WPI

Analysis of Deadline Estimations for Time-Triggered Scheduling in Autonomous Vehicles

A Major Qualifying Project
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science

March 20, 2020

Submitted By:
Amanda Chan
Chau Do
Xinzhe Jiang
Dung Nguyen

Submitted To:
Professor Mark Claypool
Worcester Polytechnic Institute

NVIDIA Corporation

Abstract

NVIDIA is a technology company in the computer chip design and manufacturing industry known for its Tegra system-on-a-chip (SoC) units. NVIDIA's Performance Architecture Team (PAT) needs to ensure that autonomous vehicle safety functions execute before their deadlines. Accordingly, we developed nvPlayfair, an automated platform for engineers to measure system software performance on the Tegra SoC. This platform consists of a control application, a benchmarking framework, and a dashboard view for data visualizations. Upon completion, we deployed nvPlayfair, which will help the PAT better understand the performance of workloads.

Acknowledgments

Thank you to our mentors, Allen Martin, Bill Armstrong, and Waqar Ali for providing valuable guidance regarding the implementation details throughout the project. We would also like to give a special thanks to our manager Mitch Luban for providing useful feedback at the end of each project phase, which helped keep our project on track to meet the end goal. Additionally, we want to thank Professor Mark Claypool for providing us with feedback and guidance on our development methodology. Last but not least, we want to thank all of the engineers from the Performance Architecture Team for helping us along the way.

Table of Contents

List of Figures

List of Tables

1. Introduction

NVIDIA is a technology company in the computer chip design and manufacturing industry known for its graphical processing units (GPUs) and system-on-a-chip (SoC) units. Specifically, Tegra is a SoC series developed by NVIDIA that is used in a wide variety of applications [1]. The Performance Architecture Team (PAT) is a cross-functional group within NVIDIA's Tegra System Software organization that aims to ensure that key performance indicators and underlying architectural requirements are met for each generation of the Tegra SoC. The PAT works closely with Tegra's ASIC organization and software product teams to define platform architecture and performance requirements for key markets, such as the automotive market.

One use case within the automotive market that the PAT is interested in is ensuring that a safety function executes before its deadline. It is required that the execution time of safety functions in NVIDIA's autonomous vehicles (AVs) meet a certain statistical confidence to guarantee the safety of passengers. Accordingly, AVs must have the ability to detect and apply the necessary maneuvers in time to avoid a collision. To ensure that safety requirement deadlines are satisfied, the PAT must understand the performance and determinism of the software involved. Additionally, if a safety function does not execute within a given deadline, the PAT needs to be able to detect and analyze the execution time outliers. There is currently no uniform process for teams to benchmark execution times and visualize outliers. Therefore, our team built nvPlayfair[1], which focused on benchmarking the execution time of a single workload consisting of a sequence of Linux system calls.

[1] See Appendix A for more information about the naming of nvPlayfair.

The goal of this project was to develop nvPlayfair, an automated platform for engineers to measure system software performance on the Tegra SoC. To achieve this goal, we divided our project into three objectives: (1) create a control application that receives test input from users to launch benchmark tests, (2) collect benchmark data on primitive kernel operations on the Tegra SoC, and (3) visualize and display the test results in a dashboard.

The control application is hosted on an internal NVIDIA Linux VM. Once the user specifies a benchmark configuration, the application runs the benchmark on a Tegra board and uploads the output data to a database. Users can view visualizations of their test results on Redash and analyze the performance of their workload.

To evaluate nvPlayfair, our team reviewed the completed project feature requirements and conducted user testing sessions. We implemented all of the basic requirements and most of our stretch goals into nvPlayfair. Additionally, we were met with positive feedback from two test users, suggesting that nvPlayfair is a viable product that can help NVIDIA engineers benchmark user-specified workloads, identify execution time outliers, and analyze their causes.

This report discusses our process of developing nvPlayfair. Chapter 2 provides context through examining related work at NVIDIA and introducing selected technologies and tools we used to develop nvPlayfair. Chapter 3 delves into key design decisions and provides a detailed description of our system architecture and implementation. Subsequently, Chapter 4 evaluates the usability of nvPlayfair. Lastly, Chapter 5 concludes our report and discusses future work.

2. Background

This chapter introduces background information about Tegra SoC, DRIVE AV safety functions, related work at NVIDIA, and technologies we used to build nvPlayfair.

2.1 Tegra SoC

The Tegra SoC integrates a central processing unit (CPU) with an ARM architecture, graphics processing unit (GPU), various input and output ports, and memory controller onto one package [2]. NVIDIA's Tegra processor family is widely used in the automotive market, specifically in AVs.

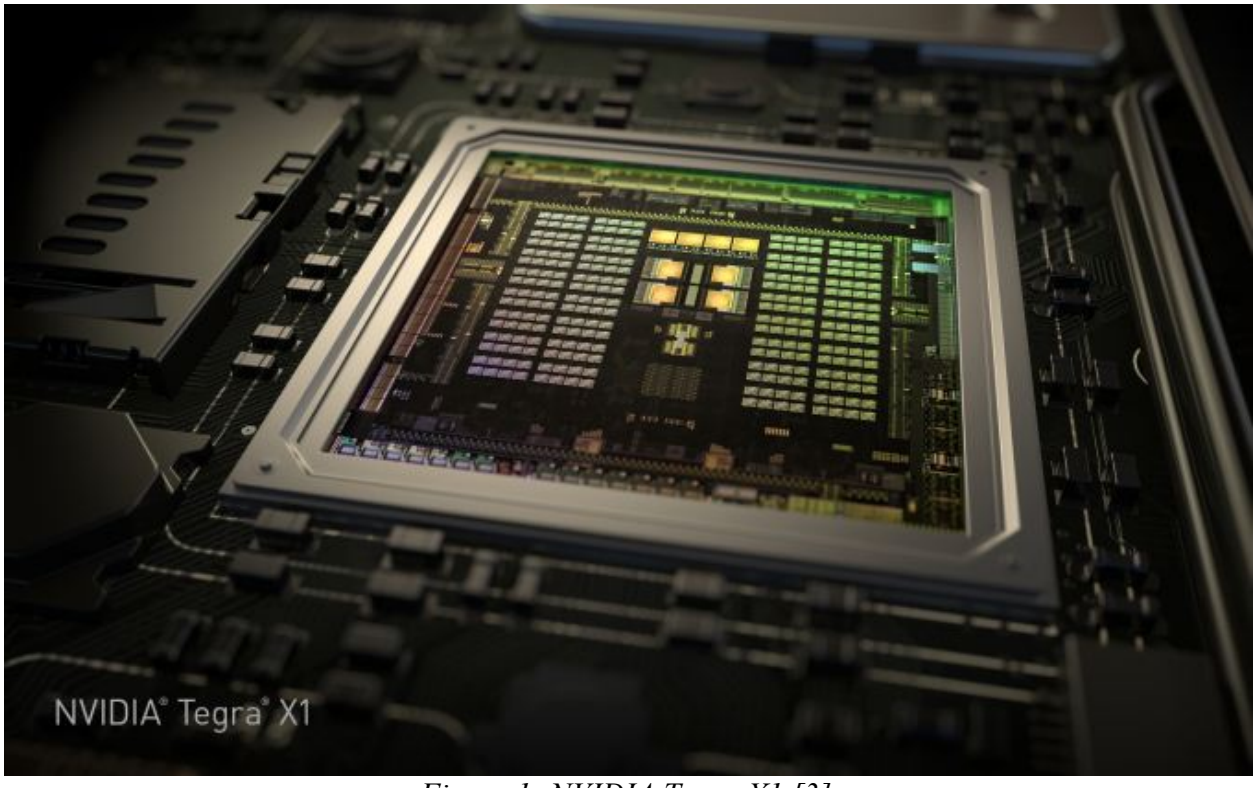

Figure 1: NVIDIA Tegra X1 [3]

Since our ultimate goal for nvPlayfair was to allow benchmarking safety functions in the Tegra chips that would be used in AVs, the Tegra SoC environment was our production

environment where we performed the benchmark test. Because development was done remotely, the board reservation was done through a virtual development tool called DriveFarm. DriveFarm is an internal virtual environment for NVIDIA developers to reserve, build, and deploy their programs. Using DriveFarm, developers can configure and flash a virtual Tegra board environment from their machine, which can then be utilized in our benchmarking framework.

2.2 DRIVE AV Safety Functions

The DRIVE team at NVIDIA works on developing and testing software for AVs, leveraging the features of the DRIVE AV platform. This platform consists of three core components: NVIDIA DRIVE OS, NVIDIA DriveWorks SDK, and NVIDIA DRIVE AV.

Closest to the hardware level, NVIDIA DRIVE OS is a foundational software stack consisting of an embedded real-time operating system, NVIDIA Hypervisor, NVIDIA CUDA libraries, NVIDIA TensorRT, and other modules that provide access to the hardware engines. DRIVE OS offers a safe and secure execution environment for applications, such as secure boot, security services, firewall, and over-the-air updates. Moving up the stack, NVIDIA DriveWorks SDK provides dedicated tools and interfaces for developers to implement real-time AV software, making use of the full throughput limits of the DRIVE OS platform [4].

The DriveAV-RoadRunner project is a self-driving car application based on DriveAV-Driveworks SDK. It uses configuration files as input to enable the car with different features such as perception, localization, planning and control [5].

