# Tech Feasibility

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

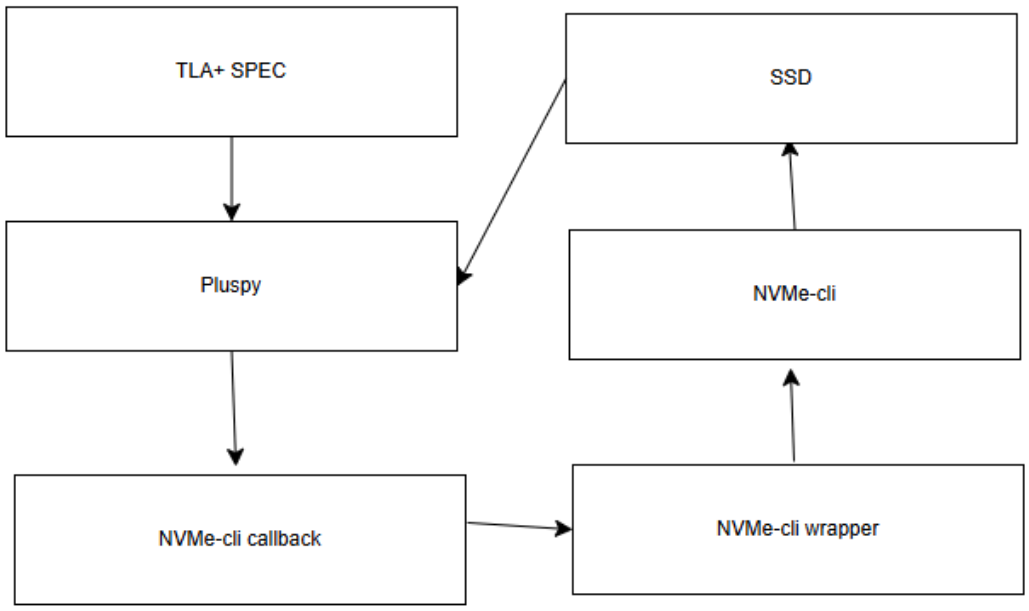
# Table of Contents

# Introduction:

Every single computer needs to have some way to store long-term data. A computer needs long-term data storage to save files and download music or videos; most applications would be too large to download for each use. A computer needs long-term storage for it to function in the same way it does today. This long-term data storage comes in several shapes and sizes; however, a solid-state drive is one of the fastest storage solutions. Solid State Drives, or SSDs, are a massive part of the long-term storage industry. The market size was estimated to be 8.56 billion USD in 2023. SSDs are fast and efficient; however, as long-term storage, they also need to be reliable. To ensure reliability, a validation team must perform adequate testing to prevent the loss of essential data like personal documents or photos.

Our client is Western Digital, a large storage solution company that innovates, produces, and validates these SSDs. Currently, Western Digital and other SSD manufacturers mostly rely on traditionally defined testing, such as unit tests that are primarily human-driven. Western Digital performs extensive automated tests that rely on predictable edge cases to ensure that new and current products are production-ready. A current issue with these test cases is that they are highly finite and run out of testable cases quickly, resulting in idling SSDs and supporting testing hardware. The idling wastes time and computing power because the validation process does not fully utilize the allocated SSDs and testing hardware. In a more optimal solution, the idling time could instead find new bugs and state combinations in testing. Our goal is to create a solution for this downtime. We are creating a brute force approach to testing SSDs repeatedly and randomly, as in this specific case, we are trying to test as much of the SSD as possible to work around traditional testing limitations.

Testing Process Figure 1



As shown in figure 1 the idea for our solution is to use an API-like terminal program called NVMe-CLI which can communicate and call commands to the hardware of an SSD, then connect this to a Python script that will call those commands. The commands that the Python script will be calling will be determined by two things, first is a TLA+ specification file and the second is a randomly generated number or seed. TLA+ is a high level programming language which can write ‘code’ above the code level, a programmer could write a specification file that has multiple tests they want the NVMe-CLI to call. The seed will randomly decide the tests written in the TLA+ specification. So essentially our solution can load TLA+ specification script then call the tests randomly to the NVME-CLI through a Python script. These tests can be done repeatedly and can end up being used during down time between other tests. This will increase the likelihood of the testing team to find errors in their SSDs.

Now that we have seen what the problems are with SSD testing and our high level solution, we will now talk in more specifics about the key technological challenges and the alternatives that we may use for our solution. As the project is in the early stage we must decide and explain what we are going to face during development. We will firstly begin by analyzing the major technological challenges we expect to face which are getting NVME-CLI to properly work with a Python script, TLA+ being able to be properly translated into Python commands through the use of PlusPy, and a logging feature for errors. In the future subsections we will then analyze each of these carefully and look at alternatives.

## Technological Challenges:

In this project, we face several key technological challenges that need to be addressed as we work towards automating and optimizing the testing of NVMe (Non-Volatile Memory Express) SSDs. These challenges relate to how we generate test sequences, manage the complexity of testing large systems, and ensure efficient use of hardware resources.

First, traditional testing methods, like unit and integration tests, don’t fully explore all possible combinations of NVMe operations, which limits their ability to uncover rare bugs. We need to develop a way to generate more comprehensive test sequences and cover a wider range of possible system states. This also brings the challenge of efficiently handling and navigating the large state spaces generated by these tests.

Another issue is that hardware resources, such as CPUs and NVMe drives, can sit idle once a set of test cases is complete. Our system must find a way to keep these resources active, reducing downtime and improving efficiency.

A core part of the project will involve integrating formal specifications of the NVMe system’s expected behavior with real-world testing. This requires ensuring that model simulations can be paused, translated into real NVMe commands, and the results logged and checked for issues.

We also face the challenge of making sure any randomly generated test sequences can be reproduced, so that bugs can be tracked down and fixed. Additionally, ensuring that the development environment is consistent across the team, managing code efficiently, and integrating with the NVMe interface for testing all present important hurdles.

Addressing these challenges will require careful planning and strategic choices in how we design and implement the system, but identifying these hurdles now helps us prepare for what lies ahead.

## Technology Analysis:

In the development process for NVMe (Non-Volatile Memory Express) SSDs (Solid State Drives), validation and testing are needed to ensure the product works as intended and conforms to NVMe standards. Traditionally, humans have programmatically defined test case sequences. For example, an engineer just updated the write command for an NVMe drive and wants to ensure it works properly. Usually, an engineer might define unit and integration testing for integration and correctness. Unit testing checks a single unit or the correctness of a function or object, like checking if issuing a write (0111 0010, 0x0000000000013FFF) will write 0111 0010 to 0x0000000000013FFF. Then, integration tests ensure that the newly updated write() adequately integrates into other functions, such as io_passthrough(). One technical limitation of this traditional approach is that depending on how they are defined, these tests either do not consider temporal testing elements (such as the order of actions or multiple actions at a time) or are limited to human creativity in the order of these tests. To achieve more comprehensive test coverage, we should generativity test numerous different temporal combinations of actions.

