Project Requirements V1.0

SSDynamics

Oct 4, 2024

Carter K. Chas D. Connor A. Charles D. Brian Donnelly

Chris Ortiz

Senior Technologist, SSD Validation

Western Digital Corp., Flash Business Unit

John Lee

Senior Director, SSD Validation

Western Digital Corp., Flash Business Unit

Client Signature : ______________________

Mentor Signature : ______________________

Table of Contents

Introduction

Solid-state drives have become integral to modern storage technology, from personal devices such as laptops, desktops, and gaming consoles to enterprise applications such as cloud data centers and public—or private-facing servers. Even with power loss, this component allows users to store data persistently without loss. In this project, we will focus on a specific type of Solid-state drive known as an NVMe (non-volatile memory express) drive, a fast, cutting-edge storage technology that relies on NAND flash memory to store electrons behind precise atomic-sized gates representing 1s and 0s. NVMe drives are renowned for their performance in efficiently handling data, which makes them ideal for intensive and high-demand applications that require high throughput and responsiveness.

Thorough validation of NVMe production is vital to remaining competitive and thriving in a rapidly growing market estimated to be worth tens of billions. Western Digital is a significant player in this space. Validation teams such as the one in Western Digital work extensively to ensure this component works correctly and that an NVMe drive meets or exceeds product requirements.

Traditionally, engineers define possible combinations of test sequences to test the product requirements based on their understanding of a system. While there is a predictable level of coverage, this testing method has some limitations; it introduces a few problems regarding validation: The test combinations rely on the engineer's low-level understanding of a system, which limits coverage and introduces unconscious bias, and finding new testing combinations becomes tedious and time-consuming. Furthermore, edge cases (scenarios outside usual use patterns) are challenging to predict manually, resulting in critical issues appearing in a product cycle or live production.

To address these challenges, Western Digital's senior technologist in SSD validation, Chris Ortiz, entrusted our NAU capstone team with creating a working proof of concept to address these workflow inefficiencies. Our proof of concept will demonstrate how random model simulation can drive real-world testing, removing the need for SSD test engineers to define low-level test case sequences to test a product. This novel approach will allow for automated testing within the validation workflow by generating test sequences, which enable a validation process to explore a more comprehensive set of test cases without requiring explicit definition or intervention from engineers. This automation will help ameliorate the tedious nature of manual test creation, increasing the overall productivity and thoroughness a validation team can expect. With a better-automated validation process, we hope to accelerate the process, ultimately enhancing the reliability and innovation of NVMe drives by paving the way for a more streamlined validation process.

Definition of Terms

Storage Technology Terms

- **NVMe (Non-Volatile Memory Express)**: A high-performance storage protocol designed specifically for SSDs that communicate via PCI Express (PCIe) bus.
- **SSD (Solid State Drive)**: A storage device that uses integrated circuit assemblies to store data persistently, typically using NAND flash memory.
- **NAND Flash Memory**: A type of non-volatile storage technology that retains data even without power, using electrons stored behind atomic-sized gates to represent binary data (1s and 0s).
- **Throughput**: The amount of data that can be processed or transferred from one location to another in a given amount of time.

Validation and Testing Terms

- **Edge Case**: An unusual or extreme situation that occurs at the outer limits of normal operating parameters.
- **Test Sequence**: A series of commands or operations executed in a specific order to validate system behavior.
- **State Space**: The set of all possible states that a system can be in, along with the valid transitions between those states.
- **Test Coverage**: The degree to which a test suite exercises the full functionality of a system.
- **Regression Testing**: Testing performed to ensure that previously developed features still perform correctly after changes.

Technical Framework Terms

- **TLA+ (Temporal Logic of Actions)**: A formal specification language used to design, model, document, and verify concurrent systems.
- **PlusPy**: A Python framework for interpreting and executing TLA+ specifications.
- **Model Simulation**: A technique that uses computer models to imitate and explore system behavior under various conditions.
- **Random Model Simulation**: An approach that uses pseudo-random number generation to explore different possible states and transitions in a system model.
- **Opcode**: Operation code; a instruction that specifies what operation is to be performed.

Command Interface Terms

- **NVMe-CLI**: Command-line interface tool for managing NVMe storage devices.
- **Admin Commands**: Special NVMe commands used for device management and configuration.
- **IO Pass-through Commands**: Commands that allow direct communication with the storage device.
- **Command Execution**: The process of running specific NVMe commands on the target device.

System Architecture Terms

- **Modular Design**: A software design approach that separates functionality into independent, interchangeable components.
- **Interface**: A shared boundary across which information is passed between components.
- **API (Application Programming Interface)**: A set of definitions and protocols for building and integrating application software.

Problem Statement

Current Validation Flow:

In the context of Western Digital's SSD validation team, the process is built around ensuring products meet stringent performance and reliability standards before product release. In this case, the workflow includes analysis, defining, executing, and iteration (Fig 1.0) of various scenarios to ensure their products, such as NVMe drives, perform precisely as intended in the real world.

