ReLiANT

Software Design Document

February 26, 2026

For

SanDisk

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

Version 2.0

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.

Implementation Overview

Solution Context

ConTiNGENT (Convert TLA+ into NVMe Generative Test) is a specification-driven testing framework designed to generate executable NVMe validation sequences directly from formal TLA+ models. The system evolves prior proof-of-concept work performed under Western Digital into a production-oriented architecture suitable for integration within SanDisk's SSD validation workflow.

The primary implementation objective is to replace unreliable Python-based simulation (PlusPy) with a deterministic, scalable, and reproducible generative testing pipeline built around:

- TLC model checker for authoritative state-space exploration and trace generation
- Rust-based execution layer for safe, performant interaction with NVMe hardware via NVMe-CLI
- Instrumentation conventions linking abstract TLA+ states to concrete NVMe device behaviors

This approach ensures generated test sequences are:

- Specification-valid rather than randomly fuzzed
- Replayable for deterministic bug reproduction
- Broad in coverage through nondeterministic state exploration
- Resource-efficient for DUT utilization

Together, these characteristics position ConTiNGENT as a candidate for long-term inclusion in enterprise SSD validation pipelines.

High-Level Implementation Strategy

The system will be implemented as a multi-stage generative testing pipeline that transforms formal specifications into executable NVMe command sequences and validation outcomes.

Stage 1 - Formal Model Processing

TLA+ specifications describing NVMe protocol behavior are authored, instrumented, and verified using TLC. TLC's generate mode produces execution traces representing valid nondeterministic behaviors of the modeled system.

Key technical decision: TLC over PlusPy

- Native support for nondeterminism
- Proven correctness and ecosystem adoption
- Deterministic trace generation suitable for replay

Stage 2 - Trace Translation Layer

Generated TLC traces are parsed and transformed into an intermediate executable representation describing:

- NVMe command ordering
- Parameterization
- Expected state transitions
- Validation checkpoints

This layer isolates formal semantics from hardware execution, enabling:

- Reproducibility
- Extensibility to additional protocols
- Safer validation before DUT interaction

Rust is selected for this layer due to:

- Memory safety without garbage collection
- Strong type guarantees for protocol modeling
- High-performance parsing and transformation
- Native ecosystem compatibility with systems tooling

Stage 3 - NVMe Execution Engine

Executable traces are issued to the Device Under Test using NVMe-CLI bindings orchestrated through Rust.

Responsibilities include:

- Command sequencing and timing control
- Result capture and normalization

- Error detection and logging
- Deterministic replay of failing traces

Rust is chosen over Python to ensure:

- Predictable low-level system interaction
- Concurrency safety for parallel test execution
- Production-grade performance compatible with enterprise validation

---

Stage 4 - Result Analysis and Retest Automation (Stretch Goal)

The final stage evaluates DUT behavior against expected trace outcomes.

Planned capabilities:

- Pass/fail classification
- Failure trace persistence
- Automatic deterministic retesting
- Diagnostic artifact generation

This enables closed-loop generative validation rather than one-time execution.

Architectural Direction and Constraints

Architecture Style

The system follows a pipeline-oriented modular architecture with clear separation between:

- Formal specification domain
- Trace transformation logic
- Hardware execution layer
- Result analysis subsystem

This separation:

- Minimizes cross-layer coupling
- Enables independent validation of each stage
- Supports future scalability and reuse

Technology Stack

Layer	Technology	Rationale
Formal modeling	TLA+, PlusCal, TLC	Industry-recognized formal verification tooling
Transformation	Rust	Safety, performance, systems-level control
Hardware interface	NVMe-CLI + Linux	Direct protocol-level SSD interaction
Execution environment	Ubuntu + OpenJDK	TLC compatibility and stable tooling
Development tooling	VSCode + Rust-Analyzer + TLA+ extensions	Integrated formal + systems workflow

---

Environmental Constraints

- Requires PCIe-attached NVMe SSD DUT
- Linux-based execution environment (Ubuntu)
- Compatibility with latest NVMe Base Specification
- Deterministic reproducibility of TLC traces
- Secure and safe command execution against physical hardware

These constraints directly inform architectural and implementation decisions.

