ReLiANT

Tech Feasibility

September 30, 2025

For

SanDisk

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

Table Of Contents

Introduction

Data storage is the core of the digital age that we are in, every device that we use needs a way to store data securely and reliably. This often takes the form of NVMe (Non Volatile Memory Express) SSDs (Solid State Drives), a bug in these drives can cause data loss and downtime, so ensuring their reliability is paramount. In order to prevent any bugs, NVMe drives are subject to rigorous testing to make sure that each drive will perform as expected, however these tests are manually written by humans and are subject to human error.

Human designed tests are unable to cover all possible test cases, which leaves holes in testing that can remain there unnoticed until a consumer acquires an NVMe drive, and discovers an error. If the person running these tests automates them this can also lead to holes in test coverage and even running the same test repeatedly which just doesn't add new value but has potential to make “test noise” which just adds more data the human managing all the tests needs to go through and this is also suboptimal.

This is where our clients Chris Ortiz and Rex Jackson come into the picture, they have both been working at the forefront of NVMe development at SanDisk. Chris has been working on applying TLA+, a formal specification language for modeling and testing NVMe drives, to try and solve the issue of achieving 100 percent test coverage without repeating or running any unnecessary tests. There was a previous project that tried to tackle this problem using python and the PlusPy library to create automated tests, and the previous team was able to create a successful proof of concept. Unfortunately, there were bugs that were traced back to the PlusPy library that made the test created with this method unreliable.

Our team's solution to this problem is to get rid of python completely and use rust as the main programming language instead. We will create a program that will use Rust to interact with the NVMe command line interface, which we can use to test the SSD features that are formally specified using TLA+. Since we will not be relying on an external library, we will not need to take into account the reliability of said library, and we can implement all the features we want without having to jump through the hoops of trying to add features to a library that we do not maintain. This will allow us to manually run tests and give us the ability to generate tests to get 100 percent test coverage.

After we have completed our project, SanDisk will be able to write any manual test for a NVMe drive that they want to see tested, and they will be able to run our program in the background and that will test parts of the drive while the engineer working on the drive can do other tasks. Since our project will be able to generate tests without an engineer moderating the program, this will lead to 100 percent test coverage and little to no unnecessary test noise.

Technological Challenges

Now that we have outlined the purpose and goals of our project, we are ready to examine its technical feasibility in greater detail. This document serves as a Technological Feasibility Analysis that evaluates the key challenges and potential solutions involved in developing a Rust-based testing framework for NVMe drives using TLA+. We begin by identifying the primary technological challenges that our team anticipates throughout development. In the sections that follow, we analyze each challenge in depth, explore possible alternatives, and explain the rationale behind our chosen approaches. Finally, we discuss how these design decisions contribute to achieving a reliable, automated, and fully comprehensive testing solution for SanDisk.

Translating NVMe Specification into TLA+

- The NVMe Base Specification is complex and hardware-focused, while TLA+ is abstract and mathematical.
  Challenge: How do we faithfully model NVMe command sequences, states, and transitions without losing meaning or overcomplicating the model?

Parsing and Interpreting TLA+ Output

- The TLA+ output is typically designed for model validation, not for test generation.
  Challenge: Writing a parser/transformer that converts traces into executable NVMe CLI command sequences.

Integration of Rust with NVMe CLI

- The nvme-cli tool is primarily a C-based utility.
  Challenge: Developing safe and efficient language bindings (via std::process, FFI, or a library) to execute and capture results of NVMe commands.

Error Handling and Robustness

- Hardware operations can fail, hang, or yield unexpected results.
  Challenge: Ensuring the test runner is fault-tolerant, with timeouts, retries, and device state recovery mechanisms.

Technology Analysis

In the following paragraphs, we outline the major technological challenges that influence the design and success of our system. Each challenge represents a critical component in translating our formal TLA+ specifications into reliable, executable behavior on real NVMe hardware. By analyzing these areas, we ensure that the final product remains accurate, maintainable, and extensible across future specification versions.