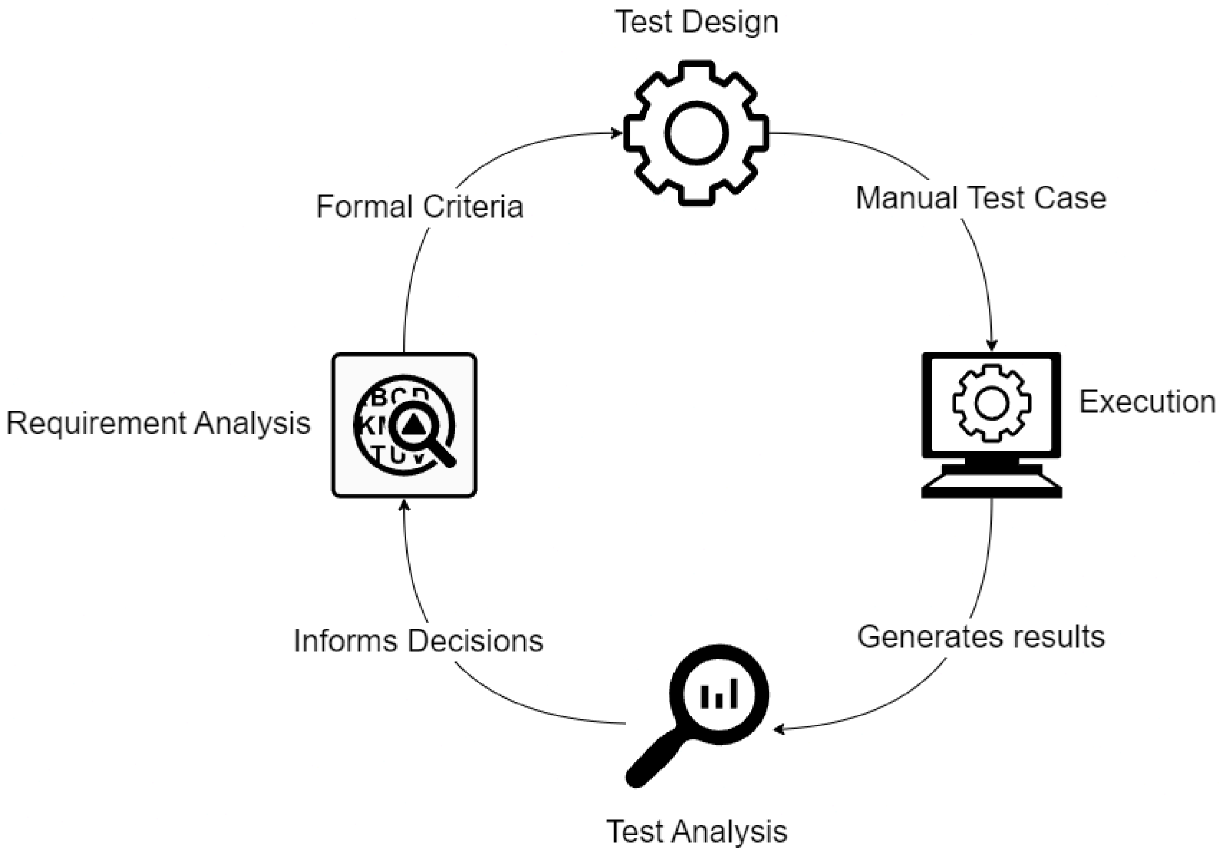


Fig 1.0 Validation flow diagram

The key components in validation follows this order:

1. Requirement Analysis: Engineers research and create thorough requirements for NVMe drives based on the customer needs and industry standards
2. Test Case Design: Test engineers will then use their understanding of SSDs to test the previously defined requirements against a product, often manually designing test sequences or automating typical cases that cover specific and expected scenarios.
3. Execution: The test sequences created by the engineers are then executed on actual NVMe drives, logging anything that would indicate deviations from the expected result.
4. Test Analysis: The logs and results are analyzed to determine bugs or failures, improving the product's next iteration or re-running tests if needed.

The current workflow is robust and ensures that an NVMe drive can meet product expectations, but it has several limitations.

Workflow Deficiencies (see fig 1.1 below):

- Limited Coverage and High Bias: Manual test sequence creation or automation is inherently limited because engineers focus on exact scenarios and known edge cases, which can lead to blind spots. Since we rely on test engineers for this lower level of testing, manual test definitions are more likely to introduce unconscious bias based on an engineer's background and knowledge, leading to overlooked or over-assumed scenarios.
- Inadequate Coverage of Edge Cases: Extending the limited coverage, traditional methods struggle to uncover the nuances of unforeseen behaviors and failures that may only appear under sporadic conditions and are often only discovered after real-world usage.
- Labor-Intensive: Manually defining and creating test sequences is highly tedious, especially when trying to provide a comprehensive coverage of possible behaviors. This adds to the project timeline and costs.
- Requirement Adaptability: As innovation and products evolve, so do the requirements. New requirements may entail considerable continuous adjustments or extensions to test cases, meaning test engineers will have more tech debt from old tests as requirements change.
- Cognitive Load/Productivity: In this project, engineers must retain detailed low-level knowledge of SSD design, specification, and requirements. Current SSD validation methods, particularly for newer engineers, impose a high cognitive load, impacting productivity and design errors.

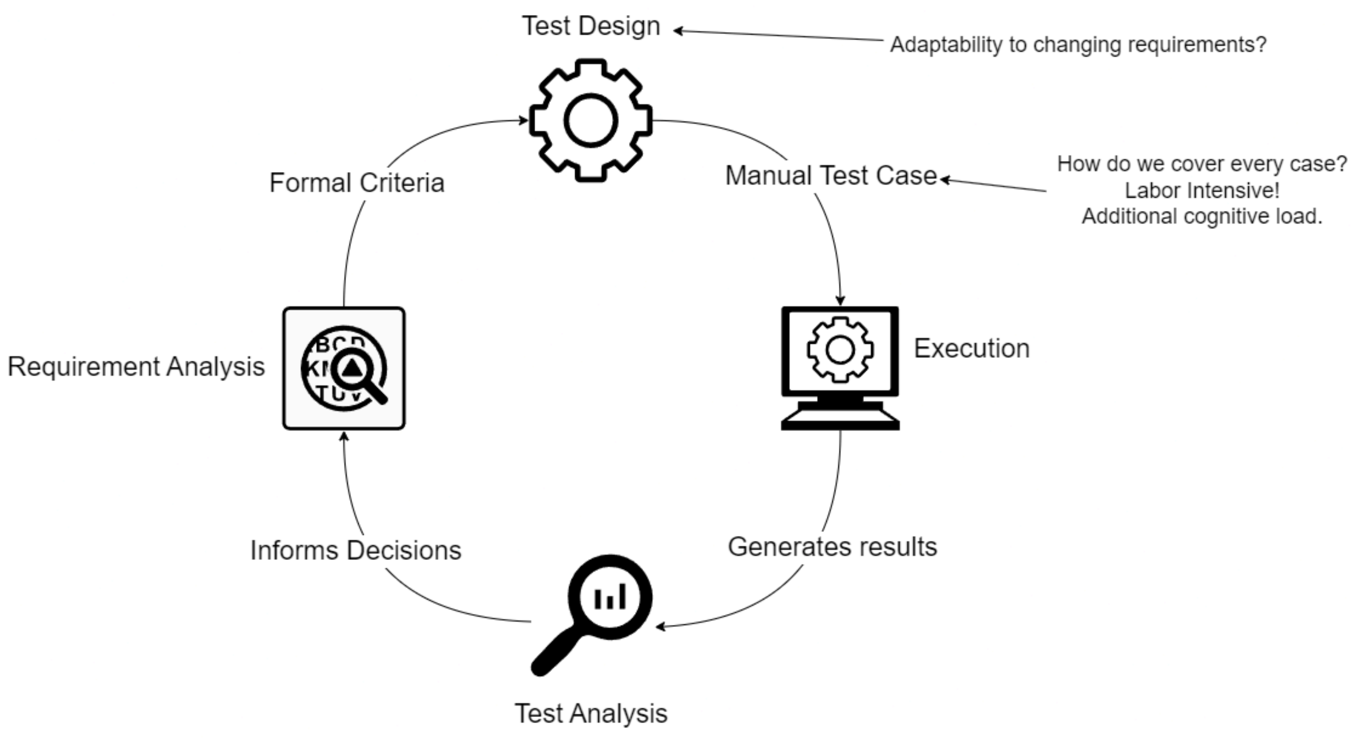


Fig 1.1 Validation flaws

Problem Statement Summary:

Western Digital's current validation workflow can be highly intensive due to its dependence on manually designed test cases. Traditional methods introduce bias, limit code/hardware coverage, and add a cognitive burden. This approach struggles to uncover all potential edge cases. It can also be slow to adapt to evolving NVMe requirements, ultimately slowing down the validation process while increasing the risk of faulty drives.