Implementation Scope and Evolution

ConTiNGENT is designed for incremental maturation:

- Core generative pipeline
    - TLC trace generation
    - Rust trace translation
    - NVMe command execution

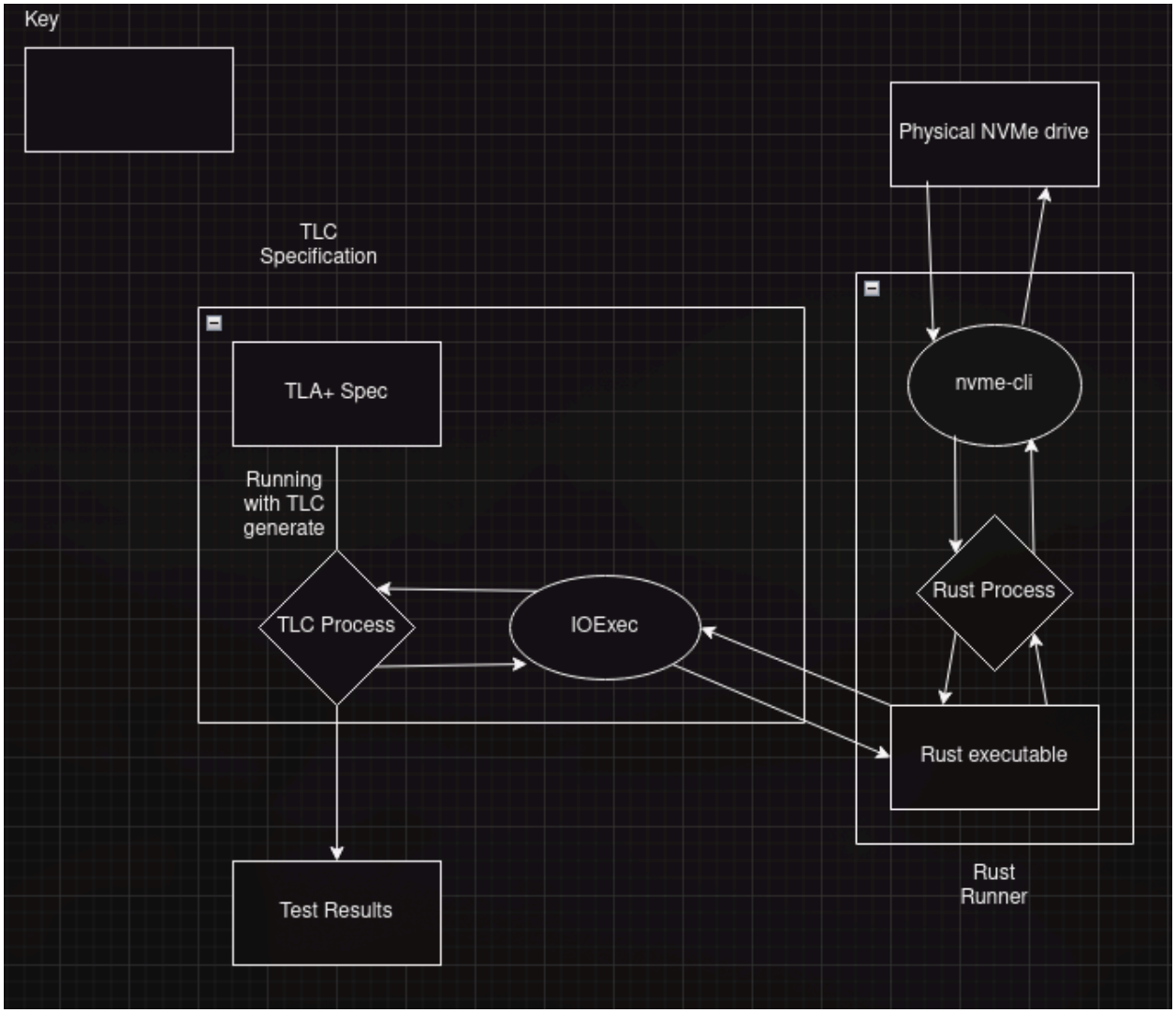
- Stabilization
    - Deterministic replay
    - Logging and diagnostics

    - Validation correctness

- Advanced capabilities
    - Automated retesting
    - Parallel execution
    - Integration into enterprise validation workflows

This staged evolution ensures early functional delivery while supporting long-term scalability.

