Final Report

SSDynamics

May 8, 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

Introduction

Insuring the reliability of cutting edge NVMe solid state drives is critical for industry leaders like Western Digital, yet traditional manual validation methods struggle with comprehensive coverage and adapting to rapid advancements. These limitations, including potential biases and the difficulty in uncovering obscure edge cases, can slow development and impact product quality. To address these challenges, our NAU capstone team, SSDynamics, developed NVMe-Gentest, an automated testing framework. This proof of concept leverages random model simulation driven by formal TLA+ specifications to enhance the depth and efficiency of NVMe drive validation.

The core vision behind NVMe-Gentest was to empower Western Digital's validation engineers by shifting their focus from the tedious, low-level definition of individual test cases to a higher-level, model-based testing strategy. By automating the generation of diverse and complex test sequences, NVMe-Gentest aims to not only improve the likelihood of discovering elusive bugs but also to free up valuable engineering resources. This allows skilled engineers to concentrate on more complex system analysis, innovative test strategy development, and the interpretation of results, ultimately fostering a more agile and thorough validation environment.

This Final Report documents the NVMe-Gentest project, from its conceptualization through to its current state. Within these pages, we will detail the identified project requirements, the system architecture and implementation choices made, the testing methodologies employed, and a reflection on the project timeline and potential avenues for future work. Our aim is to provide a comprehensive overview of the solution developed to address Western Digital's validation needs and the outcomes of our efforts to create a more streamlined and robust testing process.

Process Overview

Development Lifecycle

Our team used an agile approach, organizing and assigning tasks in sprint-like cycles supported by weekly meetings:

1. Internal meetings
2. Mentor meetings
3. Client meetings

During each sprint, we assign work about a week before an official deadline as a buffer for delays.

Tools

Version control: Github with a main/test/dev branch

Documentation: Google Suite (Docs, Slides, Drive)

Issue and Task Tracking: Google sheets + GitHub Issues

Deliverables: Markdown for code docs and PDFs for documents

Team Roles and Responsibilities

This project requires that we both handle school capstone assignments and the client requirements. As such we felt the following roles were useful: Team Leader, Customer Communicator, Recorder, Architect, Release Manager and Coder. Each member will act under one of these roles, all members will have a coder role.

- Team Leader: The team member who coordinates task assignments, runs meetings, and reviews final documents to verify requirements as well as ensures work is progressing, and makes efforts to resolve conflicts.
    - Chas Diaz
- Customer Communicator: The team member who coordinates and conducts customer communications.
    - Charles Descamps
- Recorder: This team member maintains detailed meeting minutes.
    - Carter Kaess
- Architect: This team member is primarily responsible for ensuring that core architectural decisions are followed during implementation.
    - Charles Descamps
- Release Manager: This team member coordinates project versioning and branching, reviews and cleans up commit logs for accuracy, readability, and understandability, and ensures that any build tools can quickly generate a working release.
    - Connor Aiton
- Coder: It is expected that everyone will have a role in producing code. If possible at this early stage, you might specify *what parts* of the coding (backend, front-end, node.js, MSP430 programming, etc.) that individuals will lead on.
        - Code Sections:

        - TLA+ to PlusPy
        - PlusPy Modification and error traceback
        - Python to NVMe CLI
        - Logging
    - Everyone

Organization and procedures

Version Control

GitHub (Code/Code Documentation):

ALL capstone related code (including forks of open source projects) will be under the SSDynamics GitHub organization. The main code base will have 3 main branches: dev/test/main, other branches for specific and notable features/experiments can be created as needed (stale branches will be closed). To improve and enforce code quality, GitHub actions/workflows will be used. Capstone related deliverables will be stored collaboratively on Google Drive.