Solution Vision

To address the current challenges within the validation team's workflow, our capstone team proposes a comprehensive proof of concept that will utilize a random model simulation framework to create test cases generatively. This solution will expose an interface that allows test engineers to focus on the higher design level of engineering rather than the tedious nuances of translating key design aspects into code. Introducing computer pseudo-randomness into test case design allows us to explore a sweeping range of scenarios, uncovering hidden scenarios while minimizing human bias and improving validation efficiency.

Our solution will provide Western Digital with a way to enhance test coverage while reducing bias and increasing adaptability to new requirements, which may be a valuable tool for the NVMe validation industry (fig 1.2). This project will link a model simulator to an NVMe command interface, enforcing better test case consistency and a more process-oriented workflow. Automating the lower-level aspects of test engineering can effectively reduce engineers' cognitive load, enabling a more significant focus on more complex validation aspects.

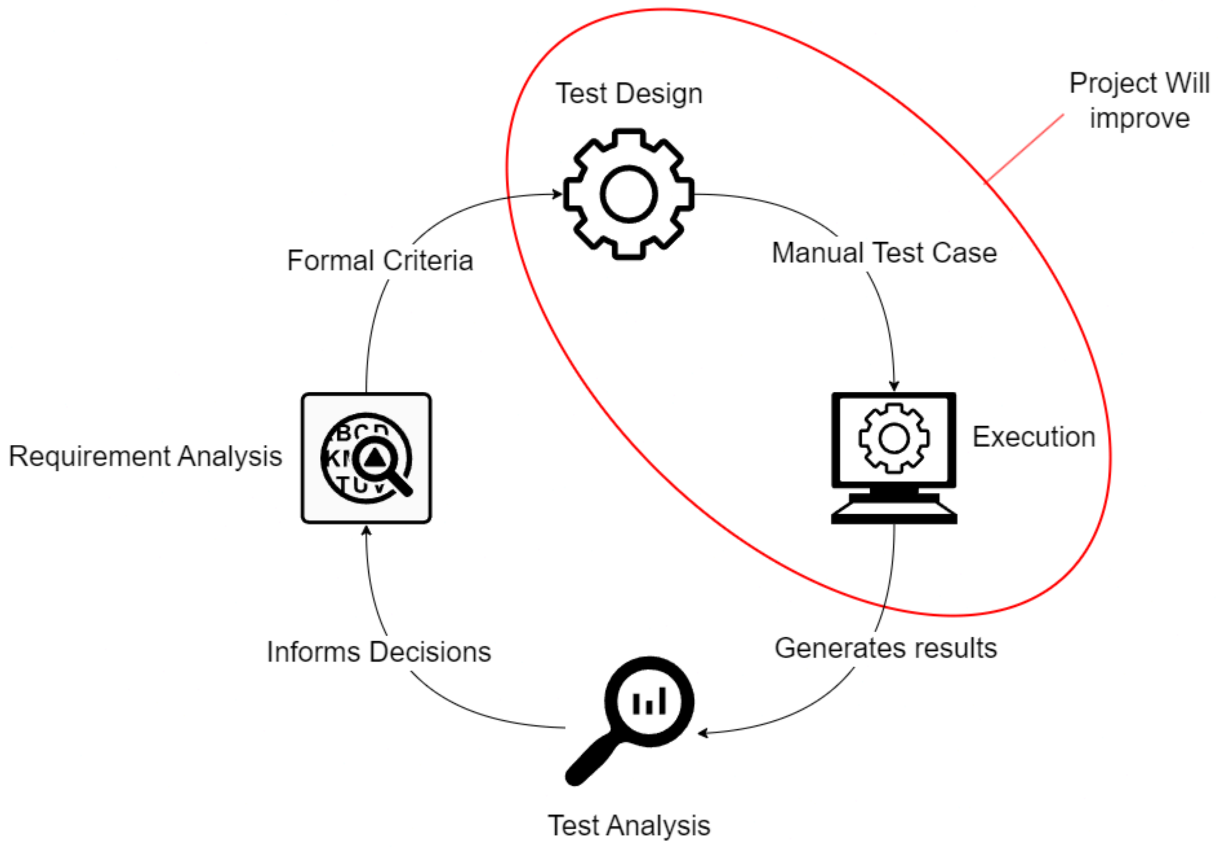


Fig 1.2 Validation flow solution

Solution Key Features:

- Automated Test Case Generation: Random motel simulation automatically creates test case sequences, ensuring a broader coverage of behaviors without explicit and manual definitions.
- Increased Edge Case Detection: Generative randomness allows the exploration of new combinations beyond what engineers traditionally consider, which helps reduce unexpected issues and boosts reliability.
- Reduced Engineer Workload: Engineers no longer need to create or adjust test sequencing by hand, which allows them to focus on harder and higher-level validation tasks that may be of higher priority.

- Dynamic Adaptation: As standards and requirements evolve, the system easily adjusts to the requirements. The interface to the defined system design will remain the same, but the backend code that directly interfaces with a command line interface can be extended or changed. This allows for one point of change or update rather than tedious manual refactoring and regressive changes.
- Improved Data Logging and Analysis: Because the system connects directly to the NVMe-cli, it can comprehensively log the results of commands driven by the model simulation. With systematic logging of these results, bugs can be traced, which can inform future product iterations.