A limitation of the traditional approach is that the test boundaries and combinations will not deeply and exhaustively cover unique sequences in large state spaces. After completing these test sequences, which only have limited coverage in the first place, the hardware allocated to run these tests runs idle. Instead of exhaustively searching for unexplored paths for testing, resources such as CPU, RAM, and NVMe drives run idle, wasting company time and money. Moreover, as NVMe development becomes faster and more cutting-edge, the cost in man hours to develop new validation and test cases will become costly. In theory, generative testing and reporting will explore every state in a complex and large state space. Though practically, the time taken to investigate these will be virtually infinite. The hope is to constantly and randomly explore new sample combinations of states, automating part of the validation process while providing deeper coverage, fully utilizing hardware, and lowering development costs.

The ideal solution involves taking a working model-simulation specification file that formally defines how the NVMe system acts and using it to drive real-world NVMe testing. This formal definition will determine possible temporal testing sequences that define modification and Input/Output (I/O) operations to accomplish this. By itself, the model-simulation specification will only be able to simulate a model that speculates how a system acts and how states change concerning each formally defined action. However, one goal of this program is to pause the model simulation, then intercept each state from the model simulation and issue the

related real-life NVMe command. After completion of the NVMe command, the results may or may not be logged depending on what is needed (bugs, error codes, success, changes). Then, the model continues to the next stage, where the process repeats until the program ends.

The tests are innumerable when considering testing possible I/O and modification operation sequences. We can better understand this problem with directed graphs, where each state (one part of the sequence) leads to another state (another part of the sequence). Naturally, the directed graph will be vast and intensive to search. The program will randomly search the directed graph in a depth-first search (DFS) style to find unique bugs in NVMe product development. As a result, this program will allow more automated usage of allocated testing resources and avoid idling.

To have unique testing sequences each run, we will utilize random seeded number generation to create unique testing samples each time we run the program. Having seeded randomness also allows us to see if a bug encountered during searching is repeatable by resampling with the same seed. If the program determines these bugs are repeatable, then the program will log this error and the sequence of actions it took to get there, which will be further explored by test and validation engineers.

Other than the previously mentioned test execution and coverage, there are a few criteria to consider when choosing the technologies we will employ for this project, such as the development process, debugging/logging, modularity, maintainability, community support, and documentation. On the development side, it is *crucial* that what we make is easily accessible, maintainable, and modular. Not only to make our development more accessible and simple, but also for Western Digital should they decide to incorporate this program in their use case. As a result, we primarily consider a high-level, object-oriented programming language that is versatile enough to connect services and different programming interfaces. Pure language performance will not be a concern as the bottleneck will mainly be in the NVMe interface, likely implemented in a lower-level language like C, which does the bulk of the hardware operations for us. Logging and debugging will also depend on our services and products, as the debugging and logging libraries/services can vary. Community support and documentation will be vital when considering libraries to conduct this project successfully.

For our development process, we need to consider our development environment and how to facilitate collaboration and sharing. Our development environment will need to be able to virtualize a separate operating system to run NVMe commands. In addition, this system(s) must have a secondary NVMe drive we pass through to the system we operate to run these operations. To ensure the development environment is uniform across all members, we will use identical versions across all technologies; we may consider access to a remote central server for running tests to encourage the same testing environment. For the actual development, we will use an IDE to streamline productivity and combine functions such as debugging, program linting, and Intelli-sense all into one place. We will use version control to track changes when storing this code, ensuring reversibility and maintainability. To facilitate better collaboration, we will combine version control with an online repository or repository server so the team can upload their repository branches, acting as cloud storage with version control and release management.

Now, with a better understanding of this project's goals, we will consider potential technologies and alternatives that will allow us to carry out this project within a realistic time frame. This project has nine main components, three of which are inflexible. Due to client requirements, we will only have one usable technology/framework.

# Technologies:

## Programming Language:

The first category is our programming language; this will be the glue that holds everything together and the base from which our logic will operate. Due to client requirements, we will only consider Python as the project requires our implementation to be easily maintainable and integrated within Western Digital's team and workflow. Using the same programming language, we significantly reduce the time and complexity it takes for Western Digital's validation team to utilize and understand the code base effectively.

Python

Python is a high-level, object-oriented, interpreted, dynamically typed programming language. Guido van Rossum first released it in 1991; by then, it had a fair amount of maturity, support, and community backing. One of the strengths it provides for us in this project is that Linux distros usually have Python installed by default, is easy to read, assuming good practices and extensive libraries, and it easily integrates into other languages and services.

## Formal Specification Model-Simulation Language:

Next is a formal specification and model simulation language; this component will allow us to formally specify how a system, in this case, an NVMe drive, is supposed to behave. These languages are less for programming languages and more for describing precisely how a design should work using a rigorous and unambiguous model of how a system should work under various conditions. A complete specification will help debug or program/engineer functionality based on the specification. In this project, our client will give us a complete, model-checked TLA+ specification to drive real-world testing programmatically. Similar to the programming language, our chosen formal specification/model simulation could be more flexible as it depends on Western Digital's validation team's language knowledge.

TLA+

TLA+ is a high-level formal specification language that allows one to define and model systems and programs explicitly with simple mathematics. This language emphasizes the temporal aspects of a system, considering concurrency and distributed actions. Its central goal is to eliminate expensive and difficult-to-find design errors in code. Leslie Lamport developed TLA+ in 1999, and it is widely used today in companies like AWS, Oracle, and Microsoft.

## Version Control:

Version control will be vital in this project, as its code base contains many components and requires asynchronous teamwork. We can track changes and resolve conflicts in asynchronous development by incorporating version control into our workflow.

### *Alternatives*

Git

One of the most used and taught forms of version control today is Git. Our team's university, Northern Arizona University, uses Git to teach its students about version control, and most job opportunities for large projects include Git and its related repository servers in their skills. The creator of the Linux kernel, Linux Torvalds, created Git to handle distributed development scalably and reliably. This version control is mature as it was created in 2005 by a reputable programmer.

Subversion

We found an alternative to Git with some web searching. Subversion is an older alternative to Git, created in 2000 by CollabNet to address issues with the version control systems available at the time. Fundamentally, it is a lot like Git and other version control systems, as it allows users to manage and keep track of files and changes over time. Subversion allows for recovering old repository versions and a detailed history of evolution.