- Each member will have a fork of the main code base to work on individually.
    - Keep private forked repositories private
    - Update fork by pulling the main repository before attempting to merge up to remote
    - Merge requests will not be accepted or reviewed if it fails automated tests (unless a flaw in automated testing is found)
        - If there is a flaw to fix in automated testing, track on task tracker with high priority
    - “Pull often, Commit often” for each subtask of a feature and at minimum submit one pull request back to main for each feature.
        - Unless highly related, do not submit a pull request for multiple features
- Branching strategy
    - New development is done in dev branch (automated linting may be added)
        - A sub strategy that is commonly used:
            1. Someone may create a new branch locally for a feature being developed,
            2. If other member pushes to organization repo first, member pulls dev branch from organization repo to local repo’s dev branch
            3. Then, member merges dev into feature branch, resolving merges (thus keeping dev branch history and conflicts more clean)

    - Complete features should be pushed back to main repo for review and basic testing
    - Testing branch may be used for unit/integration testing and ensuring that new features work as intended rigorously
    - If everything is working and passes, Test may be merged into main tagged as a stable version
        - To further add to trackable stable version, use GitHub releases and packaging
    - General workflow: local Dev→ PR to remote Dev→Test→ Main
    - Improper use of branches and repos may result in rejected PRs and administrative repo actions by release manager to maintain order

Google Drive (Deliverables and Capstone Documentation):

- Use for tracking and storing all capstone related deliverables that will be used for grading
- Drive will have at minimum three folders under a shared root folder: assignments, meeting agendas and misc.
    - Assignments: Any deliverables that are being turned in
    - Meeting agendas: Team, Mentor and Client meeting goals and schedule
    - Misc: Any other files that facilitate team productivity and collaboration

Issue Tracking

Team Task Tracker (Larger trackable tasks) and GitHub Issues (Smaller programming subtasks from task tracker)

To track issues that arise or are foreseen in this project, we will be using a mix of two trackers in this project. For larger more complex tasks or non-coding related tasks, they will be tracked using the task tracker sheet. The smaller subtasks that may appear from breaking down tasks from team task tracker, even if small, will be in GitHub Issues.

- Use the organization's repository for GitHub Issues in all coding and code documentation issues
    - Issues will remain short and concise. These should be understandable from a glance.
    - Start broad then go narrow if a longer description is needed, each issue should contain the following:
        - Issue
        - Replication
        - Suspected cause if any

    - Smaller coding subtasks that may be broken down from the task tracker will appear here
    - As features or patches are pushed back to organization's dev branch, describe fix using a GitHub closing term referencing the issue number
    - For each coding issue, assign at least one member to the issue (most likely whoever is assigned to the parent task on the task tracker sheet)
    - When there are multiple GitHub Issues for each person, GitHub Project board should be used
        - Utilize to do, in progress and completed board effectively
        - Prioritize tasks, tasks depended on by other tasks will inherit priority
        - Estimate small/medium/large
    - Bad/Non-descriptive issues will be closed

- For deliverables and capstone documentation, use Google Sheets task tracker
    - Break larger complex tasks into smaller tasks that can be completed by an individual person
        - Tasks that are larger than what is possible completing in one sitting should be broken into further subtasks
    - Failure to accurately use use task report may result in inaccurate contribution and time estimates
        - Responsibility of proper use ultimately falls upon the person doing the task
    - Each tasks will be descriptive, the type of task needs to be specified and hours will be estimated
    - If multiple people are working on the same task, mention percentage of contribution in the comments is required on completion
    - Set a default internal due date due one week before the actual due date
        - With a grace period of ~3 weeks on team initialization, regular delays on internal due dates will be discussed as a group
    - Remember to update the status of your tasks
    - Should contribution representation conflicts arise, team as a whole will review the revision and commit history together

Word Processing and Presentation

- Google Docs, Slides and Sheets completion will be a PDF unless otherwise stated
    - Particularly for simple plaintext documents that require high collaboration

- Other forms of word processing like Docs, Overleaf, Sheets and Powerpoint may be used as more formatting is needed.
- For project related documentation, use markdown.