Model Simulation Engine

Issue:
One of the first major challenges involves selecting an appropriate model simulation engine, the tool that takes our abstract TLA+ models and generates clear, step-by-step “traces” showing how the system behaves under different conditions. These traces act like blueprints for testing: they define each system action in sequence, allowing us to check whether an NVMe device responds as expected.

This task is difficult because TLA+ models include *nondeterministic* behavior, meaning that multiple valid next steps can exist at any point in a simulation. The simulation engine must interpret all of these possibilities correctly and produce reproducible traces that can later be executed on real hardware. If the engine misrepresents these states or fails under larger models, the entire verification process could break down.

Desired Characteristics:

- Full support for TLA+ nondeterministic constructs
- Stable long-term support (active maintenance, documentation)
- Good integration potential with downstream execution layers
- Replayability of generated traces

Choices:

- **PlusPy** - lightweight, easy to embed in Python, but limited support and unstable for larger specs.
- **TLC** - the official TLA+ model checker; mature, widely used in industry (AWS, Microsoft); now includes “generate” mode.
- **Apalache** - alternative model checker (symbolic, SMT-based) but less mature for test generation.

Analysis:
In our testing, TLC consistently produced reliable, step-by-step traces from smaller NVMe-like models using fixed seeds and depth limits. These traces, emitted line by line through our custom logging, were easily converted into JSON actions for downstream tools. PlusPy, while convenient, proved unstable for larger, more complex specifications, and Apalache required substantial setup effort without clear advantages for our use case.

Chosen Approach:
We selected TLC as our primary simulation engine. It ensures stability, reproducibility, and industry-standard reliability, key qualities for long-term feasibility and client expectations.

Alternative	**Nondeterminism Support**	**Stability**	**Trace Export & Replay**	**Ecosystem/ Adoption**	**Overall**
PlusPy	2	4	3	3	3
TLC	5	5	4	5	**5**
Apalache	3	4	3	3	3

Figure 1. Rating of choices for Model Simulation Engine

Execution Layer (NVMe Command Interface)

Issue:
Once model traces are generated, the next challenge is executing those sequences on real hardware through a consistent and safe command interface. We use the NVMe-CLI tool, a widely supported command-line utility for sending NVMe commands to SSDs, to interact directly with the storage devices.

This layer must reliably translate our generated sequences into real NVMe operations while maintaining safety across different vendors and devices. Any errors in this step could risk data corruption or inconsistent behavior, making execution reliability a key concern.

Desired Characteristics:

- Reliable execution of all NVMe base commands
- Low overhead between generated sequences and SSD interaction
- Error handling and replay support for failing sequences
- Active open-source support

Choices:

- **Python → nvme-cli:** Quick to implement but provides weaker safety checks and looser hardware typing.

- **Rust → nvme-cli:** Stronger safety guarantees and error handling; integrates cleanly into CI (Continuous Integration) workflows.
- **Vendor-specific SDKs (Standard Test Kits)** (e.g., proprietary APIs from drive vendors).

Analysis:
Python subprocess is quick to script but offers weaker safety guarantees and looser typing around hardware I/O. Vendor SDKs reduce portability and violate the vendor-agnostic requirement. A Rust runner invoking nvme-cli via subprocesses gives stronger error handling, better packaging for CI/ops, and clean logs.

Chosen Approach:
Use a Rust runner that shells out to nvme-cli. This is safe, maintainable, and aligns with the sponsor's request.

Alternative	**Reliability**	**Safety/Typing**	**Packaging/ CI**	**Portability**	**Overall**
Python -> nvme-cli	3	2	3	5	3
Rust -> nvme-cli	5	5	5	5	**5**
Vendor SDKs	4	4	3	1	2

Figure 2. Rating of choices for Execution Layer

Specification Instrumentation

Issue:
Another key challenge involves specification instrumentation, which means mapping the *abstract states* described in TLA+ (for example, "QueueFull") to the *real-world NVMe system states* observed on hardware (like queue depth or controller register values). Without a consistent mapping, we cannot validate that a given TLA+ action truly corresponds to what the device is doing.