Mercurial

When searching forms and the web, we also found Mercurial, a distributed version control system emphasizing multiplatform support, performance, and ease of use. Like Git, it was created in 2005 by Matt Mackall as an open-source alternative to proprietary version control.

### *Criteria*

Version control's essential aspects within the context of this project are its accessibility, development process, maintainability, and ability to facilitate collaboration. A version control system must be easily accessible for all members to contribute well. As for the development process, it needs to be flexible and decoupled enough but remain simple enough to use within a short time frame of one academic semester. Another aspect of this project that will come up during development is the maintainability of the version control system. Finally, the last aspect is how the system encourages collaboration through integrations or built-in functions.

### *Analysis*

For this project, we will consider more popular version control systems (VCS) such as Git, Subversion (SVN), and Mercurial under the scope of the criteria mentioned earlier.

Accessibility:

- Git

In modern development, Git remains one of the most widely used version control systems. Due to its popularity, the extensive documentation and community support make Git more accessible to our team. Git is also used as a tool in many academic contexts, including NAU, meaning that the team developing this project will be most familiar with this program.

- Subversion (SVN)

SVN, like Git, also has substantial documentation, mainly due to its popularity in the past. Due to a few key factors, it is not as commonly used today (older documentation, less community support), and the fact that it lacks the familiarity that our team has with Git, which makes it less accessible to us.

- Mercurial

The learning curve for Mercurial is very simple (less complex than even Git). While the way you use it as a version control system is similar to Git in the way it abstracts its functions, there are a few key differences. Due to its design philosophy, its simple implementation can be a double-edged sword because it is inflexible. Mercurial's inflexibility can be an advantage because it discourages differences in the engineering process between team members. However, if more customization and automation are needed, Mercurial falls short of Git in its feature set. In comparison, Mercurial is primarily a version control system, while Git is not only a VCS but, through supporting integration, a whole platform that allows for the automation of even non-source control tasks. For this project, this is a considerable alternative to Git.

Development process:

- Git

Git's architecture is distributed and flexible. Every developer can have a copy on their system. When merging/combining code, particular functions for merging and resolving conflicts may appear in the difference. This approach and nature favors remote/asynchronous work, which is likely how the development of this project will go. An additional aspect of Git is how its branching works. Git branching allows for much customization and flexibility, supporting complicated, decoupled workflows that separate development into smaller, more manageable parts. One downside of this flexibility is that it can become complex and challenging to understand as it scales.

- Subversion (SVN)

Subversion is an inflexible centralized architecture. This approach will need a centralized server to collaborate between members. While the centralized server approach is not inaccessible to us, any downtime in this project when development is needed is a serious concern. The centralized approach, however, allows for a single source of code, reducing the complexity of the workflow and collaboration to a much simpler one. In addition, branching and merging are less sophisticated because there is less need for complex solutions in a centralized approach. Conversely, the need to support a more complex workflow

with many moving parts makes this version control less effective for this project due to the asynchronous nature across different networks and platforms.

- Mercurial

Mercurial is similar to Git but simpler in its feature set and implementation. Compared to Git, the branching model is simpler and more streamlined, which would initially be more effective productivity-wise when our features remain small and simple. However, if the project becomes more extensive and feature-based, and components become larger, this version control system's simple and less flexible nature may become a limiting factor.

Maintainability:

- Git

Due to Git's distributed nature, data loss and resilience are strengths of this system. Git also avoids integrity violations with hashing (SHA) in file storage. The community support and widespread, large ecosystem make it easier to maintain long-term. A side note of Git's method of storing versions is that it stores changes and commits them as "snapshots" in time, which can become quite computationally expensive as a project becomes more significant (particularly those with more complex branching models). Another issue that may arise is that as project branching becomes complicated, keeping Git branches and workflows clean is challenging.

- Subversion (SVN)

As previously mentioned, SVN has typical centralized approach issues such as uptime, data loss, and accessibility. Since SVN stores all code in one place, if a developer tries to develop during downtime, they cannot contribute productively. Another issue is that since there is only one place where the data is stored, data loss is a significant concern unless there is some distributed storage/storage redundancy. The positive aspects of the centralized approach are that it allows for much simpler control of data and version history, simplifies maintenance, and assumes there are no issues with the underlying hardware or how it's used fundamentally. Assuming strong redundancy and the server is easily accessible, this technically would be the easiest option to maintain due to its simple, centralized nature.

- Mercurial

Mercurial is reliable in storing data due to its decentralized approach, which is similar to Git. It can also be simpler and more scalable than Git when considering performance due to how it stores changes. Unlike Git, Mercurial only stores the differences in each change, allowing for more efficient repositories on large projects at the cost of flexibility. The major downside of Mercurial in the way of maintainability is that it lacks maintenance tools and integrations that Git has, such as integration with CI/CD platforms like GitHub Actions, Jenkins, or GitLab CI, which can significantly slow down large/complex automation and integration tasks.

Collaboration:

- Git

Github's integration with other platforms makes it a vital tool for collaboration in many projects. Through its platform/repository servers like GitHub and GitLab, some of its essential components are pull requests, issues, and discussion boards geared towards a multi-developer collaborative effort. In addition, through Git's platforms, it has deep integration with 3rd party and community plugins/libraries/automation.

- Subversion (SVN)

Compared to the other options, SVN has fewer collaborative features and is also limited in the possibilities it has to integrate with modern platforms that support complex workflow collaboration. In addition, its centralized approach makes it hard to carry out specific tasks like code reviews, developing in separate environments, and review merging

- Mercurial

Mercurial is another solid alternative when considering collaboration. Its collaboration features are less extensive and simpler than Git, and it integrates well with platforms like BitBucket. This would provide an advantage during the initial collaboration process, as the set-up will be less complicated and complex. However, if collaboration needs to become more complex and scaled, it may become limiting. One feature it lacks is advanced pull requests or merge requests that allow for proposed changes, code review tools, PR/merge automation, and approval checks.

### *Chosen Approach*

To complete the analysis for version control systems, we create a table based on the previously mentioned alternatives and how well they apply to the criteria we are searching for.