How It Works (see fig 2.0 below):

- Input Data: The system begins by taking an NVMe specification file, written in TLA+, which defines how the system should work at a *design* level, where one only defines how components interact and possible state and state transitions on a very high level. This specification file will include existing and new requirements and industry standards to drive test sequences that align with such.
- Random Model Simulation: A core feature is a seeded random simulation algorithm that autonomously creates numerous test case sequences that run and explore only new state spaces that have yet to be explored.
- Test Execution: The generated test sequence will be executed automatically on NVMe drives. The program will then record the results for later analysis. Seed sampling will then be implemented to ensure repeatable results. If the algorithm detects an error, we resample the seed until it ends.
- Logging and Reporting: Results get captured via a modular logging component that captures data on each outcome, ensuring traceability. This logger will be customizable to the client's needs, such as only reporting certain levels of failure. Engineers will use these reports to evaluate product readiness.

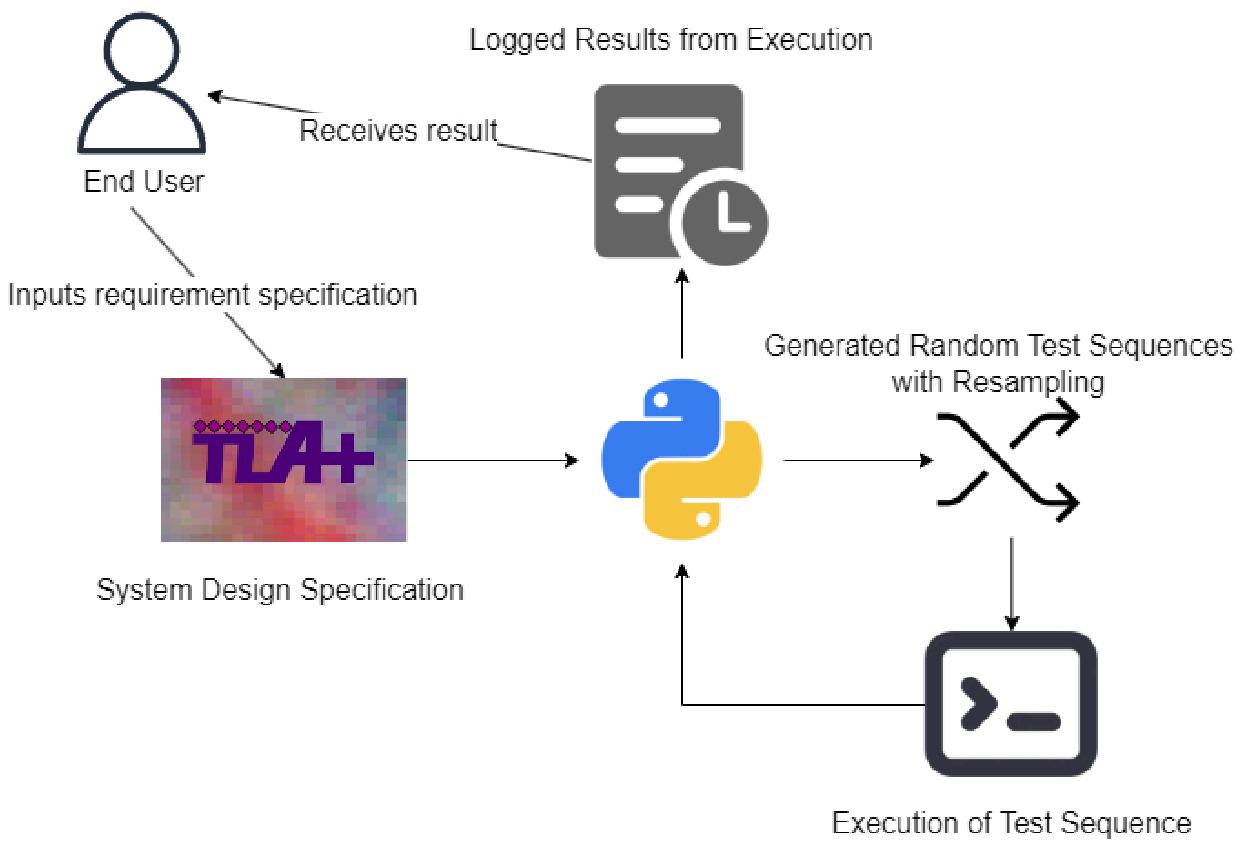

Logged Results from Execution
End User
Receives result
Inputs requirement specification
TLA+
System Design Specification
Generated Random Test Sequences with Resampling
Execution of Test Sequence

Fig 2.0 System diagram

Expected Outcomes and Predicted Benefits

- Enhanced Testing Efficiency: Automating this process saves valuable and expensive time for the engineering team by eliminating manual test sequence automation/creation.
- Improved Quality and Reliability: Random model simulation can detect issues that traditional methods may miss, resulting in better reliability.
- Adaptability and Future-Proofing: Abstracting a layer of the validation process improves the adaptability to new requirements and system changes by providing a consistent programmatic interface.
- Data-Driven Insights: Logging provides traceable results that can inform decision-making and improvement. In addition, consistent and predictable logging enables automated log parsing to be implemented on top of this project as needed.

Trade Offs

The system will likely induce an initial setup cost, which may require Wester Digital to adapt to a more generative, specification-driven workflow. This concept will also need a much larger focus on design-level thinking typical of model simulation rather than the type typically associated with programming. Further training on its use may be required when considering the way of thinking and possible future complexities relating to the project. Additionally, considering other solutions, like manually expanding test scenarios or preset model-based testing, may be more beneficial in the short term but sacrifice the long-term gains and elimination of inherent flaws this project provides. By contrast, after its short-term cost, this project will aim to provide maximum test coverage with minimal ongoing maintenance.

Solution Summary

In summary, our group envisions an efficient, robust, and more scalable approach to NVMe validation that will better ensure products meet the demands of a rapidly evolving industry. With this proof of concept, we stand to improve product reliability and the overall efficiency of traditional validation workflows.

Project Requirements

This section outlines the requirements for the **SSD Validation Proof of Concept**. This section will progressively deepen from high-level domain requirements to specific functional, performance, and environmental requirements.

Domain-Level Requirements

Domain-Level Requirements are the highest level requirement. Domain-Level Requirements are simply a brief overview at a high level of each Requirement.

1. Comprehensive NVMe SSD Validation: The program must facilitate exhaustive validation of NVMe drives by automating the generation, execution, and logging of test sequences. The generative nature of test sequences will cover large state spaces that uncover potential undiscovered edge cases.
2. Random Model Simulation: The program will utilize randomized simulation techniques that explore theoretical and possible NVMe behaviors to ensure robustness and reliability in testing methodologies and real-world use cases.
3. Traceability: The system must be able to trace back to where a particular point of interest occurred. Traceability will allow engineers to determine whether or not an event or state is worth diving further into. One such example would be the system discovering a bug, logging a trace of the bug, and rerunning the test to test the bug's repeatability.
4. Efficient test execution: The system must efficiently handle high volumes of test operations, ensuring timely feedback for validation through practical and efficient output parsing.