NVIDIA DRIVE Perception enables perception of obstacles, paths, and wait conditions with a set of pre-processing, post-processing, and fusion processing modules. Together with

NVIDIA DRIVE Networks, these form an end-to-end perception pipeline for autonomous driving that uses data from multiple sensor types, such as camera, radar, and LIDAR [6].

NVIDIA DRIVE Mapping is an open and scalable solution that combines a flexible sensor suite, software, and software APIs, with high-definition maps provided by leading mapping companies [7].

NVIDIA DRIVE Planning enables lane and route planning, as well as behavior planning and control functions. DRIVE Planning empowers developers to innovate behavior planning as well as control and actuation AV functionalities [8].

Accordingly, there are certain safety function workloads within the aforementioned DRIVE AV software that engineers would like to benchmark using our framework. The benchmarking result can help the DRIVE team pinpoint outliers and analyze them to further determine the causes for such outliers.

2.3 Related Work

Engineers from the PAT worked on a project titled “RoadRunner QNX Event Mining” which focused on analyzing Linux and QNX traces on RoadRunner. First, the team recorded the frequency of kernel calls of cyclical tasks split across numerous hardware units by using ftrace on Linux and tracelogger on QNX. As depicted in Figure 2, each CPU core is running a set of tasks in a cyclical manner. For instance, Core 0 is executing tasks 1, 2, and 6 in a continuous rotation.

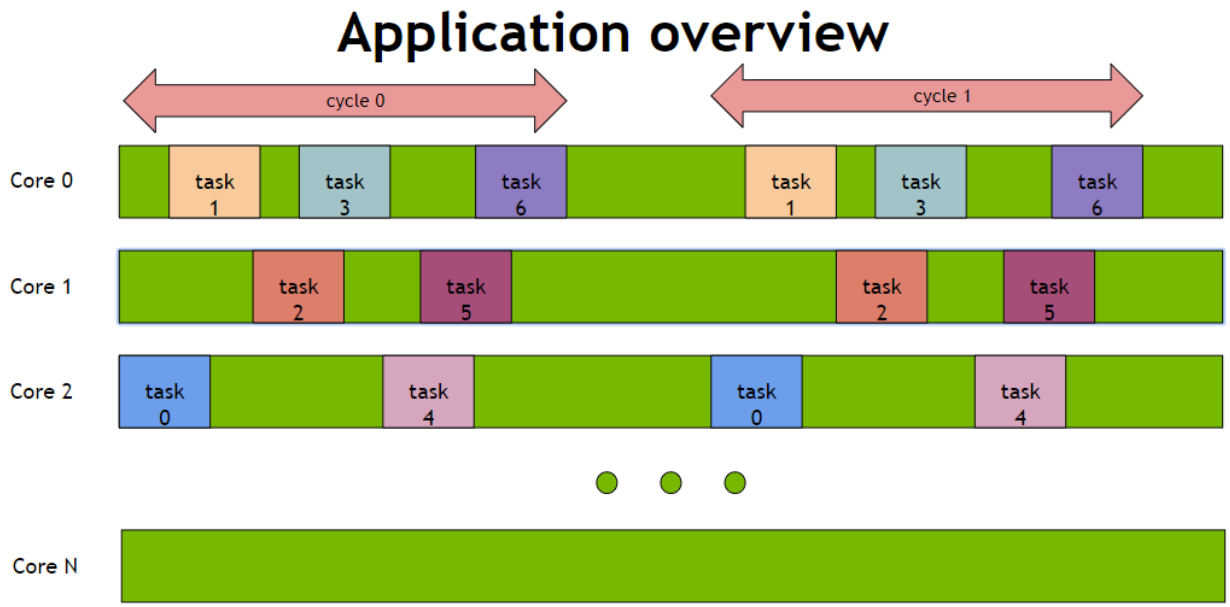

Figure 2: Application Overview [9]

Next, the team bucketized similar tasks into feature groups for scaling. Figure 3 shows a table of the results. The left column describes the three main features that were measured: (1) AV camera processing, (2) AV radar processing, and (3) AV planning / control / egomotion. The right column lists the average number of kernel calls made per second to run the feature.

Feature	kercalls / second
AV camera processing	35516
AV radar processing	8902
AV planning / control / egomotio	9948

Figure 3: Feature Results [9]

The event mining results were used to determine the most frequently used Linux system calls and libc functions, which enabled our team to prioritize the implementation for those system calls.

2.4 Tools and Technologies

In this section, we outline the technologies we used to build the benchmark framework and discuss the features of each tool. These tools provide a wide range of documentation and large extension libraries, which is helpful during the development stage. Refer to Chapter 3 for an explanation of how we integrated these technologies into the pipeline of this project.

2.4.1 React and Node.js

React is an open-source JavaScript library used for building single-page web applications [10]. React is a popular front-end framework choice given its ease of use and flexibility. nvPlayfair's user interface is written in React, integrated with the Semantic UI React framework, which offers themes and styled components.

React is a component-based framework, which enables developers to write modular code and encourages reusing components. A React component is defined as a JavaScript class or function that accepts inputs, such as 'props' and returns a visual of the React element on the user interface [11]. Each component has the ability to keep track of data that is stored in its own state. React utilizes a unidirectional data flow, where data can only be passed down from a parent component to a child component as pictured in Figure 4. In this figure, the top-level component is passing its props down to the lower-level components, denoted by the orange arrows. The green arrows indicate that each component has a state that stores data.

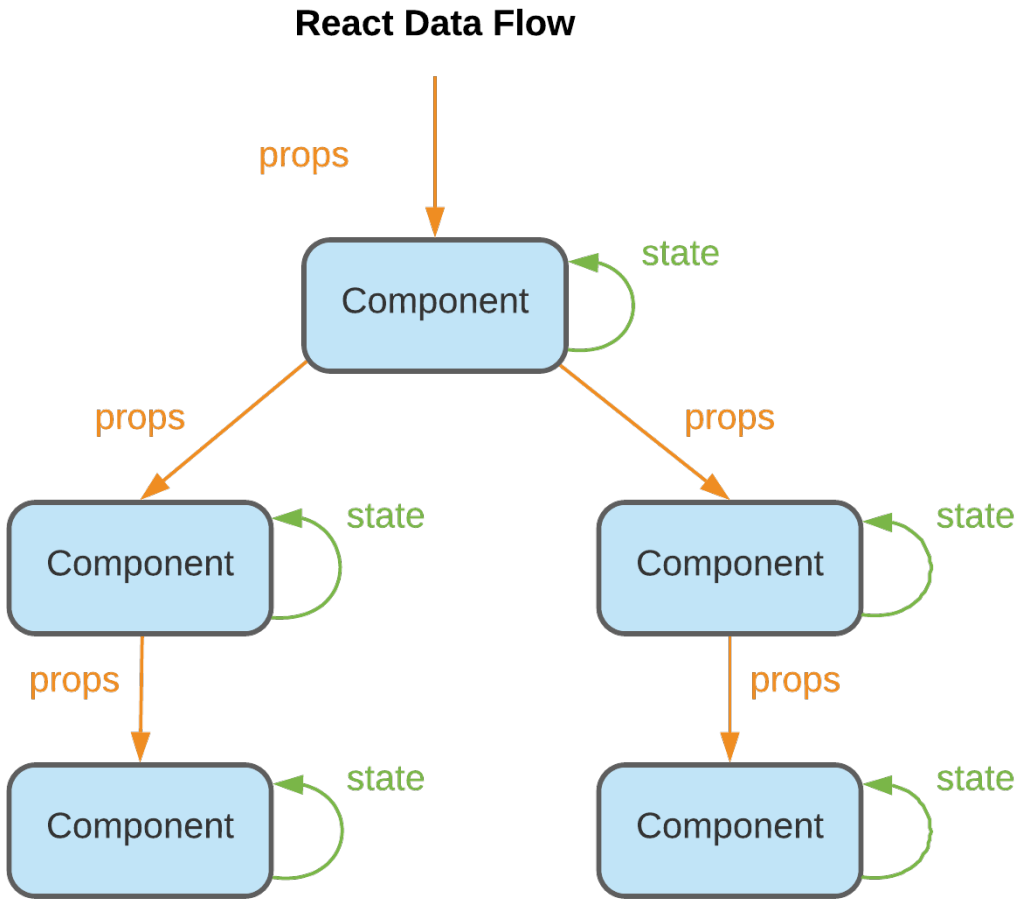

Figure 4: React Data Flow. Adapted from [12].

nvPlayfair uses Express.js, a Node.js web application server, to serve our React application build folder on an internal Linux VM.

2.4.2 Django and MySQL

Django is an open-source Python-based framework for developing web application back-ends. It was introduced in 2015 and became one of the most popular web frameworks in 2020 [13]. Django is well-established and has an active support community. nvPlayfair's back-end is

developed in Django due to its simplicity, flexibility, reliability, and easy integration with other frameworks.

Django is known for its principle of rapid development methodology, which means it provides tools for developers to quickly prototype, write code, and test. It also follows the Don't Repeat Yourself (DRY) principle by providing mechanisms to reuse existing code, thus enabling developers to focus on creating new features. Figure 5 shows a diagram of the rapid application development methodology.