In summary, Git is highly flexible, decentralized, and has an in-depth feature/integration set that provides the most flexibility at the cost of complexity and is computationally expensive at scale. Mercurial is like Git but has a more straightforward feature set that makes it easier to learn thoroughly, but it is slightly less flexible due to that very fact. Subversion is the technology with the most contrast to Git or Mercurial as it is a simple, centralized version control with a minimal feature set, making it easy to maintain but very inflexible and less accessible compared to Git or Mercurial. The qualitative nature of this analysis favors making a table that places each alternative relative to each other, such as ranking 1st, 2nd, or 3rd based on the project's needs and preferences.

| | Git | Subversion | Mercurial |
|---|---|---|---|
| Accessibility | 1st | 3rd | 2nd |
| Dev Process | 1st | 3rd | 2nd |
| Maintainability | 2nd | 1st | 3rd |
| Collaboration | 1st | 3rd | 2nd |

Table 1: Ranking of version control

Subversion is not a good fit for the team or project needs. Subversion's biggest strength lies in its simplicity and maintainability when only considering the version control system. This robust simplicity and maintainability come at a considerable cost due to the limitations and constraints to our approach to the project and how we collaborate, as we've seen in the analysis.

Mercurial is placed close to Git but barely falls short of Git, primarily due to Git's overwhelming community support and ecosystem developed around it that takes advantage of its flexibility to conform to a vast amount of needs. One area that Mercurial can potentially be stronger in is its maintainability, but when considering community support and tools to compensate for Git's complexity in maintenance, it is the worst option. If the project was less feature-based, had a less complex development workflow, and if Mercurial is potentially more community-supported, it would either be tied or beat Git across the board. With these considerations in mind, Mercurial will be an excellent alternative to consider or switch to if Git becomes a blocker in the project's development process.

Git's decentralized and broad/flexible integration with other components makes it a prime choice for accessibility and development in the context of this project. Its integration with other mature and modern platforms like GitHub and GitLab, which support advanced collaboration tools, is also one of the most substantial options for collaboration. Its complexity can potentially burden projects and make its maintainability not an ideal choice compared to the simpler options, but using 3rd party tools with extensive community support makes maintaining Git repos more viable. With these points in mind, it is clear that we should use Git as our primary version control system for this project.

## Repository Server:

A repository server is a subcomponent of version control that we must consider for this project. It further extends the function and usability of version control. A server will allow us to use our version control, automate some of our build and deploy processes, securely store our code should we lose the code base locally, and facilitate collaboration through having a central store for code and information.

### *Alternatives*

GitHub

Much like Git, our team is already familiar with GitHub and its tools because our university chose this to store our repositories when teaching us how to use version control. It is a platform launched in 2008 designed to provide a centralized repository for developers to manage and collaborate in code. Today, Microsoft owns GitHub, a free-to-use tool with additional licensing for more features.

GitLab

GitLab is an open-source platform similar to GitHub, built around Git. It is an excellent alternative to GitHub that focuses on providing features for project collaboration and workflow automation. Dmitriy

Zaporozhets created GitLab in 2011 to help his team collaborate effectively, and it has since grown to become one of the most widely used platforms for DevSecOps.

BitBucket

BitBucket is an additional Git-based repository service that allows for the storage and management of Git. It was created in 2008 by Atlassian to provide its own user-friendly, collaborative version control system built for developers.

SourceForge

SourceForge is a source code repository that stores free and open-source software (FOSS) in a central location. It allows developers to manage and distribute their creations without being charged for using a repository hosting service. SourceForge was founded in 1999 by Larry Augustin and James Vera.

Beanstalk

Beanstalk is a repository hosting service created in 1999. It focuses on writing, reviewing, and hosting code. Like other repository hosting services, it streamlines workflows and encourages collaboration.

### *Criteria*

The Repository Server has to be a centralized place to hold data, have easy accessibility, have good collaboration features, cost, maintainability, and familiarity. A repository server must be able to hold data in different stages of development, this will make it easier to prevent data loss, and develop with a proper level of professionalism. A repository server must also be able to be accessible by all members of the team and have the proper collaboration features to make development more efficient. A repository must have proper cost and reliable maintainability, so our product can be maintained cheaply and can remain usable for a long time. Lastly a repository server must be familiar to each member of the group as time is limited for our project and familiarity will decrease how long each member will take to learn a new technology. Each Criteria will be rated on a at the end by their respective placement compared to each other on the category. If all of the technologies have a feature that is being tested for or are the same in execution they will all be rated with the 1st.

### *Analysis*

Versioning: This section will be rated on whether or not the options have the ability to use branches and forks, or a replacement for these features. Essentially a feature that allows users to have multiple versions at different stages of development. If the feature does have both these features it will get an automatic 1st where if it does not it will be rated more poorly.

- GitHub

  Github was excellent in its ability to have different versions of a developing code. Github has multiple ways to do this, two of the main options being branches and forks. Branches being the option to have multiple versions or “branches” of your software and forks being a copy of a repository. These both cleanly make GitHub a favorite for keeping multiple versions of a software.
- GitLab

Like GitHub, GitLab also has branches and forks, which are important to the well being of a project, so there can be multiple versions of a product in different stages of development.

- BitBucket
  Like the Technologies before it, Bitbucket also has Branches and Forks.
- SourceForge
  Again SourceForge has branches and forks which make it another technology that has the important versioning feature.
- Beanstalk
  Beanstalk does not support Forks, however it still supports Branching which is a powerful development tool. This makes it slightly less appealing of a technology to use for this section.

Collaboration: This section will be rated on whether or not the options have issues, pull requests or other collaborative features, those could be chats and comments.

- GitHub
  GitHub hosts a wide range of collaborative tools. It has issues, pull requests, and comments on those issues and pull requests, these chats and pull requests also allow for custom tags. However GitHub does lack some collaborative features like a chat for repositories.
- GitLab
  GitLab has both issues and pull requests and a feature that it can hold over some of the other options is that in the issues you can upload diagrams and designs. The GitLab collaboration features also have a simple design with a good amount of customization.
- BitBucket
  BitBucket also has pull requests and has issues, however the issues are more like task management which may be good however is a little too extreme for the scope of our project and the level of collaboration that we need on our Repository Server.
- SourceForge
  SourceForge does not have “pull request” however they have a ticketing system to keep track and have a basic form of issues. This option lacks behind compared to the other options in this feature as it keeps a more basic approach to communication.
- Beanstalk
  Beanstalk does support issue tracking in a similar manner to SourceForge, it has a ticketing system. It also has a chat feature which can be useful for communication on issues, however in our case we are completing a large portion of our chat like communication on other platforms like discord.

Cost: Cost was evaluated based on the cost of the product for our use case, which was to have a Repository Server where our code would stay to be kept up to date and safe from issues with individual computers. So if an option is free to use it will be rated most likely as a 1st, as in the scope of this project we do not need a Repository Server to be a cost.

- GitHub

Although GitHub does have a paid option it is free to use making it a highly rated option.

- GitLab
  Again like GitHub this option has a paid option however the functionality that is needed for our project is free to use.
- BitBucket
  BitBucket also has a paid option but the main functionality is free to use.
- SourceForge
  SourceForge is free to use.
- Beanstalk
  Beanstalk is not free to use as it has a monthly subscription making this option less attractive then the other options.