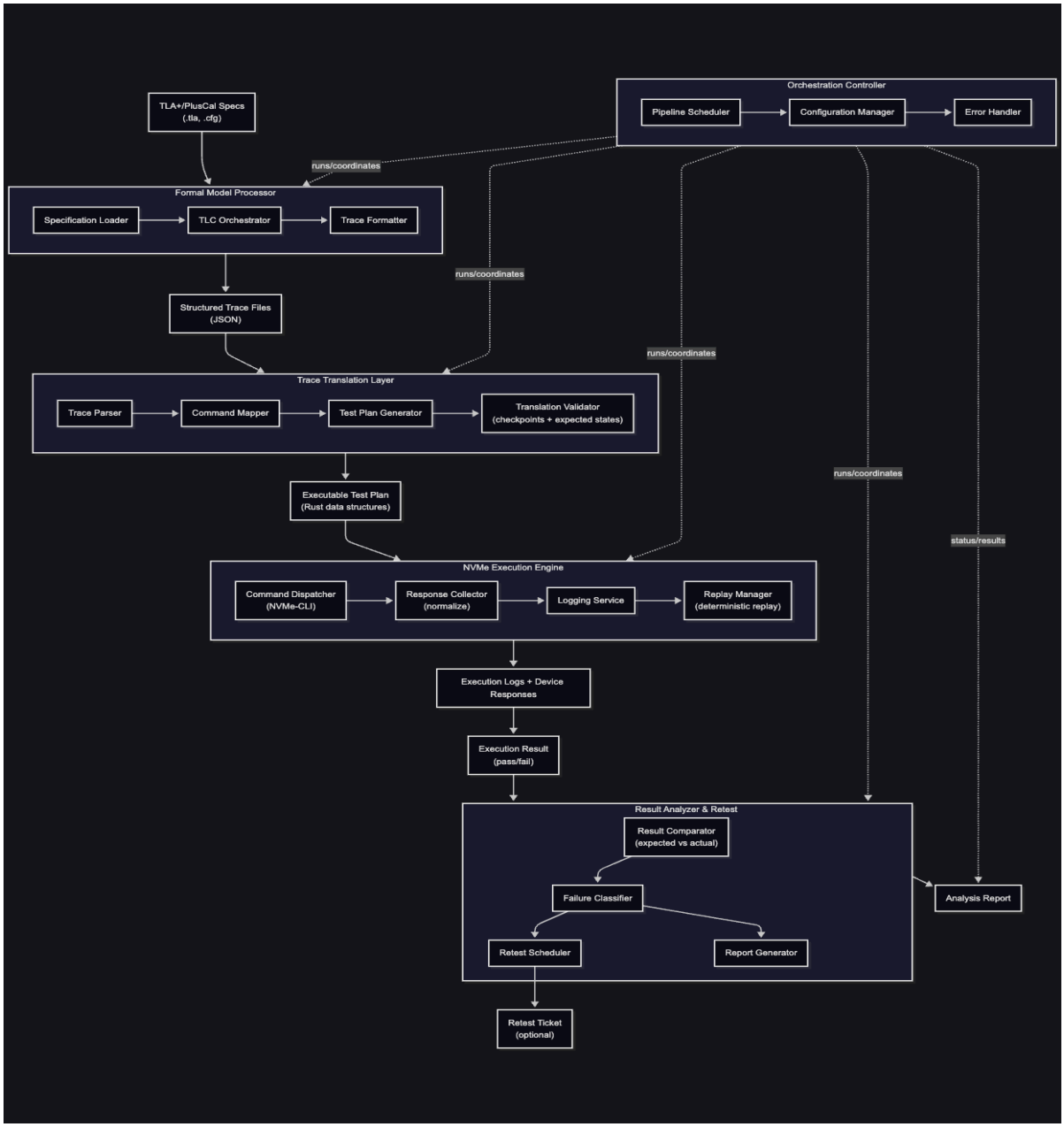
Architectural Overview



Component Level-Design

ConTiNGENT – Generative NVMe Validation Pipeline

This section describes the internal structure of ConTiNGENT at the component level. Each component is defined by its **responsibilities**, **boundaries**, **public interfaces**, and **role within the overall pipeline**. Responsibilities are explicitly partitioned to avoid overlap and to ensure deterministic behavior, testability, and long-term maintainability.