This mapping layer is essential for traceability, allowing engineers to compare simulation results directly against hardware logs. It must also be easy to extend as new NVMe specification versions introduce additional states and commands.

Desired Characteristics:

- Transparent, standardized mapping conventions
- Extensible for new NVMe spec revisions
- Easy to validate against device logs
- Documented for reuse by engineers

Choices:

- **Manual per-test mapping** defined per test spec by engineers.
- **Formal instrumentation convention** defined in YAML/JSON configuration files, reused across specs.
- **Direct encoding inside TLA+ modules** (mixing model with device details).

Analysis:
Manual mappings are error-prone and inconsistent. Embedding device specifics inside the TLA+ model makes the specification harder to maintain. A declarative approach using YAML/JSON configuration files keeps the models clean while allowing flexible updates for future NVMe features or simulated environments.

Chosen Approach:
Use a formal mapping layer in YAML/JSON next to each spec. Specs stay abstract; mappings evolve independently.

Alternative	Consistency	Extensibility	Ease of Validation	Maintainability	Overall
Ad-hoc	2	2	2	2	2
YAML/JSON Config	5	5	4	5	**5**
Embedded in Spec	3	3	3	2	2

Figure 3. Rating of choices for Specification Instrumentation

Language and Framework Selection

Issue:
Finally, we faced the challenge of choosing the right programming language and framework to connect all components, generating nondeterministic sequences from TLA+, executing them on NVMe devices, and handling safety and performance requirements.

This choice matters because the system must safely execute potentially destructive commands, handle hardware responses, and integrate into continuous testing pipelines. It also needs to be fast enough for large-scale testing but still readable and maintainable by engineers.

Desired Characteristics:

- Correct TLA+ semantics & nondeterminism
- Stable, actively maintained tooling
- Scriptable headless runs; trace export for replay
- Reasonable performance; easy local/CI use

Choices:

- **Python** - extensible, many libraries, but weaker safety guarantees.
- **Rust** - strong safety, concurrency, performance, sponsor-preferred.
- **Hybrid approach** - Python for orchestration, Rust for critical execution.

Analysis:
Python would allow faster development but risks runtime errors in production environments. Rust ensures correctness and safety but requires a steeper learning curve. A hybrid is possible but complicates integration.

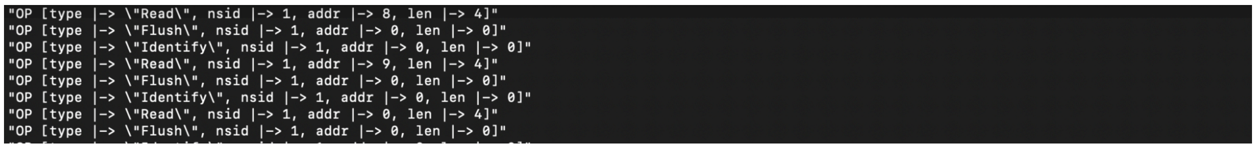
Chosen Approach:
Rust-first approach for the runner and orchestration of TLC. Use Python only for optional tooling (e.g., quick data munging) when helpful.

Alternative	Correctness	Performance	Tooling	Packaging	Overall
Python	2	3	5	4	3
Rust	5	5	4	5	**5**
Hybrid	4	4	4	3	4

Figure 4. Rating of choices for Language and Framework Selection

Proving Feasibility (Initial Demos)