Maintainability: This section was evaluated on the ease of use for a repository to be maintained.

- GitHub
  GitHub holds excellent documentation and a strong community support network. The user interface is intuitive, and integration with CI/CD tools enhances maintainability.
- GitLab
  Comprehensive built-in tools and documentation. Strong focus on DevOps practices makes it highly maintainable, especially for larger teams. Has slightly better integration with CI/CD tools then GitHub which makes it slightly more appealing.
- BitBucket
  User-friendly interface and solid integration with Atlassian products. Good collaboration features support maintainability somewhat effectively.
- SourceForge
  Historical significance but a less intuitive interface. While it has essential tools, the learning curve can hinder maintainability.
- Beanstalk
  Simple and clean interface with built-in deployment tools. Good for teams needing straightforward maintenance capabilities.

Familiarity: This section will be based on the familiarity that each member has with each different option.

- GitHub
  GitHub is by far the most used and popular option. Each member says that they have a high experience with this option.
- GitLab
  The second highest option is GitLab, as half of the members have an experience with GitLab, however two of the group members have no experience with GitLab making this option poorer then GitHub.
- BitBucket

Only one group member has experience with BitBucket the remaining members have little to no experience with BitBucket.

- SourceForge
  One group member has a small amount of experience with SourceForge the remaining have no experience with SourceForge.
- Beanstalk
  Each group member has little to no experience with Beanstalk.

*Chosen Approach*

To complete the analysis for Repository Server we created a table that demonstrates the rankings of each of the options for our Repository Server. For each option they will get ranked on the criteria of versioning, collaboration, cost maintainability, and familiarity.

| | GitHub | GitLab | BitBucket | SourceForge | Beanstalk |
|---|---|---|---|---|---|
| Versioning | 1st | 1st | 1st | 1st | 2nd |
| Collaboration | 2nd | 1st | 3rd | 5th | 4th |
| Cost | 1st | 1st | 1st | 1st | 2nd |
| Maintainability | 2st | 1nd | 5th | 4th | 3rd |
| Familiarity | 1st | 2nd | 3nd | 4nd | 2nd |

Table 2: Ranking of Repository Server

The best options for our project were between GitHub and GitLab, as these two ranked highest in our evaluation, consistently placing first and second across categories. According to our data in Table 2, GitHub had an average placement of 1.4, while GitLab led slightly with an average of 1.2. Ultimately, the team decided to use GitHub as our repository server technology. This choice was influenced by our familiarity with GitHub, which ranked first in the category that we prioritized most. Additionally, in each category where GitLab ranked first, GitHub followed closely in second, demonstrating its overall suitability for our needs. With GitHub's strong overall performance, it stands out as a robust choice for our project.

## NVMe interface:

An integral component of our project is how to "talk" to the NVMe drive. This interface is partially inflexible as the client wants to standardize it to an open-source repo called NVMe-CLI. Since they use Linux to test their drive functionality, most Linux distributions have NVMe-CLI pre-installed. This component will make calls to the NVMe drive programmatically and return whatever results or bugs may occur on the hardware

level. While there are many development tools for NVMe development and testing, only some fall within our constraints while staying high level for us to complete this project realistically.

NVMe Command Line Interface

NVMe Command Line Interface, or NVMe-CLI, is an open-source tool for managing NVMe devices on Linux. Our client recommends it, so we will likely use it for our codebase. Jon Michael Hands and Chia Network created this open-source tool using the specifications of a collaborative effort from over 100 companies.

## Environment:

Finally, we have the environment where we will develop and test our project. In this section, we must consider a few aspects of our project, such as what environment we will use to develop, what Linux distribution to use, and how to virtualize our implementation to separate the NVMe drive from the host system should we have a kernel-level crash.

We will use virtualization and/or containerization to ensure uniformity between environments and contain code in a separate environment. There are a few things to consider when considering these: containerizing for development and virtualizing for actual testing.

## Containerization:

*Alternatives*

Docker

Docker was created in 2013 by Solomon Hykes to simplify deployment and development. It is a widely used tool in both the academic and professional world. A good use case this provides for our project is that it creates a separate barebones environment that will only contain the necessary libraries for our code from the kernel up to application libraries.

Anaconda

Anaconda was a tool created in 2012 for Python to manage packages and distribution. It emphasizes data science and scientific computing. For this project, it would be a good alternative for uniformizing packages and environments.

Python venv

In Python, there is a module named venv, or virtual environment. Venv creates a contained directory with a separate Python interpreter and installs libraries to separate itself from the system defaults and files. Venv will ensure that no conflicting packages or libraries occur during runtime.

### *Criteria*

Containerization is a fall back for us in case virtualization doesn't work out, so this section is a complete alternative to virtualization. We've found a few technologies that would work in case the virtualization doesn't work out or needs to be supplemented with other technologies. We've evaluated them based on Ease of Use, the programming languages it supports, if it supports hardware passthrough, and how configurable it is.

### *Analysis*

Evaluation criteria for containerization technologies:

- Ease of Use: Based on how easy it is to start the container, how easy the container management is.
- Language Support: Based on how many languages the container format supports, how supported it is by the provider.
- Hardware Passthrough Support: Based on if hardware passthrough is supported by the containerization technology.
- Configurable: Based on how much configuration is available to the programmer/user, and if the configuration is well documented.

Using the above criteria, the technologies that fulfill our requirements have quite the pros and cons when comparing them, for example, Docker has quite the language support, as it really is just a very lightweight VM that instead runs at a higher kernel level, and does less instruction translation compared to a VM. Anaconda and Python VENV are very much language specific, as both are really just Python containers. Anaconda and Python VENV also have to interact with the hardware at the same level, while Docker will allow for hardware passthrough if done correctly. All three are easy to use in different ways, and configurable to similar extents. Docker does have the most configuration available at both the container level and the interior container level, while Anaconda and VENV are configurable usually through regular text based files. Testing was completed in a similar fashion to virtualization, as the two are very similar concepts, just focus on different levels of kernel hacking to make what they do achievable.

### *Chosen Approach*

Below are the ranks for each technology, and the reason why we will go with Docker if the need arises. Anaconda is the worst option, as the language support is Python only, and we require the use of TLA+ for our modeling language.

| | Docker | Anaconda | Python VENV |
|---|---|---|---|
| Ease of Use | 1st | 3rd | 2nd |
| Language Support | 1st | 3rd (Ties) | 3rd (Ties) |
| Supports Hardware Passthrough | 1st | 3rd (Ties) | 3rd (Ties) |

| | | | |
|---|---|---|---|
| Configurable | 1st | 3rd | 2nd |

Table 3: Ranking of Containerization

## Virtualization:

*Alternatives*

FEMU

FEMU is an accurate NVMe simulator based on QEMU/KVM that virtualizes a separate OS with an attached subsystem that behaves and runs like an NVMe drive but does not need or use one. This system supports running full-stack applications on top, allowing us to test the result of new functions without using or risking real hardware.