1. Formal Model Processor

The Formal Model Processor is responsible for **all formal specification handling**. It transforms TLA+/PlusCal specifications into concrete execution traces that represent **valid nondeterministic behaviors** of the NVMe protocol. This component is the **sole owner of formal reasoning** within the system.

It operates at the **model level only** and has no knowledge of NVMe commands, hardware, or execution semantics.

Boundaries

- **Inputs:**

    - TLA+ specification files (`.tla`)
    - TLC configuration files (`.cfg`)
- **Outputs:**
    - Structured execution traces in a machine-readable format (JSON)
- **Explicit Exclusions:**
    - No NVMe command generation
    - No device interaction
    - No execution timing or replay logic

Internal Structure (Conceptual)

- **Specification Loader:** Validates and parses TLA+/PlusCal models.
- **TLC Orchestrator:** Executes TLC to explore the model state space.
- **Trace Formatter:** Converts raw TLC output into structured trace artifacts.

Public Interface

- `generate_traces(spec_path, config_path) → List<Trace>`
  Produces nondeterministic execution traces from a validated specification.
- `validate_spec(spec_path) → bool`
  Ensures the specification is syntactically and semantically valid before execution.

Interaction Pattern

Other components **do not call TLC directly**. All formal exploration is encapsulated here, and downstream components consume only the generated trace artifacts.

2. Trace Translation Layer

The Trace Translation Layer bridges the gap between **abstract protocol behavior** and **concrete executable tests**. It translates formal traces into deterministic NVMe command sequences while preserving the semantics of the original model.
This component is the **sole owner of abstract-to-concrete mapping logic**.

Boundaries

- **Inputs:**
    - Structured execution traces (JSON)

- **Outputs:**
  - Executable test plans represented as Rust data structures
- **Explicit Exclusions:**
  - No hardware access
  - No direct NVMe command execution

Internal Structure (Conceptual)

- **Trace Parser:** Loads and validates structured trace files.
- **Command Mapper:** Maps abstract TLA+ actions to concrete NVMe operations.
- **Test Plan Generator:** Produces ordered, replayable execution plans.
- **Translation Validator:** Inserts checkpoints and expected states for validation.

Public Interface

- `translate_trace(trace) → TestPlan`
  Converts a formal trace into a deterministic execution plan.
- `serialize_test_plan(plan) → bytes`
  Enables persistence and transport of test plans.
- `deserialize_test_plan(data) → TestPlan`
  Restores a previously generated test plan.

Interaction Pattern

The Trace Translation Layer consumes traces only and produces test plans only. It does not interpret execution results or interact with devices.

3. NVMe Execution Engine

The NVMe Execution Engine is responsible for **all hardware interaction**. It executes test plans against a physical NVMe Device Under Test (DUT), captures responses, and ensures deterministic replay.
This component is the **only module allowed to interact with NVMe hardware**.

Boundaries

- **Inputs:**
  - Executable test plans
- **Outputs:**
  - Execution logs
  - Normalized device responses
  - Pass/fail execution results
- **Hardware Interaction:**
  - Direct communication with the DUT via NVMe-CLI

Internal Structure (Conceptual)

- **Command Dispatcher:** Issues NVMe-CLI commands.
- **Response Collector:** Normalizes and captures device responses.
- **Logging Service:** Persists execution artifacts.
- **Replay Manager:** Enables deterministic replay of failing tests.

Public Interface

- `execute_test_plan(plan) → ExecutionResult`
  Executes a test plan against the DUT.
- `replay_failing_test(test_id) → ExecutionResult`
  Replays a previously failing execution deterministically.
- `get_execution_log(test_id) → Log`
  Retrieves execution artifacts for analysis.

Interaction Pattern

The engine does not interpret correctness. It executes, records, and replays—nothing more.

4. Result Analyzer & Retest Automation (Stretch Goal)

This component evaluates execution outcomes and enables **closed-loop validation**. It compares observed behavior against expected results and determines whether failures warrant retesting.
It owns **all correctness evaluation and failure classification logic**.

Boundaries

- **Inputs:**
    - Execution logs
    - Expected states embedded in test plans
- **Outputs:**
    - Analysis reports
    - Failure classifications
    - Optional retest tickets
- **Explicit Exclusions:**
    - No hardware interaction
    - No modification of test plans

Internal Structure (Conceptual)

- **Result Comparator:** Compares expected vs. actual behavior.
- **Failure Classifier:** Categorizes failures (timeouts, mismatches, violations).
- **Retest Scheduler:** Queues deterministic retests.
- **Report Generator:** Produces structured reports.

