Purpose: Ensure write payload file matches the exact size expected by the NVMe write command.

Test Case Categories:
Valid inputs: file shorter than target size, file exactly target size, file longer than target size
Boundary cases: one-byte file repeated many times, exact 512-byte multiples
Invalid inputs: missing file, empty file, invalid -z value

Sample Tests:

A two-byte file with -z 8 produces an eight-byte temporary file with repeated content
A file already large enough is left unchanged

An empty file returns None and does not create a temporary payload

The rationale for this unit-testing scope is that ConTiNGENT's most error-prone isolated logic is concentrated in data normalization and helper functions rather than in large business-logic modules typical of web applications. Unit tests therefore focus where they provide the highest value: the small but critical logic that translates command output into trustworthy modeled behavior.

3. Integration Testing

Integration testing is one of the most important testing categories for ConTiNGENT because the project exists specifically to connect independently correct pieces into one working verification pipeline. In this project, integration testing means verifying that the TLA+ model, TLC execution, instrumentation conventions, Rust passthrough utility, nvme-cli, Linux execution environment, payload files, and the NVMe device work correctly when combined. Even if each part appears correct in isolation, defects can still appear at the boundaries between them through data mismatches, inconsistent assumptions, environment issues, or incorrect command translation.

Following Alpha I, the project should already have multiple connected components, which makes integration testing essential. The team's overall approach is to identify integration points where a failure would break either the correctness of test execution or the credibility of the formal-to-physical mapping. High-priority integration points include the mapping from TLA+ actions to concrete command invocations, the parsing of nvme-cli responses back into model-visible state, the consistency of write/read behavior between the abstract namespace model and the actual SSD, and the ability to restore or reformat device state safely between test runs.

The key integration points that must be verified are:

1. TLA+ model action to executable orchestration in GenIOExec
2. GenIOExec command generation to Rust passthrough argument handling
3. Rust passthrough to nvme-cli invocation
4. nvme-cli response output back into parsed model values
5. Write/read operations across the formal namespace state and the physical SSD namespace
6. Format and restore operations across saved pre-format metadata and real device configuration
7. Controller and namespace identify operations that initialize model knowledge from hardware state

Integration tests will be executed in a controlled Linux environment using Ubuntu, TLC/OpenJDK, Rust tooling, nvme-cli, and a secondary NVMe SSD configured specifically for testing. This environment

matches the equipment expectations described in the project proposal. Test execution should be stable and repeatable. Where possible, the team will use dedicated test namespaces, known payload files such as dataA.bin and dataB.bin, and stable test-device paths. Test data will be created ahead of time for repeatable runs, and any destructive commands such as format will be limited to the approved DUT environment. Cleanup procedures will include restoring known namespace parameters where possible and documenting the device state before and after tests. Because hardware testing introduces risk, the environment must be treated as controlled and not interchangeable with arbitrary developer machines.

The integration scenarios will be tied directly to project goals and user-visible workflows.

Integration Point: TLA+ Write Action → Rust Passthrough → nvme-cli Write → SSD State
Feature: Model-generated write operation

Scenario Description:
A generated write action from the TLA+ execution layer writes a known pattern to a selected namespace and range of LBAs.

Integration Steps:

1. GenIOExec chooses ns(namespace), s(start LBA), c(LBA count), and data(the data that will be written) randomly within the specified model bounds

2. The generated action calls the Rust passthrough with the corresponding --write arguments

3. The Rust tool pads the payload file if needed and invokes nvme write through sudo

4. The write returns a success or failure indicator

5. On success, the abstract namespace state is updated in the model to reflect the changes in the real device

Expected Results:

- The Rust layer builds the correct NVMe write command

- The payload size matches the requested block count

- A successful write updates model state consistently

- No out-of-range LBA access occurs

Failure Handling:

- Failed writes are reported clearly
- Model state is not falsely updated if the physical write fails
- Errors are preserved for debugging through stderr output

Integration Point: TLA+ Read Action → Rust Passthrough → nvme-cli Read → Output Comparison
Feature: Verification of read-back behavior

Scenario Description:
After one or more writes, a generated read operation retrieves blocks from the SSD and compares the observed bytes to the expected model state.