KVM + QEMU

KVM stands for Kernel-based Virtual Machine. It is the Linux implementation of system virtualization that turns a Linux into a hypervisor that allows for running separate, isolated virtual environments. KVM allows us to pass through hardware onto whatever virtual machine we create. That way, it is as if the hardware is directly attached to the virtual machine rather than through a translation layer.

Virt-Manager

Virt-Manager is a desktop interface that helps manage KVM and/or QEMU without using a terminal through the Linux library liberty. It will help speed up learning and utilizing Linux virtualization rather than manually doing what Virt-Manager already automates. Our current use of Virt-Manager also makes debugging problems more manageable.

The operating system for this project will be restricted to a Ubuntu Linux distribution due to the freedom and community support it provides to access hardware through the kernel compared to other operating systems like Windows and MacOS. In addition, most of the NVMe interfaces and virtualization that we will be realistically only support Linux. We mainly use Ubuntu because the client explicitly states that we only use Ubuntu 24.04, similar to the OS they use to test their products.

*Criteria*

Choosing a technology for virtualization needs to have a few criteria, such as ease of use, source availability, ease of configuration, hardware passthrough support, performance, and familiarity. The biggest criteria is the hardware passthrough support, as being able to pass through the NVMe drive to test it in an isolated environment, while not a requirement, will prevent our development server from fully going down if the drive crashes.

*Analysis*

Evaluation criteria for virtualization technologies:

- Ease of Use: Based on how easy it is to start the container, how easy the container management is.
- Source Available & Documentation: Based on if the source code is available, and how easy it is to get started with the virtualization technology.
- Ease of Configuration: Based on how easy configuration is to do, and if the configuration is well documented.
- Hardware Passthrough Support: Based on if hardware passthrough is supported by the virtualization technology.
- Performance: Based on how reactive the virtualization is, through both guest OS and host OS.
- Familiarity: Based on how easy the interface is to navigate in the case of GUI, or how easy it is to use commands to navigate in the case of terminal interface.

Using the above criteria, we've identified a few technologies that fit the requirements, such as FEMU, KVM + QEMU, and Virt Manager (which is just a GUI overlay for KVM + QEMU with lots of management). A few tests that we ran to determine what would be the best virtualization technology for our use case were things like how easy it was to set up a Virtual Machine in the program. We tried to also test the performance of the virtual machine with how bad the input lag is, and how familiar we are with the technology itself. FEMU was probably the worst competitor of the group, as it really wasn't meant for our use case as we soon found out. It spun up a virtual machine, but we could not get the virtualized hardware to work for our project. The source was available for the 'emulator' but horribly documented on how to 'spin up a drive' and get it working. KVM + QEMU and Virt Manager both are well documented and are 'source available' for their projects. All three are kind of difficult to configure, although the most straightforward one was the Virt Manager, as it is the only one managed by a GUI. Both KVM + QEMU and Virt Manager support hardware passthrough, which requires a custom patched kernel, and some tweaking of BIOS settings to make it work. FEMU does not allow for hardware passthrough. Performance is something of a relative requirement, as we really only need it to be performant enough to run the specific tests, which mostly involve functionalities. Lastly, familiarity is really one that matters here, and was primarily tested by asking the unfamiliar teammates to try and make something happen within the virtualization, for example, 'spin up a VM' and having the others try to create a VM through each technology. FEMU was definitely the worst one for this, as the 'spin up' required a few commands from the command line every time you wanted to spin it up. KVM + QEMU was okay at it, since it was command line based, but the configuration saved, you only had to do the setup commands once, then run the 'start' command each time you needed to turn on the VM. Virt Manager had the best interface in terms of familiarity, especially since it is GUI based, and everything has pictures or logos to guide the user through.

### *Chosen Approach*

Below are the ranks for each technology, and the reason why we went with Virt Manager to manage the Virtual Machine. FEMU is the worst option of the group, since in testing, it did not support hardware passthrough which was the biggest need in a virtualization technology.

| | FEMU | KVM + QEMU | Virt Manager |
|---|---|---|---|
| Ease of Use | 3rd | 2nd | 1st |
| Source Available & Documented | 3rd | 1st (Ties) | 1st (Ties) |
| Ease of Configuration | 3rd | 2nd | 1st |
| Supports Hardware Passthrough | 3rd | 1st (Ties) | 1st (Ties) |
| Performance | 3rd | 1st | 2nd |
| Familiarity | 3rd | 2nd | 1st |

Table 4: Rankings of Virtualization Tech

## IDE:

A final sub-component of our environment is the integrated development environment we will be using. Our project will have many testing, development, collaboration, and debugging mechanisms and will benefit from a more streamlined workflow. A positive aspect that most IDEs provide is that they provide tools for all of these considerations and can also help automate some of our workflows through extensions and built-ins.

*Alternatives*

VSCode

VSCode was developed by Microsoft and released in 2015 as a lightweight and versatile code editor. Initially, Microsoft developed VSCode for Javascript and Typescript, but further development of VSCode has optimized it for a wide range of languages, like Python. As it has grown into one of the most significant code-editing environments, community support has grown, and plenty of extensions exist.

Pycharm

JetBrains released Pycharm in 2010. It is an integrated development environment dedicated to Python. This program has many useful features, such as static code analysis, debugging, and Git integration. Due to community support, there is also an extensive library of extensions, much like VSCode.

Jupyter Notebook

Released in 2018, Jupyter was a more interactive alternative to development geared toward data science that makes creating and sharing documents and programs more accessible. It was based on another project called IPython, which only supported Python. Jupyter Notebook, however, is more agnostic because it supports multiple programming languages (particularly those used for data science).

*Criteria*

For our IDE of choice, we have some very simple criteria. Our IDE should be able to program Python effectively, as it is the language required by our client, be easy to use, be at least somewhat familiar with all group members, allow for easy collaboration, support a wide range of plugins, and be performance focused to prevent waiting. Each of our possible IDEs will be ranked similarly to the other technologies, and the one with the highest ranking will be our IDE of choice.

*Analysis*

Easy of use:

- VS Code

  VS Code has a simplistic yet full-featured design that allows for easy navigation of the project folders and files. It takes file management principles from most modern operating systems, making it initiative and easy to use. It also works with many other technologies we plan on using, such as GIT. Because of this, VS Code has a huge presence in industry applications and, as such, many tutorials t

- Pycharm

  Pycharm offers many features on top of what VS Code offers. However, this makes the learning curve much steeper. However, Pycharm is focused on languages such as Python. This would make setup much easier, as most of the libraries and packages would be included already. Avoiding the need to find, download, and install them.