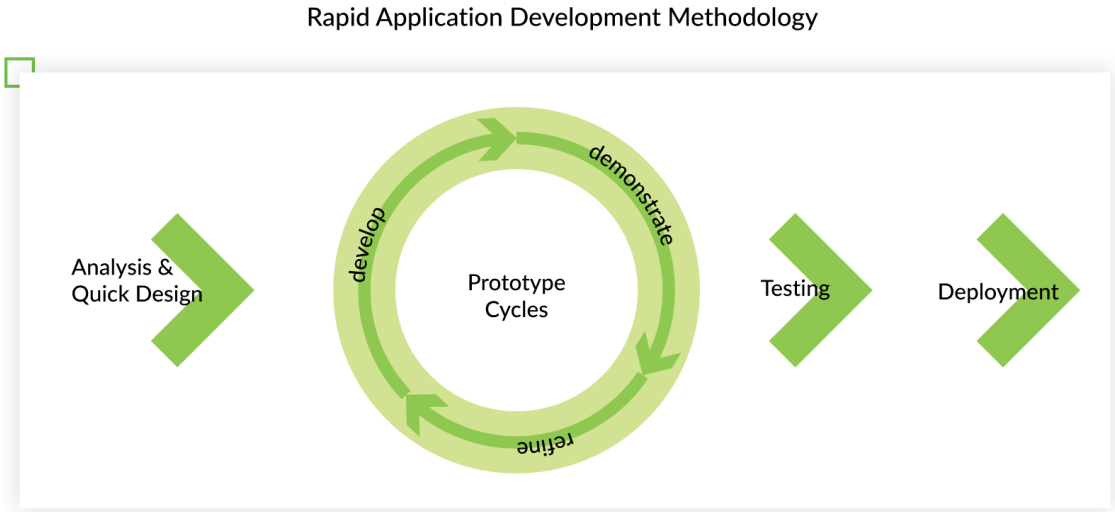

Figure 5: Django Rapid Application Development Methodology [14]

Out of the box, Django provides its own web server, an Object Relational Mapper (ORM), middleware support, Python unit test framework, and all the essentials needed to run a project, including setting configuration, a simple database, and HTTP libraries. Django's syntax is simple, and its package is widely supported. Additionally, it provides an admin panel that allows the user to view and edit the application.

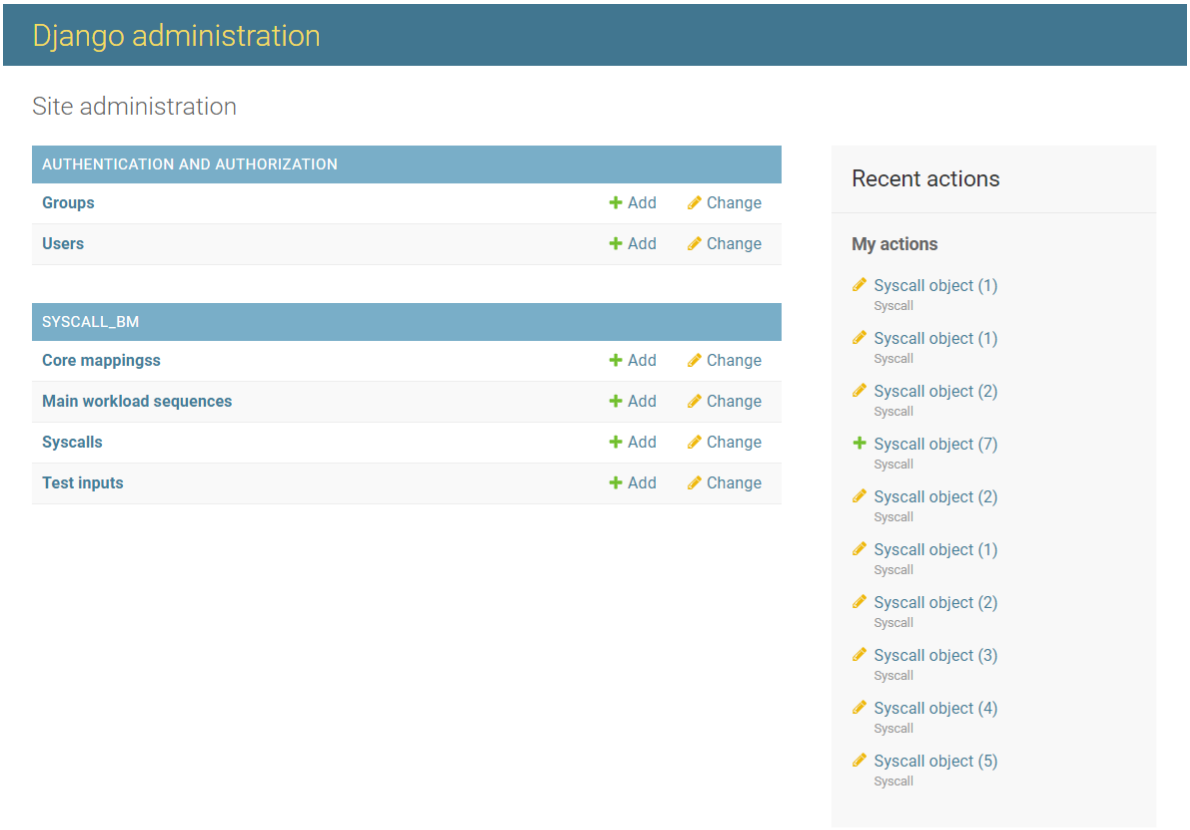

Figure 6: nvPlayfair's Django Admin Interface

Django also provides integration with many supported databases, including a MySQL database that is used in the project. Django also provides support for multiple databases at the same time if the project needs to scale in the future. Additionally, Django can integrate with other Python-based benchmark code that is performed on the Tegra board. It also provides documentation and integration guides with the React framework.

2.4.3 MagLev

MagLev is a complete hardware and software AI infrastructure, internal to NVIDIA, built to support the complex process required to develop and validate AV software. MagLev is a full-

stack data center solution composed of three core components: the MagLev Hardware Infrastructure, the MagLev Data Center Backbone, and the MagLev Software Infrastructure.

First, the MagLev Hardware Infrastructure is a base reference rack-level hardware architecture, encompassing GPU nodes (DGX), CPU nodes, network switches and topology. Second, the MagLev Data Center Backbone is a base data center management software layer comprising cluster orchestration software, storage software, monitoring software. Third, the MagLev Software Infrastructure consists of four software components, primarily services, that integrate to enable the complete AI development workflow from data ingestion to model deployment.

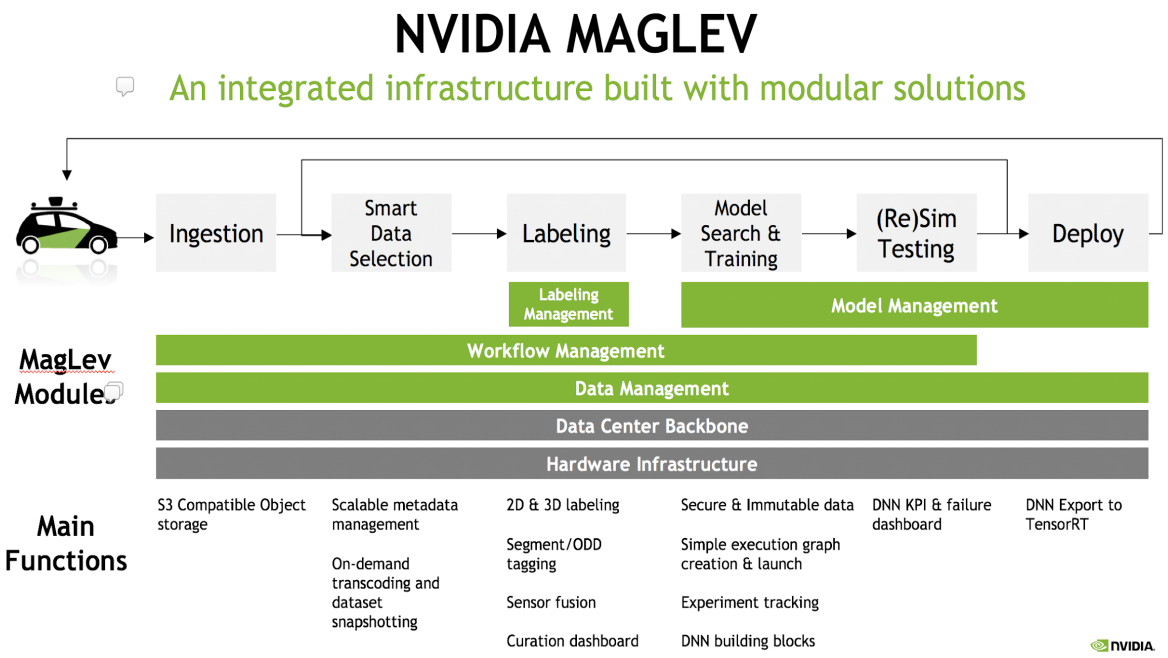

Figure 7: NVIDIA MagLev [15]

In our project, we utilize MagLev as our data storage solution to store benchmarking results given its scalability, ease-of-use, and maintainability. There are two key concepts about MagLev: Data Lake and Catalog.

The Data Lake is an abstraction of how the data is stored. Essentially, the primary way data is stored is in a key value store. The solution is normally either Swift Stack or S3.

The primary way most users interact with the Data Lake is through the catalog. Data in the catalog must be structured. In other words, the data being uploaded must have a schema associated with it. Some common data formats include CSV (Comma-Separated Values), JSON (JavaScript Object Notation), and Parquet. MagLev accepts most self-describing file formats where the schema can be read from the file, which is then used to create the tables. As an example, users can upload to MagLev data frames containing both the data and column names, and MagLev will automatically infer the data schema [15].