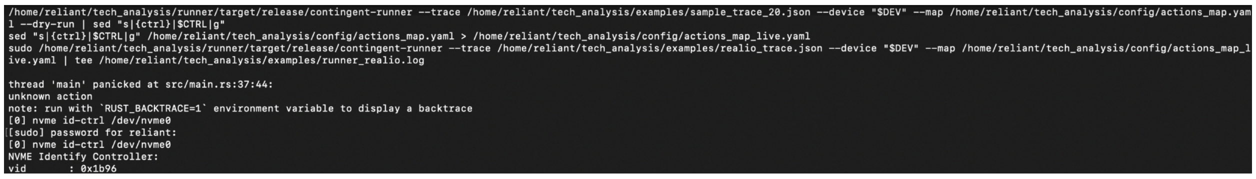
TLC test:
Ran TLC in simulation mode with fixed seed and bounded depth; each step prints an “OP [...]” record; converted to JSON actions. Runs complete in seconds with timeout and depth limits.



```
"OP [type |-> \"Read\", nsid |-> 1, addr |-> 8, len |-> 4]"
"OP [type |-> \"Flush\", nsid |-> 1, addr |-> 0, len |-> 0]"
"OP [type |-> \"Identify\", nsid |-> 1, addr |-> 0, len |-> 0]"
"OP [type |-> \"Read\", nsid |-> 1, addr |-> 9, len |-> 4]"
"OP [type |-> \"Flush\", nsid |-> 1, addr |-> 0, len |-> 0]"
"OP [type |-> \"Identify\", nsid |-> 1, addr |-> 0, len |-> 0]"
"OP [type |-> \"Read\", nsid |-> 1, addr |-> 0, len |-> 4]"
"OP [type |-> \"Flush\", nsid |-> 1, addr |-> 0, len |-> 0]"
```

Figure 5. Generating traces for simulations

Rust + nvme-cli test:
Rust runner read the JSON and executed non-destructive actions on a real device: Identify against the controller node and Flush against a namespace. Dry-run prints commands first; live run logs outputs.

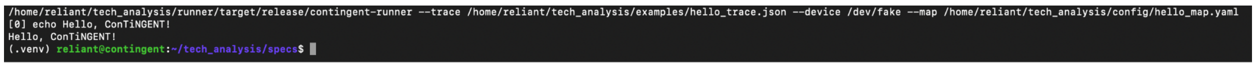


```
/home/reliant/tech_analysis/runner/target/release/contingent-runner --trace /home/reliant/tech_analysis/examples/sample_trace_20.json --device "$DEV" --map /home/reliant/tech_analysis/config/actions_map.yam
l --dry-run | sed "s|{ctrl}|$CTRL|g"
sed "s|{ctrl}|$CTRL|g" /home/reliant/tech_analysis/config/actions_map.yaml > /home/reliant/tech_analysis/config/actions_map_live.yaml
sudo /home/reliant/tech_analysis/runner/target/release/contingent-runner --trace /home/reliant/tech_analysis/examples/realio_trace.json --device "$DEV" --map /home/reliant/tech_analysis/config/actions_map_l
ive.yaml | tee /home/reliant/tech_analysis/examples/runner_realio.log

thread 'main' panicked at src/main.rs:37:44:
unknown action
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
[0] nvme id-ctrl /dev/nvme0
[sudo] password for reliant:
[0] nvme id-ctrl /dev/nvme0
NVME Identify Controller:
vid       : 0x1b96
```

Figure 6. Rust reads the generated JSON trace and performs the actions

Instrumentation draft:
YAML mapping defines how abstract actions (Identify, Read, Flush) map to concrete nvme-cli commands, with placeholders for device, controller, nsid, and parameters.



```
/home/reliant/tech_analysis/runner/target/release/contingent-runner --trace /home/reliant/tech_analysis/examples/hello_trace.json --device /dev/fake --map /home/reliant/tech_analysis/config/hello_map.yaml
[0] echo Hello, ConTiNGENT!
Hello, ConTiNGENT!
(.venv) reliant@contingent:~/tech_analysis/specs$ 
```

Figure 7. Example mapping with model NVMe and test commands

Result:

These experiments confirm the feasibility of each choice. We have an end-to-end path from spec to safe device I/O that is deterministic, vendor-agnostic, and easy to extend.