- Jupyter Notebook

  Jupyter Notebook has a very simple and straightforward interface, making it easy to learn. However, this means it forgoes a lot of the features of other IDEs. It does feature a cell-based format, allowing for testing of individual functions without the need for other functions or cells to be working. This allows for much easier debugging and compartmentalized development. However, this also makes

Dev process:

- VS Code

  VS Code offers various debugging features. Including the ability to inspect and show information about variables, functions, and classes, as well as breakpoints and a built-in console. It also offers integrated support for Jupyter notebooks, allowing for inline code execution inside of Python files.

- Pycharm

Pycharm beats out VS Code in terms of debugging, as it provides quite a few built-in profiling tools, such as inline debuggers, a memory profiler, and a performance analysis feature. It also includes several refactoring tools that make reformatting, simplifying, moving, and renaming code easier. On top of all this, it also includes an integrated test runner, which allows for string-end-to-end Pyth

- Jupyter Notebook

  While ideal for data science and analysis, Jupyter Notebooks is not as feature-full or efficient as the alternatives for our use case. While it does offer inline code execution and execution of specific code blocks, it has weak debugging capabilities and requires plugins or tools for more advanced debugging.

Familiarity:

- VS Code

  VS Code is what we have all used in previous classes and projects. It has been used frequently by all of our team members for many projects in several different languages, including Python and C++. We have the most experience with this IDE and its features.

- Pycharm

  Pycharm has been used extensively for Python programming by one of our members. However, the rest of the team has very little to no experience using it. This makes it a difficult pick as most of the team members would have to spend a good deal of time learning a new IDE.

- Jupyter Notebook

  Jupyter Notebook is the IDE of choice for Python-based classes. While it offers features specific to Python, these classes are not that popular and therefore not frequently used by our team. Jupyter Notebook also doesn't offer the same customization as other IDEs, which further slows down development per team member.

Collaboration:

- VS Code

  VS Code provides robust support for collaboration. It has built-in SSH support, which allows us to securely connect to our development system to all develop on the same system. On top of this, it also supports GIT, which will be our version control system of choice. This makes updates and saving points much easier. It also allows for real-time, collaborative coding with a simple extension.

- Pycharm

Pycharm allows for integrated version control with Git with support for pull requests and commit history. Pycharm also supports a similar real-time collaboration function to VS Code; however, it is not as seamless to set up or use.

- Jupyter Notebook

  Collaboration is where Jupyter Notebook shines brightest. The notebook files are easily shared on Google Colab, which natively enables parallel programming between multiple developers. This makes Jupyter Notebook great for sharing code between developers and live updates. However, the Jupyter notebook does not have native integration for GIT and wouldn't allow us to remotely access our testing env

Plugin support:

- VS Code

  VS Code has native support for many plugins that provide additional features, such as GIT, SSH, Jupyter Notebook support, and more. They are easily accessible, download and install in seconds, and automatically update when you relaunch VS Code. This is by far the most robust implementation of plugins out of any of the other IDEs and allows us the functionality of Jupyter Notebooks and the addition

- Pycharm

  Pycharm supports a large library of plugins with support for tools like GIT. While these plugins are focused on Python development, there are fewer of them than what is provided by VS Code. These plugins are also more focused on scientific libraries, which we will not be using, making the list of relevant plugins smaller.

- Jupyter Notebook

  Jupyter Notebook offers limited plugins compared to our alternatives. On top of this, the plugins are not integrated well into Jupyter Notebook and require more work to get each development environment set up and working properly.

Performance:

- VS Code

  VS Code is the lightweight version of its brother Visual Studio. VS Code doesn't require gigabytes of downloaded information to program in any language, and the running performance is just as good. The

application opens in seconds and has the low memory equivalent to PyCharm.  Adding too many plugins can affect this; however, we do not plan on implementing too many plugins, so this shouldn't be an

- Pycharm

  Pycharm can be resource-intensive and can even slow down lower-powered computers, especially with larger projects. It also takes longer to start up compared to Jupyter Notebook and VS Code, making it less efficient.

- Jupyter Notebook

  The Jupyter Notebook is lightweight and starts quickly. If used with Google Collab, all code is executed on a separate server, allowing for development on any system that can run a web browser. However, this does mean the amount of resources allocated is at the will of Google. While the amount of resources can be increased, this does come at a monetary cost, which none of the other IDEs require.

*Chosen Approach*

As shown in table 5 we have chosen to use VS Code as it places first in most of our requirements. VS Code has many features that bring it above and beyond for our use case. It allows us to remotely connect to our testing server via SSH directly inside the IDE. This allows us to directly interface with the code on the system, making testing and parallel programming much easier as there is no need to upload the changes to the server before running the tests. It also supports Git, which lets us keep our development code separate from our production code and allows for easy reversion to previous versions. It is the IDE that the majority of the team is familiar with and allows for the most functionality among all of our alternatives. It is the lightest-weight IDE that allows for code execution on devices, making it the most efficient in code execution and general use.

| | VS Code | Pycharm | Jupyter Notebook |
|---|---|---|---|
| Ease of Use | 2nd | 3rd | 1st |
| Dev Process | 1st | 2nd | 3rd |
| Familiarity | 1st | 2nd | 3rd |
| Collaboration | 1st | 3rd | 2nd |
| Plugin Support | 1st | 2nd | 3rd |
| Performance | 1st | 3rd | 2nd |