Composition and Review

- Split up the document into the distinct smaller parts in the task tracker to uphold responsibility
- On a word processor with no live collaboration, assign one person to handle compiling and tracking everyone's contributions as one file
- Assign proofreading in task tracker
- On draft ready for final review, use Internal deadline, which will be one week ahead of actual deadline
    - When the draft is ready, submit to lead editor: *Charles Descamps (tentative)*

Requirements

Review of the Acquisition Process

The current SSD validation workflow at Western Digital, while robust, faces several challenges that initiated the acquisition of this project. These include limitations in test coverage due to reliance on engineers' understanding, leading to potential biases and missed edge cases. Manually designing and adapting test sequences for evolving NVMe requirements is labor-intensive, time-consuming, and imposes a significant cognitive load on engineers, especially newer ones. These inefficiencies can slow down the validation process and increase the risk of undiscovered issues in NVMe drives.

To address these deficiencies, this project was conceptualized. The core idea is to develop a proof of concept that utilizes a random model simulation framework to generatively create test cases. This approach aims to automate the generation of test sequences based on a high-level TLA+ specification file, thereby removing the need for manual, low-level test case definition. The vision is to enhance test coverage, reduce human bias, improve the detection of edge cases, and increase adaptability to new requirements by allowing engineers to focus on higher-level design and validation tasks. This automated system is expected to link a model simulator to an NVMe command interface, enforcing consistency and improving the overall efficiency and reliability of the NVMe validation process.

Resulting Requirements

The acquisition process described above led to the definition of the following requirements for the SSD Validation Proof of Concept:

Domain-Level Requirements

These are high-level requirements that provide an overview of the system's objectives:

1. **Comprehensive NVMe SSD Validation:** The system must enable thorough validation of NVMe drives by automating the generation, execution, and logging of test sequences, covering large state spaces to find edge cases.
2. **Random Model Simulation:** It will use randomized simulation to explore potential NVMe behaviors, ensuring robust testing.
3. **Traceability:** The system must allow tracing back to points of interest, enabling engineers to investigate and verify repeatability of events like bugs.
4. **Efficient Test Execution:** It must handle high volumes of test operations efficiently, providing timely feedback through effective output parsing.
5. **Scalability and Versatility:** The system needs to be adaptable to support a wide range of NVMe opcodes (admin and IO passthrough) and accommodate new NVMe features or specification changes without major architectural overhauls.
6. **Drive Integrity and Consistency:** It must verify the consistency and predictability of Read/Write operations to ensure NVMe drives maintain integrity.
7. **Usability and Accessibility:** Designed for validation engineers, it will feature an intuitive terminal interface, understandable by new junior engineers.
8. **Compliance and Standardization:** The system must strictly adhere to NVMe specifications and industry best practices.

Functional Requirements

These requirements detail the specific functions and features of the project:

1. **Validation Framework Features:**
    - **Test Case Generation:** The system will generatively create random test sequences from a design-level model-simulation file (TLA+) that adheres to NVMe standards. This file will be model-checked for design flaws and will define high-level rules for state combinations mapping to real-world functions. Interpretation of this file will output states and transitions, mapping model states to validation functions to cover edge cases in large state spaces. Random seeds will determine test sequence generation.

- **Command Execution:** The system will execute all commands available in `nvme-cli` via admin and IO pass-through commands. It will support all admin opcodes (e.g., Get Features, Set Features, Identify) and IO opcodes (e.g., Write, Read, Flush). A consistent interface for pass-through functionalities will be maintained. A watchdog timer will handle hanging processes. The system will explore new paths upon reaching the end of a state space or generate a new state space if none are available.
- **Result Evaluation:** It will provide comprehensive output from command execution within a test sequence. Results from NVMe CLI will be parsed and evaluated to identify anomalies. Command output will indicate success/failure (stderr), return data (stdout), and exit codes for verification.