Public Interface

- `analyze_results(log, expected) → AnalysisReport`
- `schedule_retest(report) → RetestTicket`
- `generate_summary_report(batch_id) → Report`

5. Orchestration Controller

The Orchestration Controller coordinates all pipeline stages and provides a **single operational interface**. It owns **control flow**, not domain logic.

Boundaries

- **Inputs:**
    - User commands
    - Configuration files
- **Outputs:**

    - Pipeline status
    - Final reports

- **Explicit Exclusions:**
    - No formal modeling
    - No trace translation
    - No hardware interaction

Internal Structure (Conceptual)

- **Pipeline Scheduler:** Controls execution order.
- **Configuration Manager:** Loads runtime and DUT settings.
- **Error Handler:** Manages failures and retries.

Public Interface

- `run_full_pipeline(spec_path) → PipelineResult`
- `configure(config_path) → bool`
- `get_pipeline_status() → Status`

Summary

The ConTiNGENT implementation centers on transforming formal NVMe protocol specifications into executable, replayable hardware validation tests through a modular pipeline combining TLC and Rust-based execution.

Key technical principles guiding the design include:

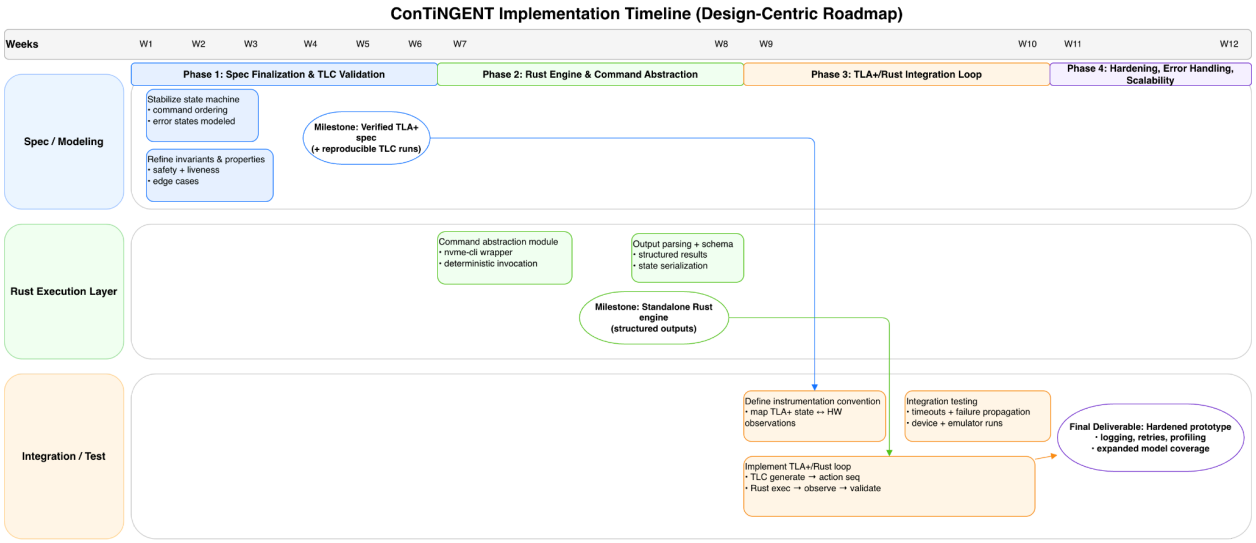
- Specification-driven correctness

- Deterministic reproducibility
- Safe low-level hardware interaction
- Modular architectural separation
- Enterprise-ready scalability

This implementation approach provides a robust foundation for next-generation SSD validation, aligning academic research with real-world engineering deployment.

Implementation Plan

The implementation will proceed in phased increments to ensure that the formal specification remains the primary driver of system behavior while allowing parallel progress across tooling, execution logic, and integration layers.



In **Phase 1 – Specification Finalization and Validation**, the team will stabilize the TLA+ model that defines valid NVMe device state transitions and command sequencing. During this phase, the focus is on ensuring completeness of the state machine, refining invariants, and validating safety and liveness properties using TLC. All edge cases related to NVMe command ordering, error handling, and device state assumptions will be modeled explicitly. The primary deliverable is a verified TLA+ specification with reproducible TLC model-checking results. This phase establishes the behavioral contract that the Rust execution layer will implement.