Technology Integration

To translate formal specifications into usable test sequences to run on devices under test (DUT's), these technologies must be integrated into a single architecture.

1. **TLA+ Specification** → fed into **TLC** → generates discrete states.

   TLA+ will be used to model an NVMe drive, creating a description of correct behavior. TLC will use the TLA+ specification to generate sequences of states that are guaranteed to be valid behavior.

2. **Command Parser** (Rust) → converts TLC states into NVMe command sequences.

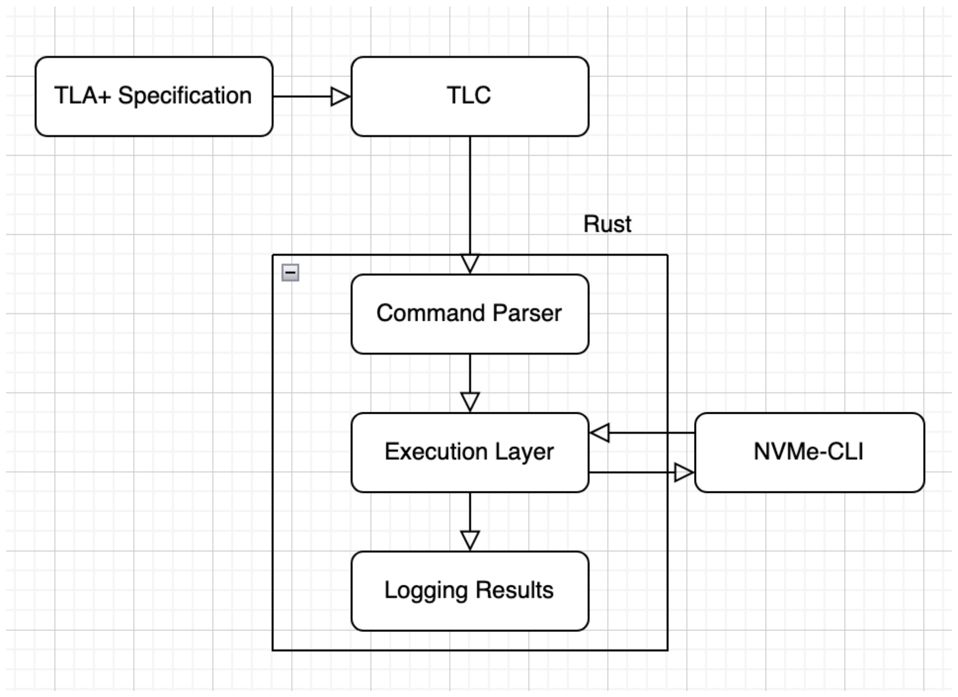
   Rust will be used to parse the states outputted from TLC in real time. It will translate the states into NVMe command sequences that can be run as test sequences on real DUT's.

3. **Execution Engine (Rust + NVMe-CLI)** → runs commands on SSD, logs results.

   Rust will be used to run the command sequence by running NVMe-cli as a subprocess. It will parse and log results of the test sequences for later analysis.

4. **Feedback Loop** → results fed back for repeatability and potential auto-retest logic (stretch goal).

   Rust will be used to automatically retest failing test sequences in an attempt to learn more about why a sequence fails.



Conclusion

In conclusion: the ReLiant project validates a robust, scalable, and fully automated framework for NVMe test generation and execution. It directly addresses the critical shortcomings of previous approaches instability-instability and inadequate coverage-to provide SanDIsk with a future-proof testing methodology.

The refined selection of TLA+/TLC, Rust, and a configuration-based instrumentation layer forms a high performance foundation that ensures long-term reliability and ease of maintenance. THe architecture's viability is already demonstrated through successful feasibility tests.

ReLiANT’s ultimate achievement is its transformation of a critical engineering process. It eliminates past fragilities and establishes a flexible platform for future test needs, empowering SanDisk to enhance product reliability, reduce development overhead, and solidify its leadership in storage innovation.