Integration Steps:

1. The model selects namespace, start LBA, and number of LBAs
2. GenIOExec issues the corresponding read command through the Rust passthrough
3. The passthrough normalizes the output into a simplified format: "1,<byte0>,<byte1>,..." or "0"
4. The TLA+ layer computes expected values using ExpectedSeq
5. Assertions compare actual and expected data

Expected Results:

- Successful reads return the expected number of per-LBA values
- Known written blocks match expected patterns
- Unknown or unwritten blocks are treated consistently with the model's allowances

Failure Handling:

- Shape mismatches trigger assertion failures
- Data mismatches are reported with enough information to reproduce the issue
- Command failures do not silently pass as valid reads

Integration Point: Identify Namespace → Parsed Metadata → Format/Restore Behavior
Feature: Preservation and restoration of namespace settings

Scenario Description:
The system captures namespace metadata such as DPS and LBA format values before format operations and later attempts to restore them.

Integration Steps:

1. The tool executes id-ns for a namespace
2. Parsed dps, nlbaf, and lbaf_in_use values are stored in both current and “before” variables
3. A format action changes namespace state
4. A restore action reconstructs command arguments from the saved metadata
5. The model updates to the restored state if the device operation succeeds

Expected Results:

- Initial identify values are parsed correctly
- Pre-format values are preserved
- Restore commands use the correct PI/PIL/LBAF values
- The model remains consistent with successful restoration

Failure Handling:

- Missing pre-format values block restore attempts
- Failed restore commands are surfaced clearly
- Inconsistent state after restore is treated as a test failure

These scenarios matter because they directly reflect the sponsor’s goal: converting TLA+ specifications into real, meaningful NVMe generative tests rather than just simulated model traces.

4. Usability Testing

Usability testing in ConTiNGENT focuses on whether the intended technical users can effectively understand, execute, and troubleshoot the framework. Since this project is not aimed at casual end users, usability does not primarily mean visual design or polished consumer UI. Instead, usability means that developers and test engineers can correctly set up the environment, run the toolchain, interpret outputs, and understand failures without unnecessary confusion. For this project, a system can be technically correct and still fail if it is too difficult for engineers to use reliably.

The main goals of usability testing are to identify confusing workflow steps, improve clarity of command usage, validate whether outputs are understandable, reduce setup and execution errors, and make the framework more practical for integration into a sponsor's real testing process. Because the project spans formal methods, command-line tooling, hardware interaction, and Linux setup, usability risks are nontrivial even for technical users.

The target users for this system are developers, capstone team members, mentors, and sponsor-side technical staff with some familiarity with command-line tools and software engineering concepts. However, not all of them will have the same background in TLA+, NVMe commands, TLC execution, or Rust-based tooling. This mixed technical profile shapes the usability strategy. A project like ConTiNGENT does not need a consumer-grade interface, but it does need documentation, command conventions, error messages, and workflows that are understandable enough for engineers to adopt and maintain. Poor usability here could lead to false assumptions, misconfigured environments, unsafe device usage, or underuse of the tool entirely.

The team will use practical, lightweight usability methods. These will include guided setup walkthroughs, task-based testing sessions, mentor/client feedback, and expert review by technically capable users who were not the primary implementers of a given feature. Planned sessions may include at least one mentor review, one or two peer developer walkthroughs, and one sponsor-facing or sponsor-informed workflow review if available. Representative tasks will include:

- Setting up the required environment and dependencies
- Running the Rust passthrough manually with sample arguments
- Running the TLA+ specification or generation workflow
- Interpreting a failed read or assertion mismatch
- Identifying where a generated test sequence came from
- Understanding how format/restore operations affect the DUT

- Following documentation to reproduce a generated hardware test

Measures will include task completion, time to complete setup or execution, number of clarification questions needed, observed confusion points, and qualitative feedback on command clarity, output formatting, and documentation quality. Because the sponsor explicitly requested detailed and professionally composed documentation as a deliverable, usability testing will also include whether technical users can successfully use the documentation as a reference manual.

Usability testing will be distributed through the development cycle rather than performed only once. Early usability review will focus on setup and command discoverability after Alpha I. Additional sessions before Alpha II will focus on whether generated tests and outputs are understandable. Final review before acceptance will focus on documentation, reproducibility, and how easily another technical user can operate the system with minimal verbal explanation. Findings will be logged as issues, prioritized based on user impact and likelihood of misuse, and addressed in upcoming iterations. This creates a practical feedback loop that supports adoption instead of treating usability as an afterthought.

5. Additional Testing: Formal Model and Invariant Validation

Because ConTiNGENT is built around a formal specification, model validation is a core testing activity, not just an optional enhancement. The project's credibility depends on whether the TLA+ model remains internally sound and whether its invariants still hold as the specification evolves. If the model is incorrect, then even perfectly functioning Rust and hardware execution layers would only automate the wrong behavior.