The primary abstraction of the Catalog is a database and table. Similar to that of other big data solutions like Hadoop, a table is made of a few pieces of metadata: schema, paths, files, and partitions. The schema describes the structure of the data behind the table. The paths represent the pointers to directories in the Lake that make up the table's data. The files represent the serialized data in the directory paths for the AV Data Platform, which is always Parquet. And optionally, a table can be partitioned on certain columns making use of Lake filesystem paths [15].

All of this metadata is stored in a relational database. The actual raw data, however, is stored in the Parquet files directly in the Lake. This allows the system to scale much better than a normal relational database.

The Catalog also utilizes a GRPC service, which serves four purposes. First, it validates that a user making the request can read, write, and delete the table in the call. Second, when adding data to the catalog, it confirms that the new data follows schema migration rules. Third, it validates that the data is readable by the system, not corrupt or an invalid file format. Last but not least, it returns credentials to the client, and allows them to get/upload/delete data for the request table [15].

Note that this service never directly reads the raw data behind the table. This allows it to scale to handle many users. Developers working with MagLev mostly interact with this service through several native clients in different programming languages (Python, GoLang, Scala, Java, MagLev CLI, Spark).

For querying the data, the MagLev Data Platform provides distributed SQL engines that run on top of the catalog. This makes it quick to do things as you expect in a normal relational database such as select, join, group, etc., but at scales much larger than a normal relational database. Two main examples that NVIDIA utilizes are Presto, and SparkSQL.

Presto is an open-source distributed SQL query engine for running interactive analytic queries against data sources of all sizes ranging from gigabytes to petabytes [16]. Presto was designed and written for interactive analytics and approaches the speed of commercial data warehouses while scaling to the size of organizations like Facebook. Presto allows querying data where it lives, including Hive, Cassandra, relational databases or even proprietary data stores. A single Presto query can combine data from multiple sources, allowing for analytics across an entire organization. It also supports querying MagLev. In fact, our dashboarding tool, Redash, connects directly to NVIDIA's Presto cluster on top of the existing Catalog(s).

Spark SQL is a Spark module for structured data processing. Unlike the basic Spark RDD API, the interfaces provided by Spark SQL provide Spark with more information about the structure of both the data and the computation being performed. Internally, Spark SQL uses this extra information to perform extra optimizations. There are several ways to interact with Spark SQL including SQL and the Dataset API [17].

On top of these aforementioned distributed SQL engines, MagLev Data Platform also utilizes the Query Service. The Query Service is an internal GRPC service that sits on top of Presto, SparkSQL, and possibly other query engines in the future. It accepts SQL queries and redirects them to the requested query engine. One benefit of this approach is it can also store the results of the query back into the catalog. This allows for ETL jobs to be written that turn a query into a table that can be consumed by other users [15].

2.4.4 Redash

Redash is a browser-based data visualization tool with the ability to connect to a variety of data sources. Within nvPlayfair, Redash is connected to a MagLev data lake and enables users to create data visualizations tied to SQL queries. Users can query data from tables using the Redash interface shown in Figure 8. Redash provides real-time query support and collaboration features.

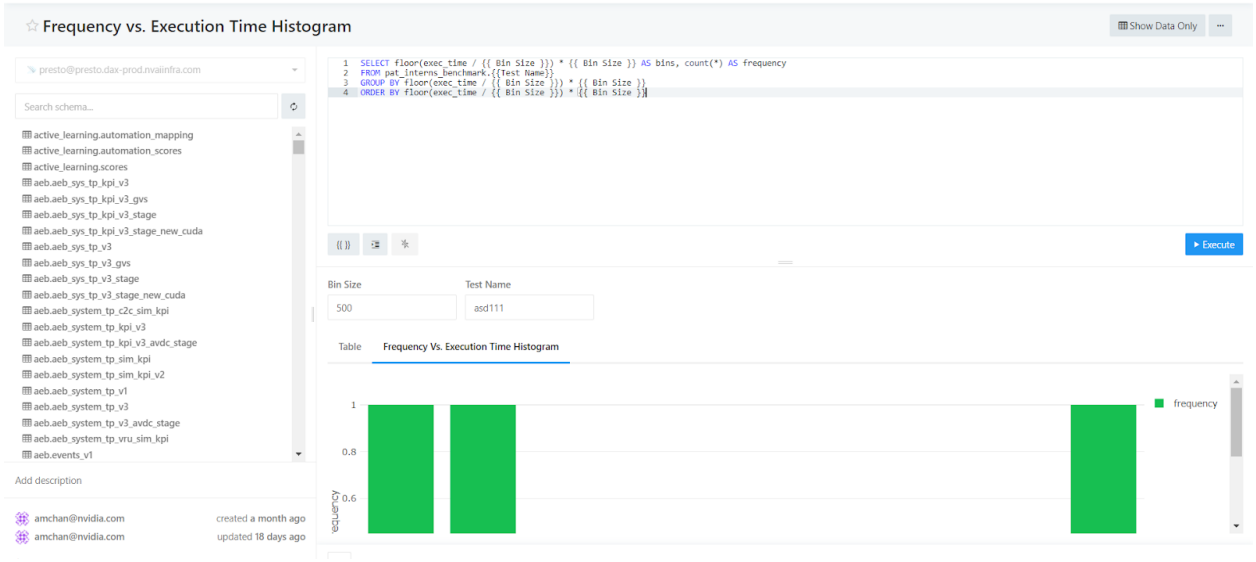

Figure 8: Redash Query Interface

Figure 9 shows the visualization configuration interface that populates data based on the result of a specific query.

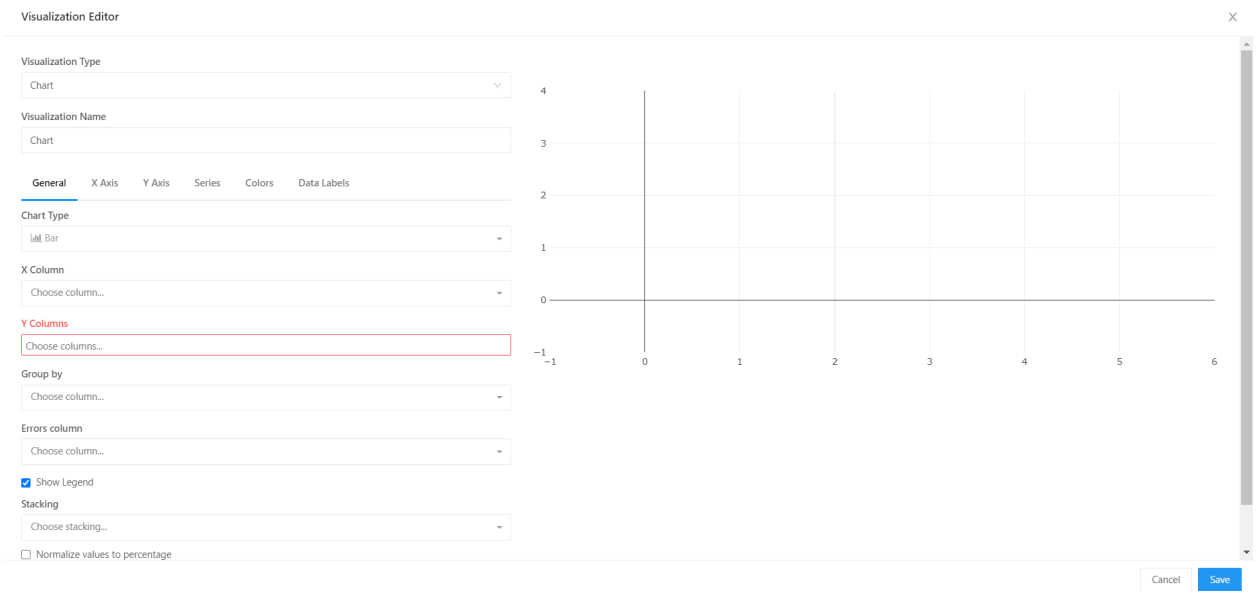

Figure 9: Redash Visualization Editor

3. Methodology

The purpose of this project was to develop nvPlayfair, an automated platform for engineers to measure system software performance on the Tegra SoC. To achieve this goal, we implemented the following objectives:

1. Create a control application that receives test input from users to launch benchmark tests.
2. Collect benchmark data on primitive kernel operations on the Tegra SoC.
3. Visualize and display the test results in a dashboard.

Figure 10 showcases our project workflow. We describe this workflow in more depth in the following sections.