2. **Simulation Capabilities:**

    - **Randomized Generation:** The system will use randomly created and stored seeds to generate test sequences based on constraints from the model simulation file, aiming to avoid human bias. New seeds will be generated when a state space is fully explored.
    - **State Exploration:** Exploration will follow a depth-first search pattern to quickly find unique states. Deviations from expected states can identify anomalies.
    - **Resampling Capabilities:** Traceability and resampling will be implemented by storing and utilizing seeds to recreate previously explored issues or bugs, though identical path execution is not guaranteed.
    - **Bridge between Model Simulation and Real-World:** The system will be driven by state space model simulation. It will intercept model simulation outputs and make blocking calls to another interface that maps these outputs to callable functions and arguments; the model simulation will pause until the call returns.

3. **Data Handling and Reporting:**

    - **Data Integrity Checks:** Test execution data will be analyzed to ensure functionalities manipulate and read data as expected during feature development.
    - **Reporting and Logging:** The CLI will output hex dumps following NVMe specifications, with functions to translate these to human-readable data for critical errors/functionalities. Results must be traceable for error recreation. Logging should be efficient and not significantly impact performance. Engineers might need to understand raw hex dumps for untranslated outputs.

4. **Accessibility and Maintainability:**

    - **Modular Design:** The program will feature a modular, layered design with four main modules: TLA interpreter, NVMe interface, logging, and a bridging model connecting them. This decoupled design allows for code changes as long as interfaces remain consistent or consuming functions adapt.
    - **User Interface:** It will present a user interface similar to IEEE Linux utility conventions (IEEE Std 1003.1-2017). A customizable logger (via config file) and a user guide (prompted or on error) will be included. The interface will cater to both new and experienced engineers, abstracting inner workings while providing control through options and argument passing.
    - **Documentation:** Hierarchical documentation will be provided, from surface-level functions to deeper implementations, using diagrams for high-level concepts. Automated documentation will be used for lower-level program aspects for consistency. Source code will have specific comments, and commonly used functions will have docstrings. Each file's purpose will be described.
    - **Tech Debt and Limitations:** The proof of concept will be simplified to meet deadlines and may not fully align with all validation team needs. It allows anomaly replication but doesn't guarantee deterministic execution paths. Reliance on external tools may require maintenance. Features like AI simulation tuning, state-space visualization, automated output analysis, and Jira API integrations are outside the project's scope.

Performance Requirements

These are testable requirements for system performance:

1. **Execution Speed:**
    - Validate 10,000+ simple admin test sequence executions within 24 hours.
    - Ability to parse NVMe logs and generate summaries in under 'x' seconds per log (specific 'x' TBD).
    - Logging from execution should not cause significant overhead (less than 500 milliseconds).
    - The system will loop indefinitely until the end of an execution path is found.
2. **System Responsiveness:**
    - User interactions should have a response time of less than 3 seconds; longer functions must indicate progress.

    - If any opcode's blocking interval exceeds 60 seconds, the system shall automatically proceed to the next opcode.
    - A log will be returned within 10 seconds after an error.
3. **Accuracy:**
    - The program will detect and flag anomalies with nearly complete accuracy, verified through resampling.
4. **Resource Efficiency:**
    - Optimize CPU and memory usage for smooth operation under heavy test loads.

Environmental Requirements

These specify the software and technology context for the project:

1. **Hardware Constraints:**
    - Compatible with Linux-based systems (kernel 6.8 or later).
    - Support for NVMe SSDs (NVMe 1.4 or later).
    - Requires a secondary target NVMe SSD.
    - RAM and CPU capable of virtualization and high throughput/IO operations.
    - Motherboard with current NVMe compatibility.
    - Internet/internal network accessibility for the server.
    - Testing server located with consistent power and internet.
2. **Software Dependencies:**
    - Must run Ubuntu Server 24.04 LTS distribution.
    - Integrate with NVMe CLI and Python-based tooling.
    - KVM+libvirt support required for sandboxing and portability.
    - PlusPy needed to interpret TLA files.
    - Latest `nvme-cli` installed.
    - Developed under Python 3.12.
3. **Standards Compliance:**
    - Ensure compliance with NVMe command specifications and testing protocols.
    - Modifying source code and implementing model-simulation files requires a deep understanding of industry-standard specifications.