In **Phase 2 – Rust Execution Engine and Command Abstraction**, development will focus on building a Rust-based execution layer capable of invoking nvme-cli commands and capturing structured results. This includes implementing a command abstraction module, output parsing logic, and a state serialization layer that maps real-world command responses into representations compatible with the TLA+ model. Work in this phase can proceed in parallel

with final TLC property refinement, as long as the Rust interfaces are aligned with the agreed-upon state schema. The main milestone is a standalone Rust component that can execute NVMe commands deterministically and produce structured outputs suitable for model feedback.

In **Phase 3 – TLA+/Rust Integration Loop**, the system will integrate the verified TLA+ specification with the Rust execution engine. TLC will be used to validate allowed transitions and determine the next expected state, while Rust will execute the corresponding NVMe command and feed the observed results back into the model. This bidirectional loop creates a runtime verification mechanism where the specification actively guides execution. Integration testing will validate consistency between modeled transitions and actual device responses. Particular attention will be paid to synchronization semantics, failure propagation, and timeout handling. Incremental delivery will begin here, enabling early validation against real NVMe hardware or emulated environments.

In **Phase 4 – Hardening, Error Handling, and Scalability Enhancements**, the focus shifts to robustness and operational reliability. This includes improved error classification, retry mechanisms, structured logging, and expanded model coverage for additional NVMe features. Performance profiling will assess overhead introduced by model interaction. Where necessary, caching or batching strategies may be introduced to reduce repeated model invocations without violating correctness guarantees. Final system validation will combine stress testing on hardware with model re-verification.

Dependencies and Assumptions

The implementation assumes stable NVMe-cli behavior and deterministic command output formats. Any version changes in NVMe-cli may require updates to parsing logic. Additionally, it is assumed that the TLA+ model remains tractable for TLC model checking within acceptable runtime bounds. Hardware access constraints or variability in device firmware behavior may introduce nondeterminism that must be explicitly modeled.

A key dependency is the schema alignment between TLA+ state representations and Rust data structures. Early agreement on serialization format reduces integration friction. Parallel development is feasible between specification refinement and Rust execution scaffolding, but final integration depends on state model stability.

Technical Risks and Trade-offs

One primary risk is **state explosion within TLC**, particularly if NVMe command combinations produce a large transition space. This may slow validation cycles and constrain model

complexity. Mitigation strategies include abstraction of non-essential state variables, bounded model checking, and decomposition of the model into smaller verified components.

Another risk involves **mismatch between modeled behavior and hardware reality**. Real NVMe devices may exhibit timing-based or firmware-specific behavior not captured in the specification. Iterative refinement of the TLA+ model based on empirical observations will be necessary. Introducing controlled nondeterminism in the model can help accommodate real-world variability.

There is also a **toolchain integration risk**. TLC is not designed as a runtime orchestration engine, so coupling it tightly with Rust introduces architectural complexity. A trade-off exists between strict runtime model validation and performance overhead. To mitigate this, the system can separate offline exhaustive verification (via TLC) from runtime conformance checking with reduced scope.

Finally, acknowledged technical debt may arise in output parsing and command abstraction if NVMe-cli interfaces evolve. Encapsulating command handling behind stable Rust traits and isolating parsing logic will limit downstream refactoring costs.

Overall, the implementation emphasizes a specification-first workflow, with Rust serving as a controlled execution layer guided by formally verified state transitions. The phased approach ensures that correctness guarantees remain central while enabling incremental integration and empirical validation.

Conclusion

This project presents ConTiNGENT, a specification-driven NVMe validation framework designed to improve test coverage, reliability, and efficiency within SanDisk's SSD validation process. By generating executable tests directly from formally verified TLA+ models, the system reduces human error, eliminates redundant testing, and enables deterministic, reproducible validation against real hardware.

The design combines TLC-based formal trace generation, a Rust-based trace translation layer, a dedicated NVMe execution engine, and an extensible results analysis workflow. Clear separation between components ensures maintainability, safe hardware interaction, and scalability for future expansion. Overall, the proposed design provides a strong foundation for reliable NVMe validation and supports successful project outcomes through improved correctness, repeatability, and long-term integration potential.