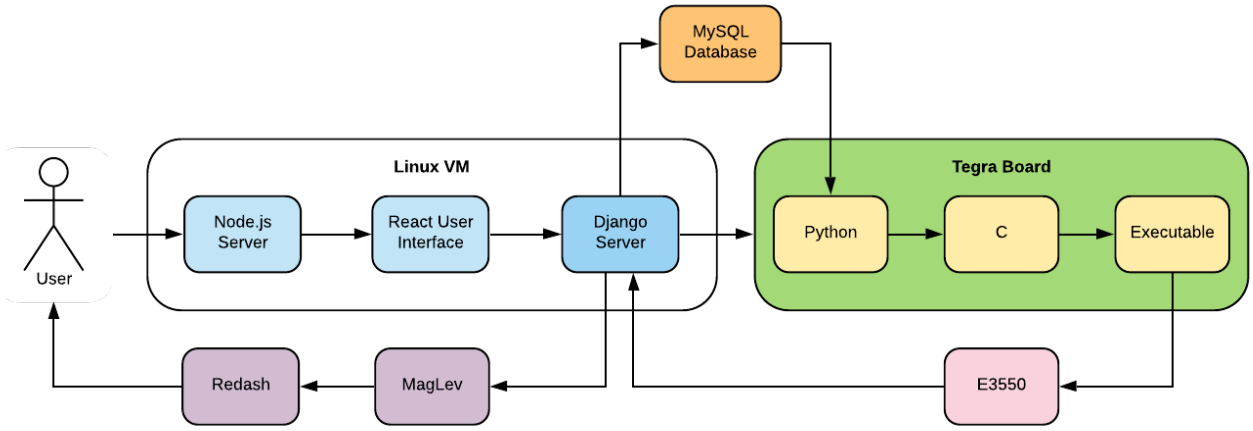

Figure 10: Project Workflow

3.1 Development Process

The Agile development methodology is a development process that involves breaking up a project into several stages and doing continuous improvement and iteration at every stage. The process includes building software incrementally from the start of the project, instead of delivering it all at once near the end. Since the Agile development process focuses on iterative and incremental progression of a project, it allows constant feedback and improvement to ensure the project aligns with stakeholders' expectations. Agile development is broken down into sprints, which can range between two to five weeks, depending on the team. As shown in Figure 11, during each sprint, the team follows the planning, coding, feedback iteration to develop features for the project.

Figure 11: General Agile Development Process [18]

Our team followed Agile methodologies and two-week sprints for development. Requirements, plans, and results were evaluated continuously so that we had an opportunity to respond to changes quickly. Our team chose to follow two-week sprints after considering our timeline of about seven weeks. Additionally, two-week sprints provided less overhead with sprint processes, such as planning and retrospective compared to a one-week sprint. Figure 12 illustrates our team's process for each sprint.

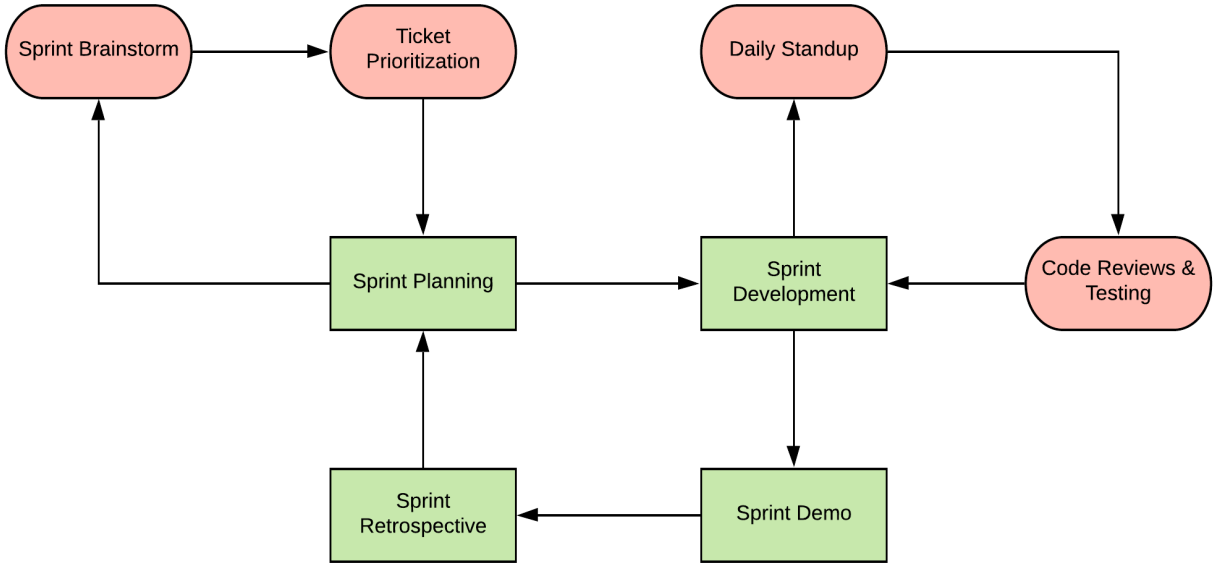

Figure 12: Team Process

The sprint brainstorm was a 30-minute meeting when our team came together to brainstorm features and ideas for the next sprint. Following the sprint brainstorm, there would be a one-hour follow-up meeting for our team to discuss which tickets would be implemented in the next sprint and which would go to the backlog. Ticket priority and implementation time would be taken into consideration for which ticket would be implemented in the next sprint. Tickets were added to our team's scrum board as shown in Figure 13. We used Microsoft Teams' Kanban board feature

as our scrum board. Our scrum board was divided into four sections: Backlog, Sprint 1, Sprint 2, and Sprint 3. Each section consisted of tickets that were estimated and assigned to people. Once a ticket was implemented into nvPlayfair, it was marked as complete.

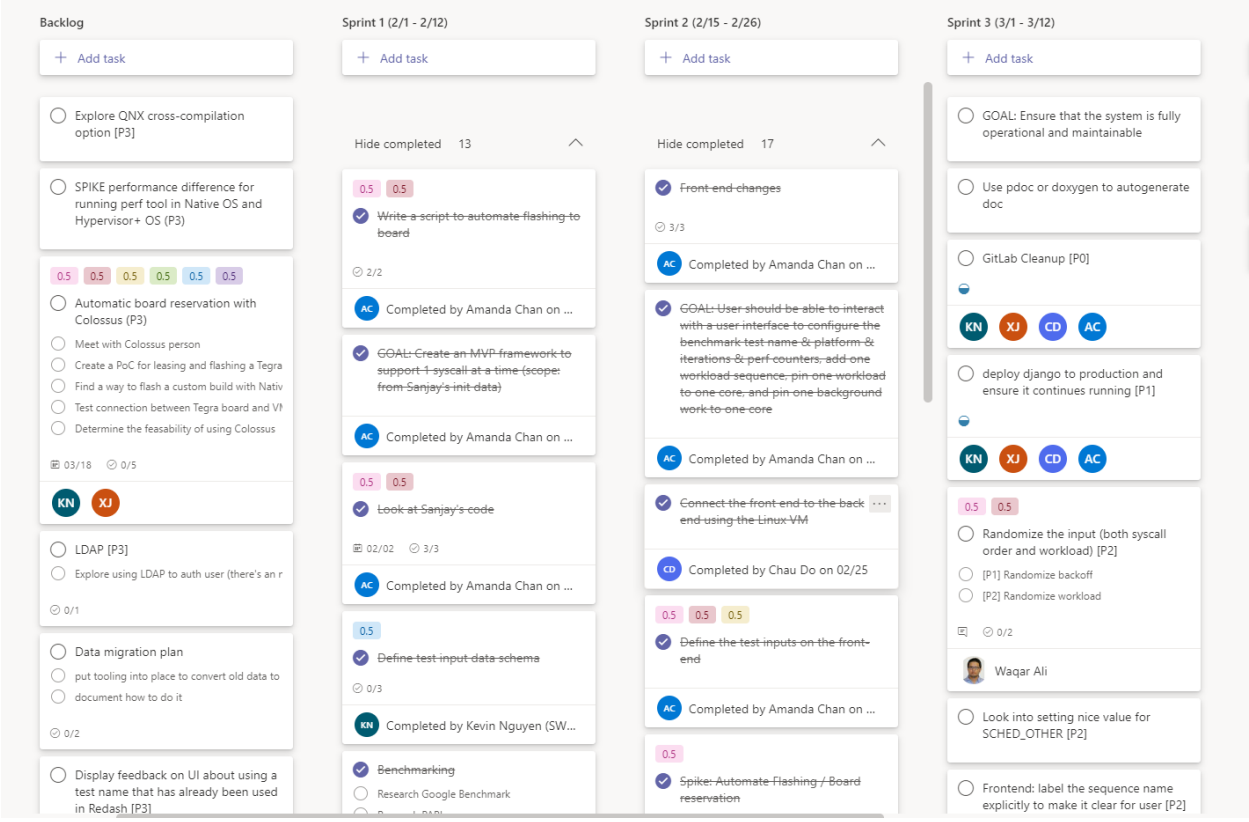

Figure 13: Scrum Board

After the sprint planning phase, which was at the start of the sprint, our team moved to the development phase. The development phase included daily standup where we reported progress to the mentors and advisors. Development included pair programming, mutual code reviews, and testing.

At the end of the sprint, our team presented a 30-minute demo showcasing what we accomplished during the two-week period. Questions and feedback were taken to the sprint brainstorm for further consideration. After the sprint demo, there was a 30-minute sprint retrospective meeting. In this meeting, our team provided feedback about the previous sprint,

including what the team should start doing, stop doing, or continue doing. We used a tool called RemoteRetro[2] to facilitate our sprint retrospective. RemoteRetro provided an interactive platform for giving feedback and generating action items.

3.2 Control Application

The purpose of the control application is to enable users to launch benchmark tests based on their specific input. The user-friendly interface makes the application easy to use. Our final control application was written using React and Semantic UI for the front-end and Django for the back-end. Figure 14 shows a high-level overview of the control application architecture. On the Linux VM where both the front-end and back-end is hosted, we utilized a simple Express.js web server, managed by PM2[3], to serve the React build to the user. The black circle represents the user's entry point into nvPlayfair. Upon submitting a benchmark test, the front-end will send a POST request to the Django back-end, which triggers the entire test deployment pipeline.