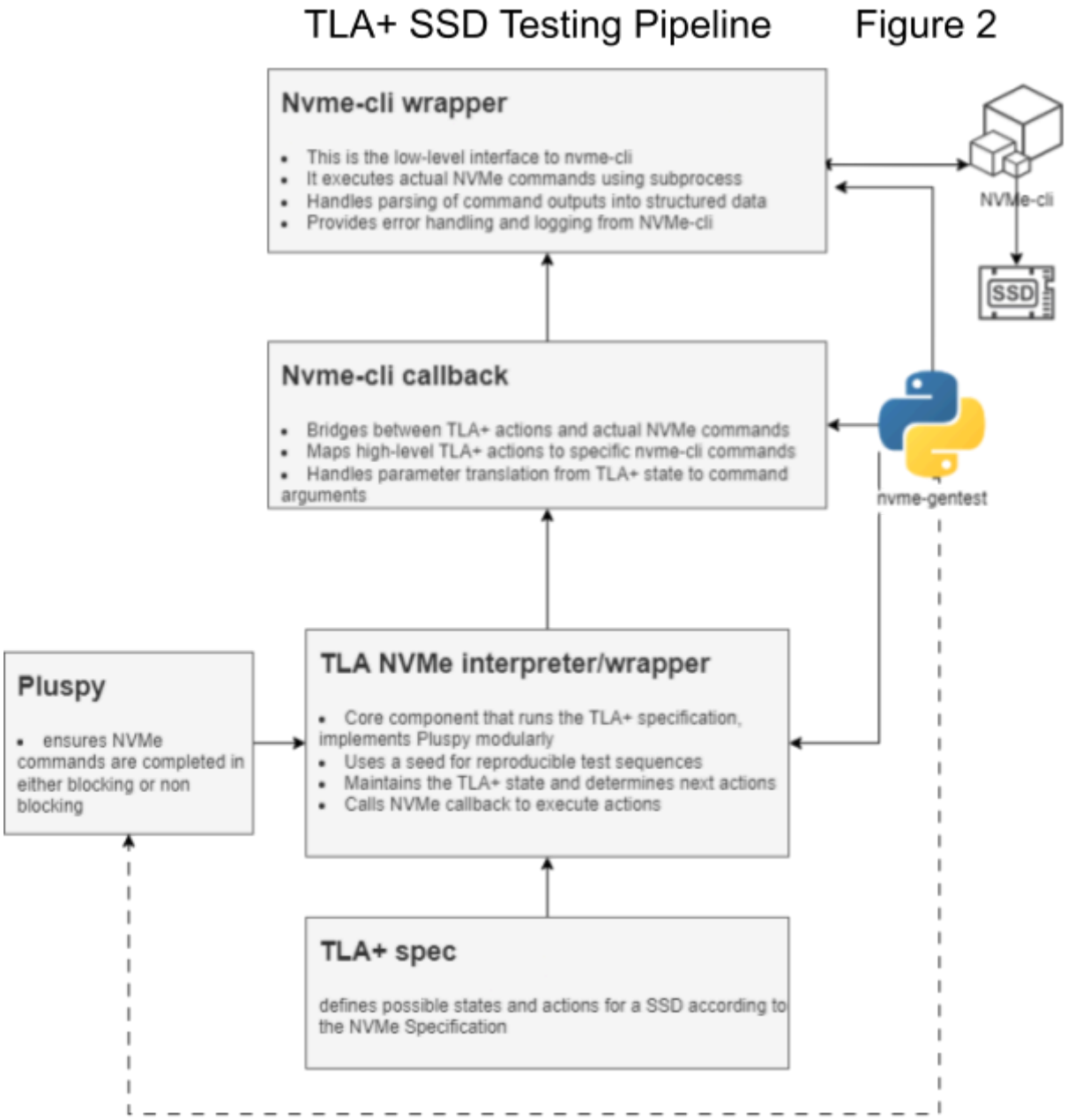
Table 5: Rankings of Integrated Development Environment

## Proving Feasibility:

To further provide these technologies will work for our use case we will create small demos that will validate our research and assumptions. To provide some concrete examples of a few of these tests, we will create a test to ensure that Python is capable of sending properly formatted NVMe cli commands. As well as using Python to parse a TLA+ file to capture and format at least two states of a drive and a transition between them. We will create these small test cases for all vital interactions in our system, however some technologies will be tested indirectly and won't need any dedicated tests. For example we will not be doing a specific test case for our IDE as it will be tested with each of our other programming tests. If our IDE of choice doesn't hinder us in those tests then it will be considered tested and will work for our project.

## Technology Integration:

As previously stated It is crucial that all these pieces come together in a coherent, uniform solution that will allow our clients to easily test solid state devices to ensure reliability and functionality. A solution that is incomprehensible or obscure will only lead to unnecessary work for our client down the road as they try to make sense of a half baked product. This is unacceptable, as we plan to provide a product that can be easily understood and integrated into the current and future workflow of testing SSDs at Western Digital. As such we have put a lot of thought into how the system will be organized and work. To help visualize this we have created a system diagram detailing how each component of the system will work with each other, starting with the TLA+ Specification at the bottom of the graph.



To support modularity and future updates we have split the Python part of our program into several different files. These files are the TLA NVMe Interpreter/wrapper, NVMe-CLI callback, and the NVMe-CLI Wrapper. Starting at the bottom with the TLA+ Specification, we will be using the TLA NVMe Interpreter/wrapper to interpret the TLA+ file, giving us all possible testing options that align with the NVMe specification. Then, with the inclusion of a modified version of PlusPy, we create test cases for a NVMe compliant drive. PlusPy ensures that blocking commands are respected and that the proper output of these commands is returned before the next command is called. This ensures that the command was successful before the next command is called. The program generates each test case with a specific seed that will be included in the logging information. This creates a specific test that can be run over and over again to ensure the fault is present and not just a fluke of another part of the system. This also provides a way for the human tester to run specific tests more often than others, such as read/writes which will be done more often in the real world than reading the power on hours. We then move on to the NVMe-CLI callback which creates a bridge between the TLA+ part of the program and the NVMe-CLI commands. This file specifically translates the TLA+ states and parameters into properly formatted NVMe-CLI commands. After this the last Python file attempts to run these NVMe commands on the CLI executing them on actual hardware. This file then grabs any output from the command including errors and other logging information. This information is then saved as a text file to be reviewed by the tester if any errors arise. If any program ending errors are found the wrapper will alert the human tester that a specific test has failed.

## Conclusion:

Testing SSDs helps both the business and consumer, with everyone wanting faster and more stable SSDs, there needs to be improvements to the ways SSDs are tested. Our team is coming up with a new way to test SSDs, using open source tools that already exist, and putting them together in a cohesive product. Overall, we plan to take the TLA+ model that we are given, modify the current PlusPy implementation to allow for communication to NVMe-CLI, and let it log the output from NVMe-CLI to make sure that it adheres to NVMe standards, as well as validate the output by a user that knows what exactly what is happening throughout the test. We are using tools that we have been shown in order to help us separate the NVMe drive to prevent crashing using QEMU/KVM. Git is being used as version control as well as programming issue management. Using Ubuntu Server 24.04 LTS, we have a stable operating system that is being used as a test bench for the NVMe SSD. Python is our language of choice, as the main program we have to modify is Python based, and we are using it as an interpreter for our TLA+ model. One of our biggest challenges is making sure that all of these technologies are cohesive enough, although we do have alternatives to pivot to if the need arises. Our biggest next step is getting the tools created that we need, the basic connectors that connect the TLA+ file to the PlusPy interpreter, and then have the interpreter call the NVMe-CLI and wait for it to return its output, and log.