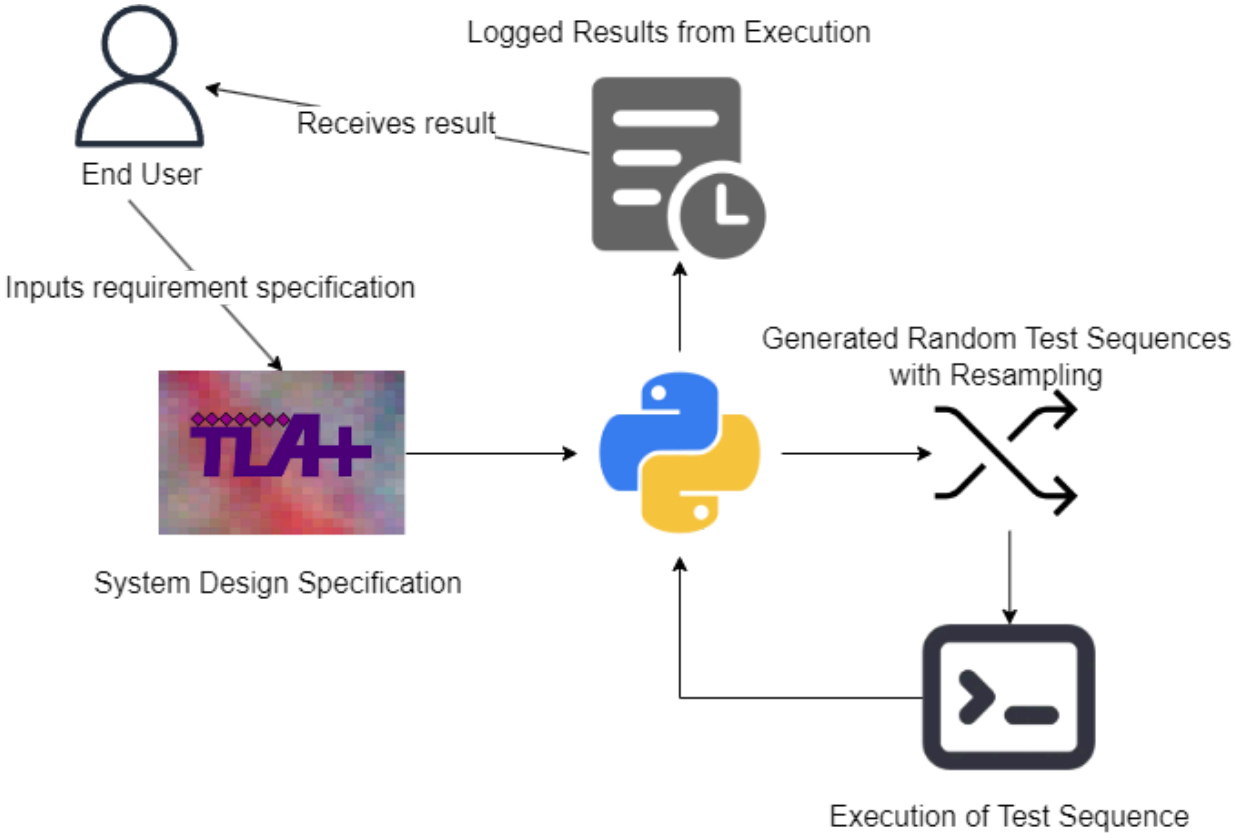
Architecture and Implementation [Carter]

System Overview

NVMe-Gentest is designed as a modular testing framework for NVMe SSD devices, driven by formal specifications written in TLA+. At its core, the system uses a layered architecture with clean separation of responsibilities across modules that parse formal specs, generate test scenarios, execute commands via the NVMe CLI, and capture logs for analysis.