[2] https://remoteretro.org
[3] https://pm2.keymetrics.io/

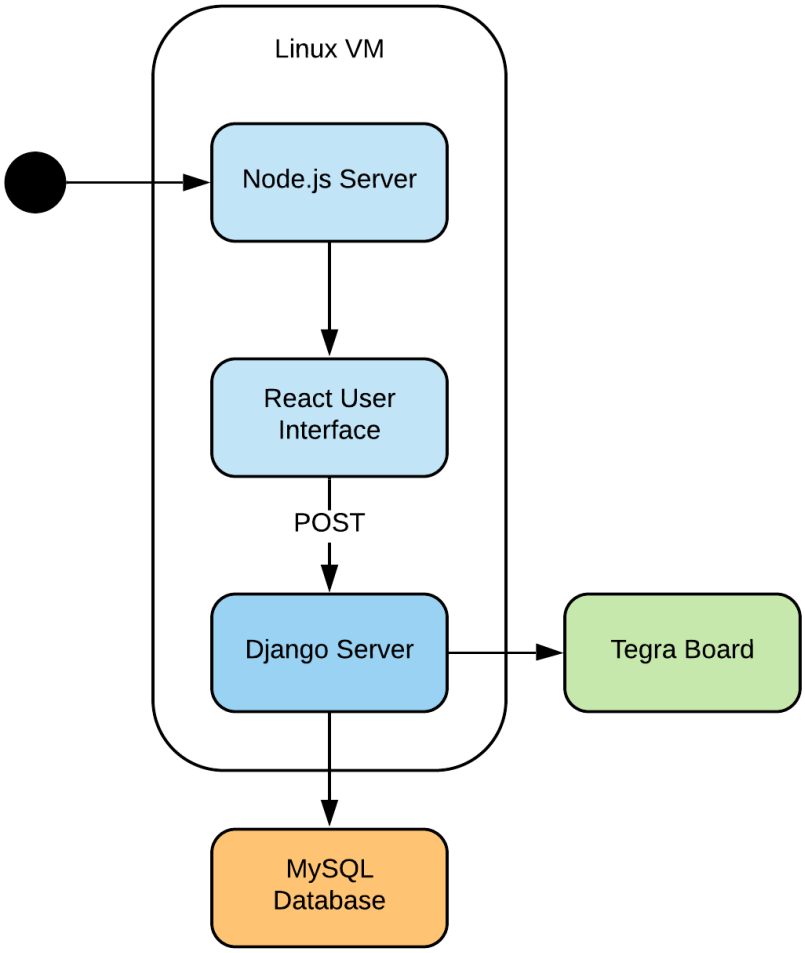

Figure 14: Front-End to Back-End Communication

In the next sections, we discuss key design decisions along with our final implementation.

3.2.1 Design Decisions

The control application requires an interface where users can easily configure the benchmarking tests and workload generations. Accordingly, there were three potential solutions for the front-end: React, Jenkins, and Splunk. React is a JavaScript front-end framework for building user interfaces. Jenkins is an open-source automation server, used for automating the

Continuous Integration/Continuous Deployment (CI/CD) process in software development. Splunk is a deployment pipeline, which also has an integrated dashboard feature.

Table 1 shows a brief comparison summary between React, Jenkins, and Splunk.

Requirements	**Jenkins**	**Splunk**	**React**
Ability to accept user input	Yes	Yes	Yes
Flexibility in adding new user inputs	Yes	Yes	Yes
Ability to add complex user inputs field	No	No	Yes
Ability to validate user inputs	No	Maybe, since Splunk can run scripts	Yes
Ability to transform user inputs	Yes, but not flexible	Yes, Splunk can run other scripts	Yes
Ease of use	Users need to get access to the Jenkins pipeline, and be familiar with it	Users need to request access to Splunk	Yes, since it is just a typical web application
Compatibility with our existing tools	Maybe, will need to do a lot of setup and integration work	Maybe, will need to do a lot of setup and integration work	Yes, easily configurable
Hosting	Internal Blossom	Internal Server	Internal Linux VM

Table 1: Comparison Chart Between React vs. Jenkins vs. Splunk

While it is possible for Jenkins and Splunk to accept user input, they are not built for accepting complicated benchmarking test input. Instead, Jenkins is geared towards building CI/CD pipelines which only requires simple user inputs, while Splunk is geared towards

monitoring and analyzing machine-generated big data. React, on the other hand, is specifically built for creating customizable user interfaces.

Therefore, we decided to utilize React to build our control application user interface due to its flexibility and customizability. Since React provided our team with full control over the front-end, the process of adding new features was simple. Additionally, our team had previous experience with React so there was a smaller learning curve compared to us utilizing Jenkins and Splunk.

3.2.2 User Interface Design

Over the course of the project, we came up with four user interface designs. First, we came up with Version I of the user interface during the first week of the second sprint. There are two assumptions that we made for this version. First, users would know how many cores a chosen system has. Second, users would be aware of the list of supported system calls. Here are the known requirements at the time:

- Users can specify their test's name.
- Users can specify the platform they would like the test to run on (Linux or QNX).
- Users can specify the sequence of system calls they would like to benchmark.
- Users can specify the list of performance counters (perf counters) they would like to collect.
- Users can specify the number of iterations to run the defined sequence.

Figure 15 depicts a mock up of the first version of our UI.

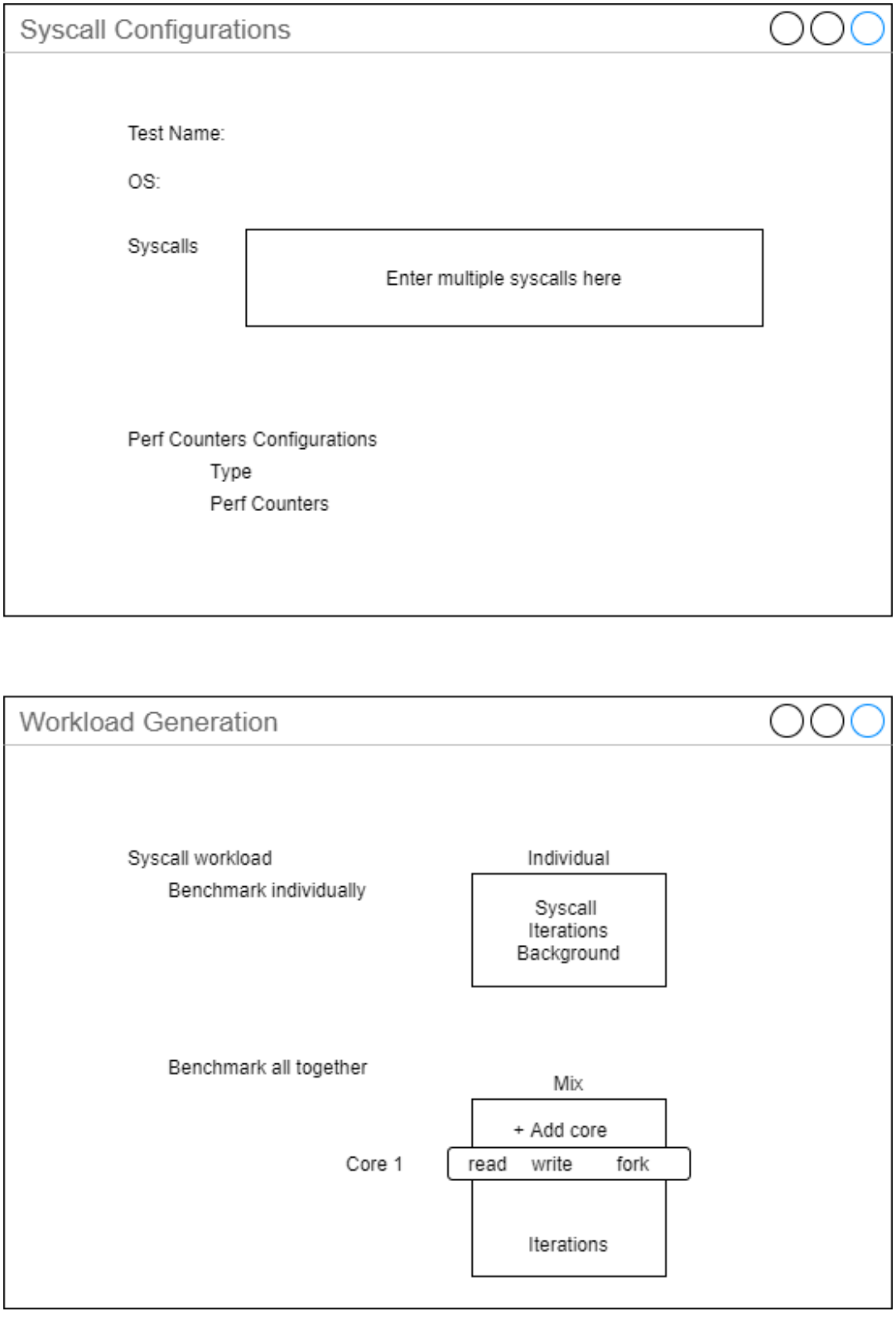

Figure 15: First Mock UI Version

After receiving feedback from our mentors and manager, we came up with Version II of the user interface. Here are the new requirements:

- Change from “sequence of system calls” to “workload” since our final goal would be geared towards high-level workloads.
- Users can specify a mix of different system calls or libc functions.
- Users can specify running the workloads on different cores.
- Users can select a predefined background workload.
- Users can pin workloads to cores.