5. Scalability and versatility: To fully consider a wide range of NVMe behavior, the system must be adaptable and support a broad range of NVMe opcodes, including those from admin and IO passthrough commands. It must also be ready to easily refactor changes or updates in the interfaces provided by the NVMe controller. Similarly, the framework used to build the proof of concept must accommodate additional NVMe features or specifications without requiring significant architectural overhauls or large-scale code refactoring.
6. Drive Integrity and Consistency: As part of the validation process, the system must verify the consistency and predictability of R/W operations to ensure NVMe drives maintain integrity and proper behavior across testing and development scenarios.
7. Usability and Accessibility: This system is designed and intended only for validation engineers. It will provide a terminal interface similar to other Linux utility tools. The system will provide an accessible, straightforward, forward, and intuitive interface such that a new junior engineer can easily understand the purpose and function of the program.
8. Compliance and Standardization: The system must strictly adhere to NVMe specifications and any common best practices to ensure alignment with industry standards

Functional Requirements

Functional Requirements are the performance oriented requirements, so they will deal with expectations of what each feature or function of the project should do.

1. Validation Framework Features

- 1.1 Test Case Generation:
  Domains: (1, 2, 8)
  - The system will generativity create random test sequences based on a design-level model-simulation file that follows the NVMe industry standard specification.
  - The model simulation file will be model-checked for any design-level flaws before being consumed by the system.
  - The model-simulation file will contain the high level rules for every possible state combination and the resulting output, the output of which maps to real-world functions.

- Interpreting a compatible model-simulation file will output states and transitions between states, which will map model-simulation states to real-world validation functionality.
- Generative test sequence generation resulting from interpretation must comprehensively cover edge cases in large state spaces.
- The system will generate random seeds, determining how to generate test sequence samples.

- <u>1.2 Command Execution:</u>
  Domains: (1, 4, 6)
  - Will run all commands available in nvme-cli through admin and IO pass-thru commands, enabling low-level control
  - Additional functionality past what is provided in the nvme-cli is available through admin and IO pass-thru commands.
  - Different pass-thru functionalities should be presented with a consistent interface that maintains the same argument structure across various opcode functions.
  - Support all admin commands/opcodes (Get Features, Set Features, Identify)
  - Support all IO commands/opcodes (Write, Read, Flush)
  - Execution will have a watchdog timer feature that considers hanging processes.
  - Upon reaching the end of a state space path, the system will attempt to explore a new path.
  - If no state space path is available, the system will generate a new state space.
- <u>1.3 Result Evaluation:</u>
  Domains: (3, 4, 6, 8)
  - Provide comprehensive output from the result of running a command within a test sequence
  - Parse and evaluate results from NVMe CLI to identify anomalies or unwanted behavior
  - The output of a command will indicate whether or not the NVMe runs successfully from stderr.
    - The program may use this to implement granular exception handling or flagging of errors.
  - The output of a command will also return any resulting data or output from stdout.
  - The command will also indicate the exit code, which will be used in further verification functions.

2. Simulation Capabilities

- <u>2.1 Randomized Generation:</u>
  Domains: (1, 2)
  - The system will randomly create and store a seed, determining later randomness.
  - The system will use seeds to generate random test sequence states based on the constraints interpreted from the model simulation file.
  - Computer pseudorandomness avoids human biases from differing backgrounds and will provide a more process-oriented approach.
  - Upon exploring a state space and reaching a dead end, the system will generate a new seed to sample test sequences from
- <u>2.2 State Exploration:</u>
  Domains: (1, 2, 4)
  - The exploration pattern of this system reflects that of a depth-first search pattern in a directed graph
  - Exploring depth-first will explore unique and new states more quickly, avoiding similar, already explored states.
  - Reaching the end of a state path can be represented as the end of a test sequence path in terms of finding unique cases
  - States that differ from real-world expectations can be used to determine anomalies
- <u>2.3 Resampling Capabilities:</u>
  Domains: (1, 3, 7)
  - The system implements traceability and resampling in randomness through storing and utilizing a seeded generation technique, remembering the currently used seed.
  - The system can resample the same seed previously stored to attempt to recreate the previously explored issue, ensuring repeatability.
  - If the system encounters a bug or anomaly before the end of a state path is reached, resampling will occur.
  - Resampling does not ensure the same path execution.
- <u>2.4 Bridge between model simulation and real-world:</u>
  Domains: (1, 2, 4)
  - State space model simulation will drive the system.

- The program will intercept outputs from the model simulation and implement a blocking call to another interface.
- The model simulation shall not run until the blocking call returns.
- Additionally, the system will contain an interface that maps model simulation output to callable functions and passable arguments.

3. Data handling and reporting

- 3.1 Data Integrity Checks:
  Domains: (6, 8)
  - Data from test execution will be analyzed to ensure that the functionalities manipulate and read data as expected when developing new features.
- 3.2 Reporting and Logging:
  Domains: (3, 4, 7, 8)
  - CLI will output a hex dump that follows NVMe specifications
  - The program will provide functions to translate hex dumps to human-readable data where needed using NVMe specifications, particularly for critical errors or functionalities.
  - The result must be traceable so an engineer can recreate the error.
  - Logging functionality will efficiently consume all returned data without significantly impacting the main program's performance.
  - An engineer must understand NVMe hex dumps following NVMe specifications that fall outside the program's translations.

4. Accessibility and Maintainability

- 4.1 Modular Design:
  Domains: (5, 7)
  - The program will implement a modular and layered design.
  - The program will have four main modules: TLA interpreter, nvme interface, logging, and a bridging model that contains the logic that connects the three modules.
  - The bridging module will use functions exposed in the Interpreter, Interface, and logger module.