The architecture supports both automated and manual test workflows and was influenced by **layered** and **pipeline** architectural styles, where data flows in a clear, top-down path and each layer transforms or acts upon the input from the previous one.

Architectural Diagram



The system is composed of the following high-level components, connected in a logical flow:

1. **TLA+ Specification File**
   Provided by the tester, this defines the expected behavior of the NVMe device under test, including allowed commands and state transitions.

2. **PlusPy + Parser Module**
   - **Responsibility**: Loads and interprets the TLA+ file.
   - **Example**: Reads a specification that allows a sequence like `write → read → flush`.
     Uses the `PlusPy` library to simulate the state machine and emits valid state transition outputs for the test runner.

3. **Test Runner (`NVMeTester`)**
   Responsibility: Central controller that manages the test lifecycle.

   - **Key Tasks**:
     - Loads TLA+ constants and initializes the interpreter
     - Generates or uses a provided seed
     - Loops through state transitions
     - Resamples seed if a bug is detected
   - **Communication**: Calls the dispatcher with parsed output at each step.

4. **Dispatcher**
   - **Responsibility**: Translates PlusPy output into CLI-ready arguments.
   - **Example**: Extracts `"cdw10 |-> 5"` and converts it to `--cdw10 5`.

5. **Executor**
   - **Responsibility**: Builds and executes the final NVMe CLI command using Python's `subprocess`.
   - **Features**:
     - Handles namespace path formatting (e.g., `/dev/nvme0n1`)
     - Supports both `admin-passthru` and `io-passthru`

6. **Logger System**
   - **Responsibility**: Tracks all events in a rolling log file.
   - **Features**:
     - Timestamped entries
     - Error summary section
     - Seed, command, and stderr trace capture

    - Custom error handler used by the test runner

Component Breakdown

`interface.py` (CLI Entry Point)

- Parses command-line arguments
  Passes runtime configuration (e.g., seed, count, CLI device) to `NVMeTester`

`tester.py`

- Manages full test execution logic
- Interfaces directly with `PlusPy`, dispatcher, and error handling
- Contains logic for seed resampling and bug detection
- Calls `clear_logs()` and sets up the error summary mechanism

`dispatcher.py`

- Parses raw PlusPy state output
- Translates TLA+ dictionary output into structured argument dictionaries

`executor.py`

- Executes real commands using `nvme-cli`
- Captures stdout, stderr, and exit codes
- Logs both human-readable and machine-parseable results

`logger_config.py`

- Manages logging setup and teardown
- Implements custom rolling file handler and error collector
- Writes a summary of all error-level messages at test end

Architectural Influences

The system primarily follows a **layered architecture**, where:

- Each layer has a single responsibility

- Control flows downward (from spec to output)
  Each module passes structured data to the next

It also reflects a **pipeline style** in the way state → command → result is processed sequentially.

As-Built vs. As-Planned

During early planning, we envisioned a three-phase model:

1. Parse TLA+
2. Generate CLI tests
3. Analyze results

While this model largely held, there were several key deviations:

- **Deferred Features**:
  The logging verbosity flag (`--logging-level`) was partially implemented but excluded from the final release due to time constraints and lack of test coverage.

- **Enhanced Features**:
  Rolling log management and detailed error summaries were not in the original plan but became critical during early test cycles and were added accordingly.

- **Architecture Adjustments**:
  Originally, the PlusPy integration and the parser were to be separate components. In practice, they were combined to simplify state tracking and reduce parsing complexity.