These are the assumptions we made:

- Users are aware that the framework will only support running benchmarking tests on Native Embedded Linux flashed to a E3550 Tegra board.
- Users know what the predefined background workloads are.
- Users are limited to the system calls and libc functions implemented in the benchmarking framework.
- The framework is allowed to hardcode the number of cores a specific system has. Specifically, Native Embedded Linux has 8 cores available for use.
- Users are aware that each core can only have one workload (either one main workload or one background workload).

Figure 16, 17, 18, and 19 below show all the features of the second version of our UI, which is the final product of our second sprint. Figure 16 shows the basic configuration. Figure 17 and Figure 18 shows two parts of the core configuration, where users can pin the main

workloads and background workloads respectively to specific cores. Figure 19 shows the perf counters configuration, where users can specify which perf counters they would like to collect on the main workload during the benchmark test.

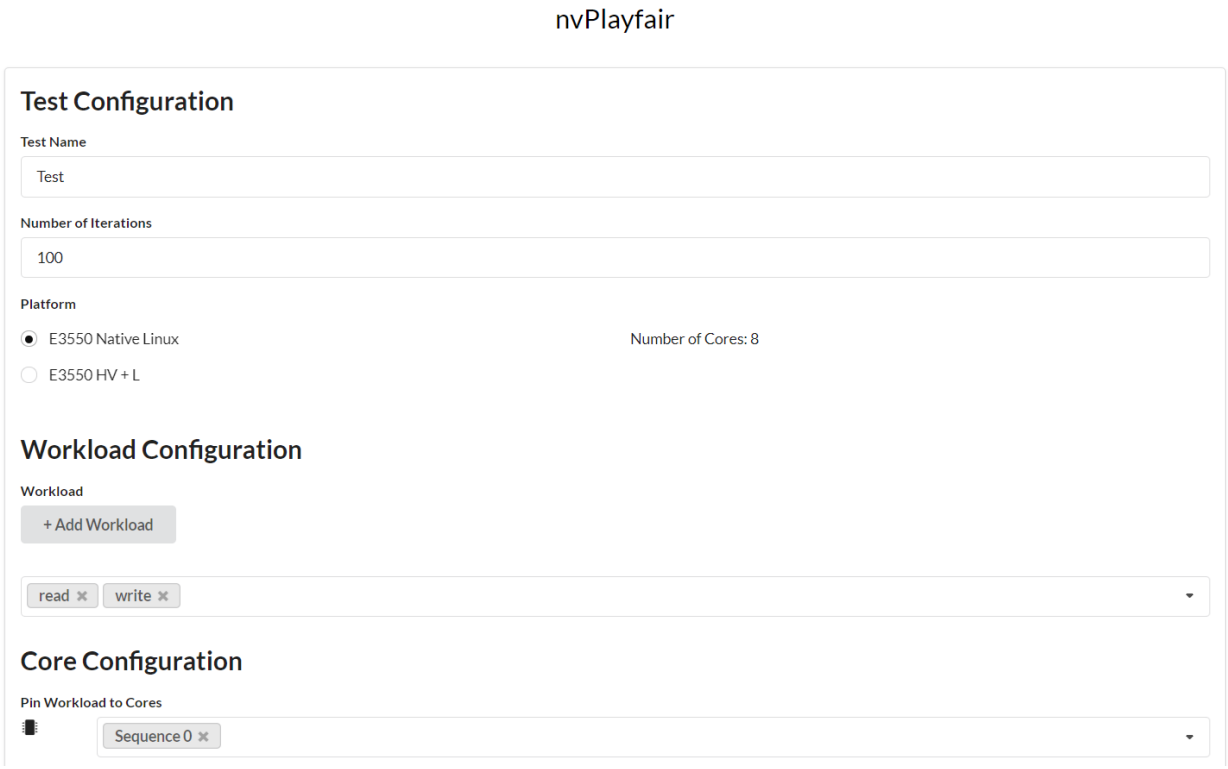

Figure 16: Basic Configuration (Second UI Version)

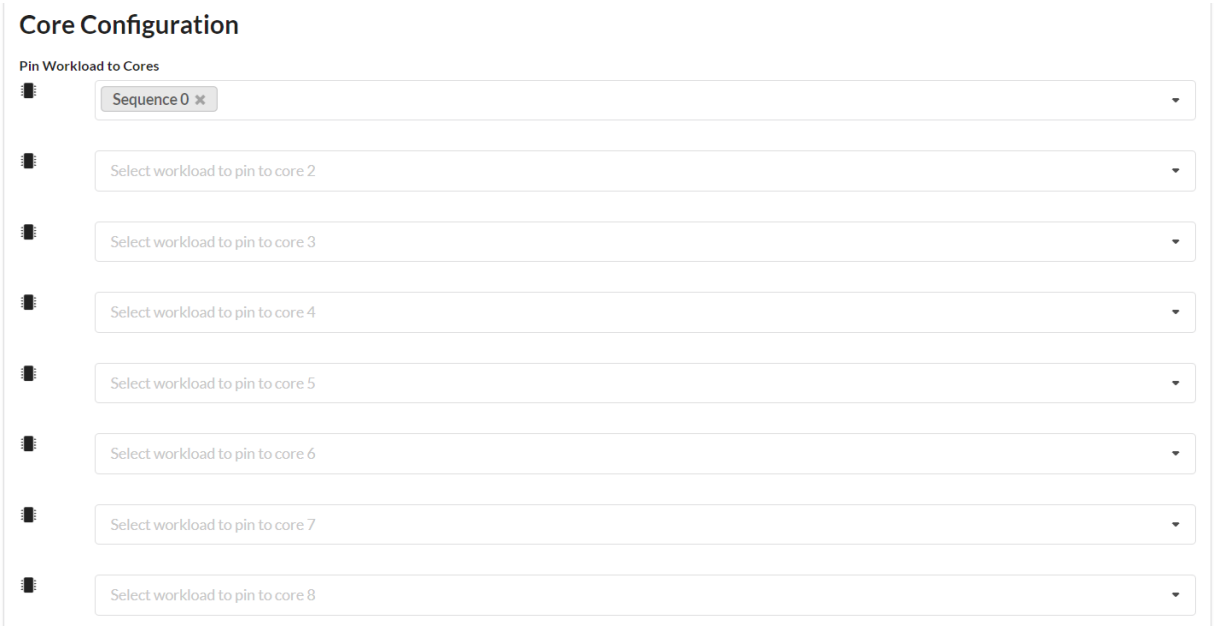

Figure 17: Core Configuration - Pin Main Workload to Core (Second UI Version)

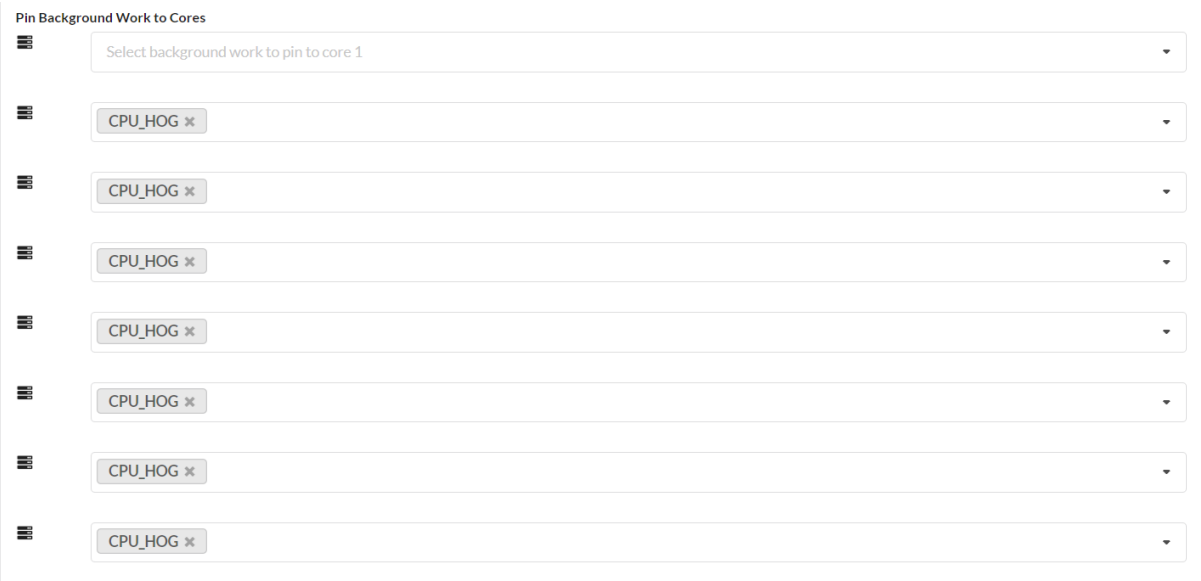

Figure 18: Core Configuration - Pin Background Workload to Core (Second UI Version)

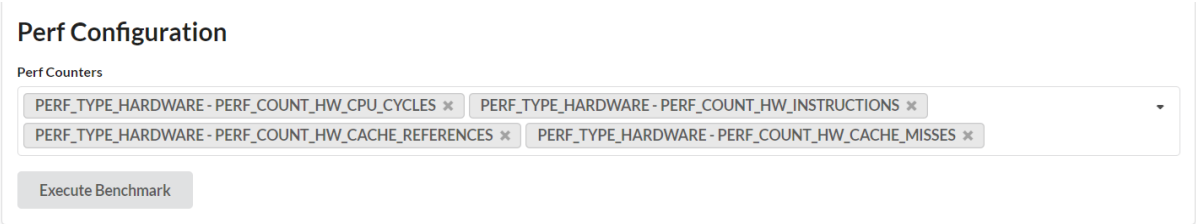

Figure 19: Perf Counter Configuration (Second UI Version)

Then, we came up with Version III of the user interface after receiving feedback from our mentors, manager, Professor Claypool, and other members in the PAT during the Sprint 2 demo. Here are the new requirements:

- Users can define their custom background workloads.
- Users can only specify one main workload to benchmark.
- Users can define the scheduling policy and priority for each workload per core.