  - The decoupled nature allows for different code so long as the exposed interface remains the same or the consuming functions adapt to new interfaces.
- <u>4.2 User Interface:</u>
  Domains: (7)
  - The program will present a user interface similar to the IEEE Linux utility conventions (IEEE Std 1003.1-2017).
  - The program will contain a customizable logger that reads off a config file to determine how to present and parse information.
  - The interface will cater to both new and experienced test engineers.
  - The user interface will contain a guide manual that can be prompted or appear due to wrong usage.
  - The user interface will contain options determining how and what the program runs.
  - The interface will abstract the system's inner workings enough to allow test engineers to focus on more complex validation functions while providing enough control to carry out explicit validation functions.
  - The program will provide better control by enabling users to pass arguments over default parameters.
- <u>4.3 Documentation:</u>
  Domains: (7)
  - The delivered product will contain hierarchical documentation that presents surface-level functions and interfaces and then dives into the deeper functions that the surfaces use.
  - For conceptual and high-level documentation, depictions and diagrams are used as needed.
  - Lower-level program documentation will use automated documentation to enforce consistency and increase usability and predictability.
  - The source code will have comments on lines of code specific enough to extend past the scope of higher-level documentation.
  - Commonly used functions will have a high-level description of the function as a docstring or function.
  - The documentation will describe each file and its purpose.
- <u>4.4 Tech Debt and Limitations</u>
  Domains: (7)

- The proof of concept will be simplified to meet deadlines and develop past a certain degree of entropy and expectations, which likely will only partially align with the validation team's needs.
- The proof of concept allows for the replication of anomalies. However, it will not guarantee deterministic execution paths.
- The proof of concept relies on external tools and libraries, which may depreciate functions being utilized and require some degree of maintenance.
- This project differs in some features that fall outside the scope of the expectations of this project that would significantly enhance its useability, such as AI simulation tuning, state-space visualization, automated output analysis, and Jira API integrations.

Performance Requirements

Performance Requirements are the testable requirements that the system is expected to meet in 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 60 seconds per log.
    - Logging from execution should not cause significant execution overhead (<500 milliseconds)
    - The system will loop indefinitely until the end of the execution path is found.
2. **System Responsiveness:**
    - User interactions should have a response time (e.g., from the user submitting input to the terminal responding) of less than 3 seconds; for functions that take longer, the program must indicate progress or loading.
    - If the blocking interval for any opcode exceeds 60 seconds, the system shall automatically proceed to the next opcode, verified by monitoring the elapsed time of the opcode execution for each call.
    - Our system will return a log after an error within 10 seconds.
3. **Accuracy:**
    - The program will detect and flag anomalies with nearly complete accuracy, double-checked through resampling.
4. **Resource Efficiency:**

- Optimize CPU and memory usage to ensure smooth operation even under heavy test loads.

Environmental Requirements

Environmental Requirements are the expected requirements of the softwares and technologies being used in the project.

1. **Hardware Constraints:**
   - The system must be compatible with Linux-based systems containing a kernel version 6.8 or later.
   - Must support NVMe SSDs conforming to the NVMe 1.4 or later specification.
   - There must be a secondary target NVMe SSD
   - RAM and CPU must be capable of virtualization and high throughput and I/O operations
   - The motherboard must have current NVMe compatibility.
   - The server must have internet/internal network accessibility.
   - The testing server must be located where there is consistent power and internet.
2. **Software Dependencies:**
   - The system must run the 24.04 LTS Ubuntu Server Distribution.
   - The framework must integrate with NVMe CLI and Python-based tooling.
   - KVM+libvirt support is required for sandboxing and portability.
   - The project will need Pluspy to interpret TLA files.
   - The testing environment will need the latest nvme-cli installed for the project to work.
   - This project will be 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 and compliance.

Environmental Setup Requirements

Hardware Requirements

1. **Server Specifications**

  - CPU: Virtualization-capable processor
  - RAM: Sufficient for high I/O operations (minimum 16GB recommended)
  - Storage: NVMe SSD for testing
  - Motherboard: NVMe-compatible with PCIe slots
  - Network: Reliable internet/internal network connection
  - Power: Stable power supply with UPS recommended
2. **Testing Environment**
   - Primary NVMe drive for system
   - Secondary NVMe drive for testing
   - Proper cooling and ventilation
   - Physical security measures

Software Requirements

1. **Operating System**
   - Ubuntu Server 24.04 LTS
   - Kernel version 6.8 or later
   - KVM+libvirt support
2. **Required Software**
   - Python 3.12
   - NVMe-CLI (latest version)
   - PlusPy framework
   - Required Python dependencies
   - Git for version control

User Stories and Use Cases

Test Engineer User Stories

Test Creation and Automation

- As a tester, I want to automate the test creation process to save time.
  - Acceptance Criteria:
    - Can generate test sequences automatically
    - Reduces manual intervention in test creation

        - Provides feedback on test generation progress
        - Saves test configurations for future use
- As a tester, I want to be able to rerun tests to recreate errors.
    - Acceptance Criteria:
        - Can reproduce test sequences through resampling old seed
        - Maintains consistent test runs
        - Logs traceable steps for reconstruction
        - Verifies reproduced conditions

Hardware Utilization

- As a tester, I want to be able to utilize hardware fully.
    - Acceptance Criteria:
        - Utilizes available resources and attempts to maximize nvme throughput
        - Provides hardware usage statistics
- As a tester, I want multiple commands to run automatically to save time.
    - Acceptance Criteria:
        - Executes command sequences without intervention
        - Handles dependencies and prereqs between commands
        - Reports progress execution
        - Appropriately manages timeouts

Test Customization

- As a tester, I want to customize tests to test specific functionality.
    - Acceptance Criteria:
        - Target testing through specification file
        - Parameter customization
        - Enables focused testing of specific components

TLA+ Integration

- As a tester, I want to use TLA+'s full functionality to test various functionalities.
    - Acceptance Criteria:
        - Supports all relevant TLA+ operators
        - Enables complex state machine definitions
        - Allows temporal property specification
        - Integrates with TLA+ tools seamlessly

- As a tester, I want to test with TLA+ files to ensure our hardware complies with the NVMe standard.
    - Acceptance Criteria:
        - Validates against NVMe specifications
        - Reports compliance violations
        - Provides traceability to standards
        - Supports multiple specification versions

Error Handling and Output

- As a tester, I want to review errors generated to fix problems with the tested hardware.
    - Acceptance Criteria:
        - Provides detailed error reports
        - Shows error context and conditions
        - Enables error categorization
        - Supports error pattern analysis
- As a tester, I want the readable human output to review results quickly.
    - Acceptance Criteria:
        - Presents clear, formatted output
        - Includes relevant test details
        - Provides summary views
        - Supports different output formats
- As a tester, I want to limit the output to errors so that I can focus on what went wrong.
    - Acceptance Criteria:
        - Allows filtering by error types
        - Supports custom output levels
        - Enables focused error review
        - Provides error-only reports

Usability and Standards

- As a tester, I want an easy way to interact with the program to spend more time testing and verifying and less time setting up my testing environment.
    - Acceptance Criteria:
        - Provides an intuitive command-line interface
        - Minimizes setup requirements
        - Includes clear documentation
        - Offers quick start guides