- **Testing Strategy Evolution**:
  Early versions used fixed test cases. Through iteration, the system evolved to support both fixed and generative (resampled) test seeds, which significantly improved bug reproduction and coverage.

Testing

Testing the generative testing platform took some work, because it's generative, it requires some more manual validation to make sure that the platform works. We wrote a

lot of unit tests, as this is where most of the work happens, within our modules. Each module needed to work as best as possible at its one job. We decided to use PyTest for our unit testing to keep it simple and within the standard library so that it would make installation easier.

Unit tests tested all of the modules that we wrote for the project. We decided that modularity was important for the project, and wrote the platform to be as modular as possible. This made it easier to do unit testing, as we would treat each module as their own set of test cases.

Integration testing was pretty much checking that the modules communicated with each other. In most circumstances, we tested the integrations by running the platform in its entirety. This is because each module was used within our project, and that they were coupled in a way that allows for the integrations tested.

Usability testing was performed similarly to integration testing, as we took a stance of how the test engineers would be using the platform. Using the platform includes providing an NVMe device path and a TLA+ specification. The real utility here is in making sure that the TLA+ is parsed correctly, which our modified PlusPy library is taking over. Finalizing usability testing will be done through giving the platform to engineers to work with and getting feedback. Since our product is mainly used by test engineers, we really wanted to keep the options simple and straightforward, which is why the product has ‘in-line’ help. This is intended for the engineers so they don’t have to consult a manual, they can instead focus on writing TLA+ testing specifications.

Overall, testing the NVMe-Gentest platform showed a few problems, mainly through demonstration/usability testing. The NVMe-Gentest platform, being a command line interface only, means that all the problems will show through really easily. This is because errors or other verbose language is seen within the program and the logs for it. We had to make very little changes because of testing through unit or integration tests, which is great for the modularity of the program.

Project Timeline

Establishing a timeline for the project, knowing what we know now, we can see the very obvious path to take. However, when we were initially planning out the project, we had to make sure that we focused on the big parts of the project. We had scheduled things so that we could focus on making modifications to PlusPy, and jump right into the project. Below are the tasks that we focused on in the project:

- Opcode Validation
- TLA+ Parsing (Modifications to the PlusPy library included here)
- PlusPy Command Blocking (More modifications to PlusPy)

- NVMe-CLI Admin Passthrough
- TLA+↔PlusPy (Communication between TLA+ model and PlusPy parser)
- Logging (Decided to use a rolling logging method)
- Seeding and Resampling
- TLA+↔PlusPy↔NVMe-CLI (Communication between all major parts of the project)

These tasks were decided on in the first semester, and when we came back from break, our tasks were more specific. This helped us keep on track for the project, and focus more on the implementation rather than high level requirements. These are the milestones that we focused on within this last semester:

- Interface Python File
- Logging Configuration
- NVMe Python File
- TLA Parser
- PlusPy Modifications
- Testing

Notice that there are a lot less tasks, this is due to the fact that we had focused a lot more on the functionality here, and less on the high level requirements, since we had a much better understanding of the project by the end of the first semester. Overall, the timeline for our project was linear, and we immediately started on the next part of the project instead of taking a break. During the first semester, the team was very much trying to figure out what we would specialize in during programming. Since we didn't understand the project well at that point, it was hard to assign tasks to the team. During the second semester, after we figured out what the project was all about and what tasks everyone was comfortable with completing, we found more specialized roles for everyone to complete. Right now, we are focused on making our last finishing touches on the project before delivery to the client.

Future Work

Further work on the NVMe-Gentest platform was mainly decided by the team and discussed with the client. We found three main improvements.

First, is threading. Threading would allow for multiple tests to be run concurrently, thus allowing testing to be done on multiple simulations, and allow for communication between them. This would essentially allow for a LOT of testing to happen at once, 'brute-force' style, and could allow one to catch a lot of cases that developers might not catch.