This testing category focuses on validating the TLA+ modules themselves, especially SampleSpec and GenIOExec. The primary property shown in the current specification is the invariant TypeInv, which states that namespace LBAs remain within valid bounds and that stored values belong to DataPattern. The team will use TLC to check that Spec => []TypeInv holds over the explored state space. As the specification matures, additional properties may be added to capture stronger correctness requirements, such as consistency after write/read operations, proper reset behavior after format, or restoration of preserved namespace metadata.

Formal-model testing will include:

- Running TLC model checking on SampleSpec

- Checking that the generated or instrumented model preserves required invariants
- Reviewing whether nondeterministic choices remain constrained to meaningful ranges
- Confirming that state updates in operations such as Write, Read, Format, and Restore match the intended semantics
- Ensuring that instrumentation added for executable generation does not accidentally weaken or distort the specification

This testing category is especially important because the proposal specifically emphasizes TLC's generate mode as an improvement over the earlier PlusPy-based approach. That means the correctness and trustworthiness of TLC-driven behavior are central to the project's purpose.

6. Additional Testing: Reliability and Performance Testing

Reliability testing evaluates whether the framework behaves consistently across repeated runs and whether the same generated or replayed sequences produce stable, diagnosable outcomes. This is directly tied to one of the proposal's strongest motivations: replayability and repeatability compared with ordinary fuzzing or noisy randomized testing. If ConTiNGENT cannot produce repeatable evidence, then it fails one of its key project goals.

Reliability testing will involve repeated execution of selected generated sequences, especially around write/read, identify, format, and restore flows. The team will verify whether:

- The same sequence can be replayed consistently
- Failures can be reproduced reliably
- Repeated execution leaves the system in a diagnosable state
- State transitions remain consistent across runs

Performance testing in this project is less about optimizing user-facing latency and more about practical execution cost. Since one of the sponsor motivations is better use of DUT resources and meaningful use of idle systems, the team will measure practical concerns such as:

- Time to execute representative generated sequences
- Time spent in command execution versus orchestration overhead
- Whether the framework introduces unnecessary delays or excessive setup cost
- Whether repeated runs remain feasible in a real validation workflow

The goal is not to compete on benchmark speed but to show that the approach is stable and operationally practical.

7. Testing Workflow and Quality Controls

Defect handling in ConTiNGENT will follow a structured workflow. Bugs found through any testing category will be recorded with a description, severity, reproduction steps, affected component, and related user story or use case. At minimum, each reported defect should identify whether it belongs primarily to the TLA+ model, the execution layer, the Rust passthrough, environment configuration, or device interaction. Defects should also include the observed behavior, expected behavior, and relevant command output or trace information.

The team will use severity levels such as:

- Critical: Produces unsafe device behavior, invalidates model credibility, or blocks all meaningful testing
- High: Breaks a major workflow such as read verification or command generation
- Medium: Causes incorrect output formatting, inconsistent diagnostics, or partial functionality loss
- Low: Minor usability issue, documentation gap, or non-blocking inconvenience

A development cycle will begin when the sprint backlog, selected stories, and environment assumptions are clear. A cycle will be considered complete when the relevant implementation is finished, required unit tests pass, associated integration scenarios pass, major known defects are documented, and the feature is stable enough for the planned alpha or acceptance demonstration. Not every minor issue must be fixed before a demo, but critical and high-severity defects affecting the demonstration workflow must be resolved or explicitly accepted by the team and mentor.

Quality controls for this project will include code review, specification review, consistent use of known payloads and test devices, documentation of environment assumptions, and regression reruns after significant changes. Any feature that affects the formal-to-physical mapping should require extra scrutiny because such defects can silently undermine the project's main value proposition.

8. Conclusion

ConTiNGENT is not a conventional application; it is a verification-oriented testing framework intended to convert TLA+ specifications into meaningful NVMe generative tests on real hardware. Because of that, its testing plan must address both standard software quality concerns and the special demands of formal modeling, instrumentation, and hardware interaction. This plan does that by combining unit testing, integration testing, and usability testing with additional formal-model, reliability, and performance validation.

The proposed testing strategy is well-matched to the sponsor's goals and the structure of the project. Unit testing protects the correctness of critical parsing and helper logic. Integration testing verifies that the full toolchain-from TLA+ to TLC to Rust to nvme-cli to SSD hardware-works correctly as one system. Usability testing ensures that technical users can actually operate and understand the framework. Formal validation protects the meaning of the specification itself, while reliability testing supports repeatability and practical adoption.

Together, these testing activities provide a strong basis for delivering a functional, repeatable, and professionally credible system that supports SanDisk's goal of using specification-driven NVMe generative testing as part of a more robust standard test process.