- As a tester, I want to follow industry standards to align with best practices.
    - Acceptance Criteria:
        - Encourages development alignment with NVMe specifications
        - Maintains compatibility with other related tools
        - Updates with industry changes
- A user is expected to be able to use the system through the command line.
    - Acceptance Criteria:
        - Provides CLI functionality
        - Supports standard CLI conventions
        - Includes help documentation
        - Enables scripting and automation

Use Cases

UC1: Automated General Testing

Primary Actor: Test Engineer

Preconditions:

- System configured
- Valid TLA+ specification

Main Flow:

1. Engineer selects TLA+ specification
2. System generates test sequences
    a. System begins running sequence
    b. System outputs sequence commands
3. System logs results
4. Engineer reviews logs

Alternative Flows:

- A1: Invalid specification
- A2: Device not responding
- A3: Execution timeout
- A4: Resource constraints

UC2: Error Analysis

Primary Actor: Test Engineer

Preconditions:

- Test execution completed

- Error logs available

Main Flow:

1. Engineer selects logs
2. Engineer processes log data via parsing techniques
3. Engineer identifies error patterns
4. Engineer generates error report

Alternative Flows:

- A1: Log file corruption
- A2: Insufficient error data

UC3: Result Analysis

Primary Actor: Test Engineer

Preconditions:

- Test execution completed
- Results data available

Main Flow:

1. Engineer selects logs
2. Engineer processes test results
3. Engineer/Logging generates analytics
4. Engineer presents findings
5. Engineer exports results

Alternative Flows:

- A1: Insufficient data
- A2: Analysis error

Interaction Scenarios

Scenario 1: New Feature Validation

1. Engineer creates TLA+ specification for new feature
2. Generates test sequences
3. Executes tests on target device
4. Analyzes results for compliance
5. Documents findings

Scenario 2: Regression Testing

1. Engineer inputs specification file that tests for known bug
2. Executes regression tests

3. Compares results with baseline
4. Reports any regressions

Scenario 3: Bug Investigation

1. Engineer receives bug report
2. Reproduces issue through tracing
3. Analyzes error conditions
4. Modifies test parameters
5. Verifies fix effectiveness

Scenario 4: Performance Analysis

1. Engineer configures performance metrics
2. Executes performance test focused specification files
3. Collects performance data
4. Analyzes performance
5. Generates performance report
6. Success Metrics

Efficiency Metrics

- Execution time within 24 hours for 10,000+ admin commands
- Log processing time < 500 milliseconds per log entry

Quality Metrics (Optimistic)

- reproducibility of test sequences with same seed
- full coverage of a specified state space on state machine completion
- <1% false positive rate for error detection

Usability Metrics

- <30 minutes training or self teaching time for basic use
- <5 steps to generate and execute tests
- <10 steps to set up tool
- <10% likelihood of modifying code to run basic testing functionalities within NVMe

Potential Risks

As with any software development project, there are a few risks to take into account. Our biggest risk for our project is getting the connection between PlusPy and NVMe-CLI completely working, since the project requires the use of both technologies. If this doesn't work, the entire project will have to be reenvisioned with the guidance of our client. Another big potential risk is the TLA+ to PlusPy bridge, as this will also prevent the project from continuing, since TLA+ is a very hard requirement of the client, and will prevent us from continuing the project. TLA+ is very important as our client has test engineers who write TLA files to test NVMe drives. Mitigating these risks are more client specific, as we cannot just change the project without consulting and showing the client that these would not work. These are considered high risk, but low likeliness in terms of potential.

The next biggest risk, in the medium risk category would be the logging and opcode verification. The logging part of the project is considered to be a lower risk likeliness, as the logging uses file I/O which shouldn't be too hard, just taking the contents of the terminal and saving it. The opcode verification is considered higher risk likeliness, as we need to use the NVMe-CLI to make sure that each admin passthrough command creates the exact same output as the regular non-admin command. Mitigation for these risks can be fixed by us, and would likely not require client intervention.

The last category of risks are the ones that are both low risk and low likeliness. The biggest risk in the low risk category is seeding for reproducibility. It is considered to have a low likelihood of failure.

Some non software development risks can also affect our project, for example, hardware failure or unexpected behavior of hardware. A few terrible things that could happen in this category are firmware corruption, Operating System failure, or 'server' hardware component failure (like CPU, RAM, Motherboard, Power Supply). Things that aren't as awful, but could still happen are failures like kernel level crashing (especially since we had to modify and recompile to get hardware passthrough working), power loss, power surges, or opcode incompatibility. Mitigation for some of these risks are preventable, like with hardware failure, we can monitor components and look out for signs of failure. Risks like power loss can be mitigated with a UPS(uninterruptible power supply), but the server does not have to be on 24/7, and development can mostly continue unhindered. Power surges can be prevented with a simple surge protector (if it isn't already, as the university power isn't always the 'cleanest'). Mitigation for risks like Operating System failure and kernel level crashing can be mostly prevented with regular backups, although we have documented how to recreate our current setup if even the backups have failed, allowing for some down time while getting the development server fully fixed and returned to working order.

When evaluating risk, we decided to rank each one between 0 and 10, depending on how likely and how severe each risk is, and then plot on the table below.

Risk Categories and Mitigation Strategies

High Risk Items

1. **PlusPy to NVMe-CLI Integration**
    - Risk Level: High
    - Impact: Critical
    - Mitigation: Early proof-of-concept testing, regular client consultation
2. **TLA+ to PlusPy Bridge**
    - Risk Level: High
    - Impact: Critical
    - Mitigation: Thorough testing of interpretation layer, fallback plans

Medium Risk Items

1. **Logging System**
    - Risk Level: Medium
    - Impact: Moderate
    - Mitigation: Modular design, multiple logging options
2. **Opcode Verification**
    - Risk Level: Medium
    - Impact: Significant
    - Mitigation: Comprehensive testing suite, verification protocols

Low Risk Items

1. **Seeding Implementation**
    - Risk Level: Low
    - Impact: Moderate
    - Mitigation: Robust random number generation, proper seed management
2. **Hardware/System Risks**
    - Risk Level: Low to Medium
    - Impact: Various
    - Mitigation: Regular backups, UPS, surge protection, monitoring systems

	Low Severity (0-1)	Medium Severity (2-3)	High Severity (4-5)
Low Likeliness (0-1)	Reproducible Seeding (2) Power Loss (1) Power Surge (1)	Kernel Level Crashing (4)	Server Hardware Failure (6) Firmware Corruption (4) Operating System Failure (5)
Medium Likeliness (2-3)		Logging Output (5) Opcode Compatibility (6)	PlusPy to Python Software Dev (7)
High Likeliness (4-5)		Opcode Validation (8)	