Second, is external communication. Our product is quite closed, and extending it, means that it must be done at the Python level. The best thing to do would be to allow for communication between the TLA+ model and the scripts. This is sort of implemented as it is, but doesn't fully work. Further work would have to be done to make it completely operational.

Third, is improved simulator communication. Our current implementation has a very basic way to communicate between the PlusPy library and the TLA+ model. This means that we know that the TLA+ model can be parsed, and we've made enough modifications to it to allow for most of the basic use cases, however, additional functionality might require future work to allow for more thorough communication.

If we were to pass this product off to another group to have them improve it, they would certainly have to walk through the entire code before even attempting to improve upon it with these discussed features.

Conclusion

The NVMe-Gentest platform was created with the intention of adding another tool to the testing toolbox of validation testing engineers for NVMe SSDs. The project was meant to allow for a 'brute-force' method of testing, testing the boundaries of logic. While targeted testing is great for catching about 90% of the logical bugs, the last 10% are difficult to impossible to find when the human is involved. Our platform attempts to improve this by finding all possible states and paths a drive could possibly take and figure out if there is a specific path in which it will crash, and the steps it took to create that problem so that it could be fixed. This new method of testing will help test engineers find new bugs that could have never been found through traditional testing.

As a team, we created a testing platform that should improve the current methods of testing that are in use right now. We certainly struggled with the new concepts that we had to learn in order to finish the project, like TLA+. Our team strived to get our project as close as possible to what our client wanted, and we feel like we came up with the best solution we could with our time. The Capstone class and all of its documentation helped us create a better project for our client, and without all of their help, this generative testing platform would likely not exist.

Glossary

Appendix A: Development Environment and Toolchain

This section will provide a quick overview of how to get set up with our project and start contributing.

Prerequisites: Linux (Ubuntu LTS or any modern linux distro with nvme-cli and Python 3.12+ installed)

Hardware: When setting up this project, we recommend having a separate NVMe drive to run validate that is not the main drive (with OS stored). The project does not use that much compute power but will benefit from higher clock speed. More storage in the main drive will also allow for more logging to be stored.

Toolchain:
Version control: Git/GitHub

- We use Git/GitHub due to it already being installed on linux and its ease of use. There are 3 branches:
    - Main: The final code/release
    - Test: Code that is ready for testing
    - Dev: All other code/development code
- To automate some of the testing, we have included some github Actions scripts
    - Normally github branch rules and some deployment scripts would be working here but because the repo is private, GitHub does not allow this under free use.

IDE: Visual Studio Code + TLA+ Toolbox (optional)

- You can use any coding environment that supports python but we recommend visual studio code because of the plugins that it supports that makes development much easier
    - Any extensions that you would use for Python
    - TLA+ extension by TLA+ foundation (allows for simulation of TLA+ files without the TLA+ toolkit within the editor)
    - SSH, if you are using a separate machine to run the validations on, the SSH extension written by microsoft makes the workflow of developing and operation easier

Setup:

1. Install or virtualize linux on a system with a spare NVMe drive
    a. If you are virtualizing linux you need to make sure that the NVMe is passthru from host to virtual machine. You also will need to ensure that virtualization is enabled on the CPU in BIOS
2. Install IDE with Python and TLA+ plugins
3. Read the main README for directions on how to start the project

Production cycle:
Luckfully because all of the code is mostly interpreted, and the code uses all builtin libraries, there is not much to do for the production cycle. There is no build or compile to do. When developing, you'll want to make your own fork and develop in the “dev” branch, then our team usually follows the following process:

1. Save small changes to dev
2. When code is ready, run “flake8 .” to ensure that there are no formatting issues, our configuration for flake8 is stored in the setup.cfg file
3. Push into your fork
4. Pull fork into main/collaborative repository, resolving conflicts.

5. Merge dev branch with test on major feature completions and run any unit/integration testing
6. Final product should be merged into the main branch on the collaborative repository