Fig 4.0 Risk Severity Table

Project Plan

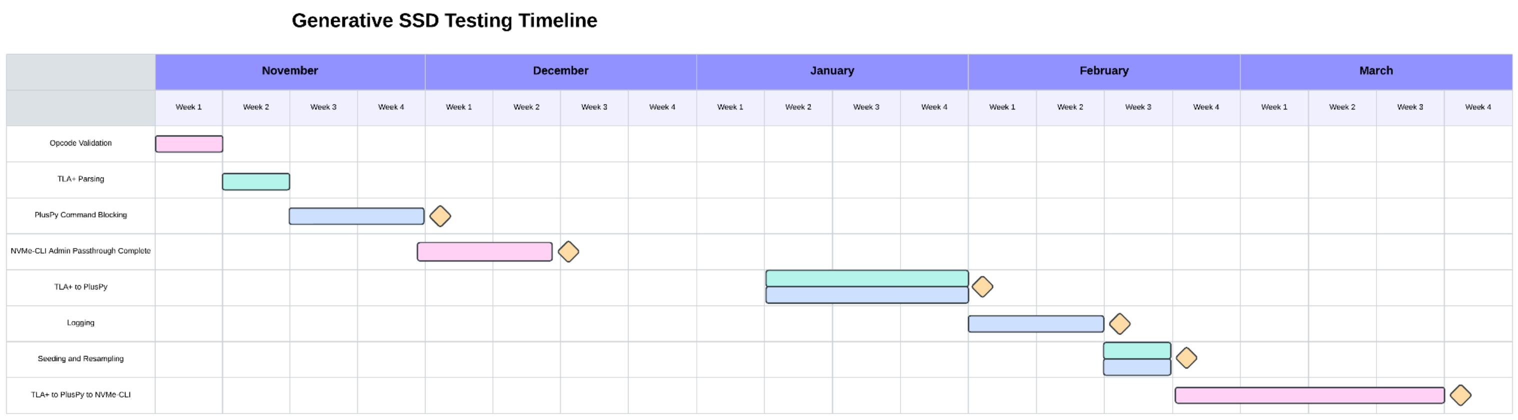


Fig 4.0 Development Timeline (See Appendix A for larger version)

The overall development timeline, with a couple of months cut out to over generously estimate our time, is set for a few tasks. Each color is coded to a separate focus, Pink is focused on NVMe-CLI based development, Blue is focused on PlusPy style development and Green is focused on TLA+ development. The yellow diamond is things that have not been actively worked towards yet, and are also intended to be deadlines for each section.

Conclusion

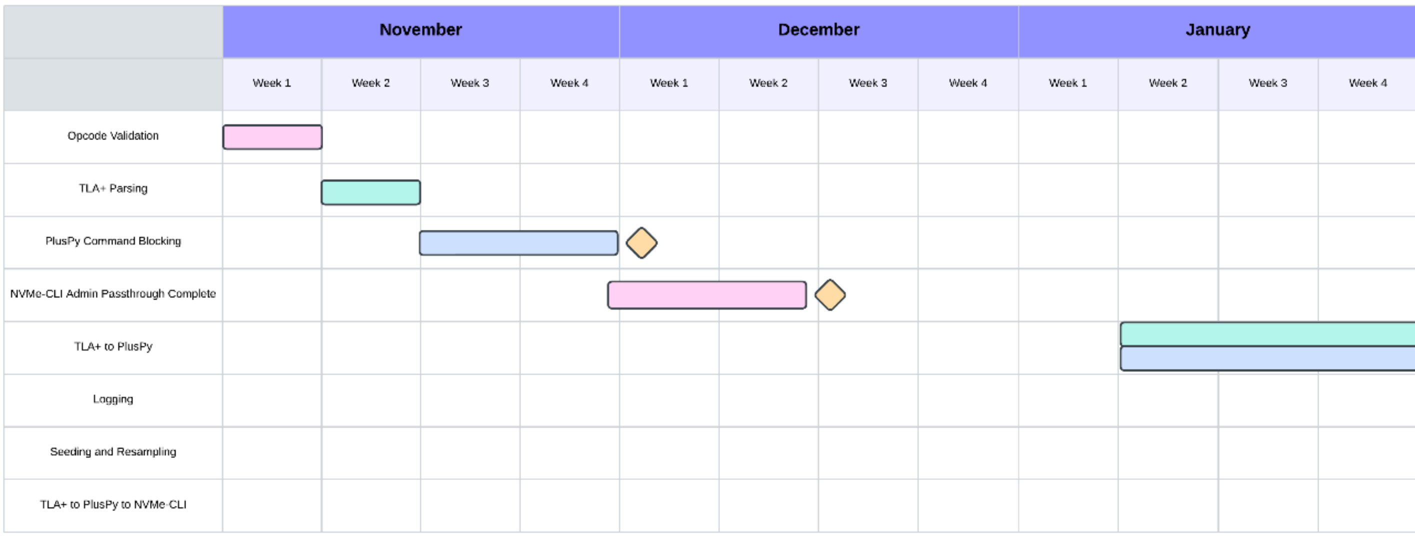
In summary, this requirements document is a blueprint to help Western Digital's NVMe validation team overcome key challenges. The goal is to improve the efficiency, coverage, and flexibility of testing NVMe solid-state drives by shifting to an automated framework that uses random model simulation. This approach will reduce the need for manual test creation, decrease bias, and increase coverage, especially in complex cases that are often missed by traditional methods.

We've identified main issues in the current workflow, such as limited coverage, high demands on engineers, and slow adaptation to new requirements. Our solution is a shift to generative testing using automation, which will make test sequences more consistent and thorough, allowing engineers to focus on more complex tasks.

This document outlines the project's scope, including key requirements for functionality, performance, and environment. Each section highlights our commitment to creating a scalable and adaptable testing framework that ensures reliable validation of NVMe drives. With this solution, we aim to boost quality, reliability, and efficiency, allowing Western Digital to speed up product readiness while upholding high standards.

Appendix A: Development Schedule

Generative SSD Testing Timeline


November
December
January
Week 1
Week 2
Week 3
Week 4
Opcode Validation
TLA+ Parsing
PlusPy Command Blocking
NVMe-CLI Admin Passthrough Complete
TLA+ to PlusPy
Logging
Seeding and Resampling
TLA+ to PlusPy to NVMe-CLI