- Users can pin multiple background workloads per core.
- Users are limited to pin the main workload to core once.
- There should be a random backoff time between each iteration of running the main workload.
- Users can specify the Tegra board IP address that they would like the tests to be run on.

These are the assumptions we made:

- Users are aware that they have to flash a custom Native Embedded Linux to enable ARM performance monitoring units to their Tegra board in order for the tests to run correctly.

Our final UI version can be found in Section 3.2.4.

3.2.3 Input Data Schema

Over the course of the project, we came up with four versions of the input data schema.

Our first version of the input data schema was designed during the first sprint. Here are the requirements:

- Users can specify the individual system calls they would like to benchmark.
- Users can specify the iterations per system call.
- Users can specify how to group the benchmarking result together.

Figure 20 shows the data schema as a CSV file format.

```
1    syscall_name, iterations, group_id
```

Figure 20: Version I - Input Data Schema

For this first version, we intentionally kept the data schema simple enough to efficiently cover the requirements. We could have the format in JSON, but since this schema will be processed by C++ code in the framework, JSON format would make the work heavier. Accordingly, given that there are still many changes to be made (since our initial framework is just a proof of concept), we decided to not invest much effort on this data parsing yet.

Our second version of the input data schema was designed during the first week of the second sprint. Here are the requirements:

- Users can specify how many iterations per syscall or libc function to run.
- Users can specify the list of perf counters to collect.
- Data schema should be flexible and easy to extend.

Figure 21 shows the JSON data schema, along with explanations:

```
{
    "syscalls_input": [
        {
            "syscall_name": <string>,
            "iterations": <integer>,
            "perf_counters_list": [
                {
                    "type": <string>,
                    "perf_counter": <string>
                }
            ]
        }
    ]
}
```

Figure 21: Version 2 – Input Data Schema

Descriptions**:**

syscalls_input = list of syscalls / libc input entries to benchmark

syscall_name = name of the system call or libc function. Current options are limited to those that we have implemented.

iterations = number of iterations the benchmark test should run

perf_counters_list = list of perf counters config pairs

type = type of perf counters

perf_counter = the exact perf counter

For the third version, we switched to JSON format for two reasons. First, JSON format is more extensible for future use cases. Second, for the benchmarking code, we already switched to

using Python to generate C code, so the issue of parsing JSON is resolved thanks to Python's JSON parsing libraries. Here are the new requirements:

- Users can specify multiple sequences of system calls or libc functions.
- Users can specify the test name.
- Users can only run tests on E3550 Native Linux.
- Users can pin workload to cores.

Figure 22 shows the JSON data schema, along with a description of each field.

```
{
    “test_name”: <string>,
    “iterations”: <int,
    “platform”: “E3550 Native Linux”,
    “sequences”: [
        {
            “id”: <int>,
            “syscalls”: [<string>]
        }
    ],
    “coreMappings”: [
        {
            “id”: <int>,
            “sequences”: [<int::sequence_id>],
            “backgroundWork”: [<string>]
        }
    ],
    “perfCounters”: [
        {
            “type”: <string>,
            “perf_counter”: <string>
        }
    ]
}
```

Figure 22: Version 3 – Input Data Schema

Descriptions**:**

test_name = user-specified test name

iterations = number of iterations to run the workload for benchmarking

platform = platform to run the tests on. Fixed to “E3550 Native Linux”

sequences = list of syscalls / libc sequences

id = auto-generated sequence id

syscalls = list of the system call or libc function names. Current options are limited to those that we have implemented.

coreMappings = list of core settings to pin workload to cores

id = auto-generated core id, auto-increment, start from 1

sequences = list of ids of defined sequences

backgroundWork = list of strings of predefined background work. Options are ["CPU_HOG", "MEMORY_HOG", "FORK_BOMB"]

perfCounters = list of perf counters config pairs

type = type of perf counters

perf_counter = the exact perf counter

Last but not least, we fulfilled the requirements for the final input data schema version:

- Users can specify schedule policy and priority
- Users can only pin the main workload thread once
- Users can specify the board's IP address which is already flashed with Native Linux for the benchmarking tests to run on

Section 3.2.4 explains the format and description of our final data schema.

3.2.4 Front-End Implementation

The purpose of implementing a front-end into nvPlayfair was to enable users to easily configure test specifications. nvPlayfair uses a form with various input and selector fields to accept input from the user. For ease of use, the form is divided into six sections: Test Configuration, Main Workload Configuration, Background Workload Configuration, Core Configuration, Scheduling Configuration, and Perf Configuration.

Figure 23 depicts the Test Configuration component. In this section, users are required to specify a test name, a number of iterations to run their benchmark test for, a Tegra board IP address, and the type of platform to run the benchmark tests on. The test name field must be unique as it is how nvPlayfair keeps track of a specific run. The number of iterations specified must be greater than 0 and less than 100,000. Capping the maximum number of iterations to 100,000 allows the user to run a large test without having to wait a long time for it to finish. nvPlayfair requires users to specify their board IP address so that it knows which board to run the tests on. Enabling this feature allows multiple users to run benchmark tests using nvPlayfair. Board IP addresses must match one of the following formats to be considered valid: 10.255.14.*, 10.255.16.*, or 10.255.18.*, where the asterisk can be any combination of numbers up to three digits in length. The last option in the Test Configuration section is the platform type. nvPlayfair currently offers two platform options on the user interface: E3550 Native Linux and the E3550 Hypervisor + Linux (E2550 HV + L). It is important to note that there is only functionality built in for the E3550 Native Linux option and selecting E3550 HV + L will not yield accurate results.

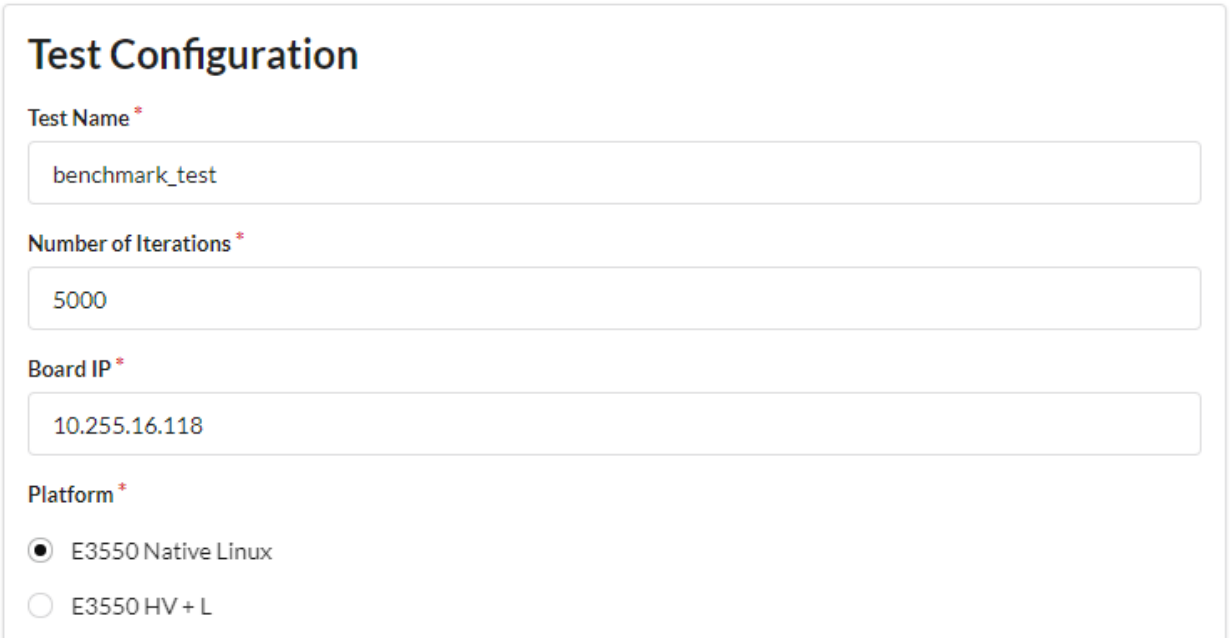

Figure 23: Test Configuration

The next major part of the form is the Main Workload Configuration section. In this section, users can add a main workload sequence to benchmark as shown in Figure 24. Only main workload sequences can be benchmarked in terms of execution time and perf counters. Main workload sequences are currently made up of a sequence of Linux system calls, including clock_gettime, nanosleep, poll, read, select, and write. The order in which a user selects the system calls is the order in which they will be benchmarked. Users can only add one main workload sequence as nvPlayfair does not currently support benchmarking multiple main workloads.

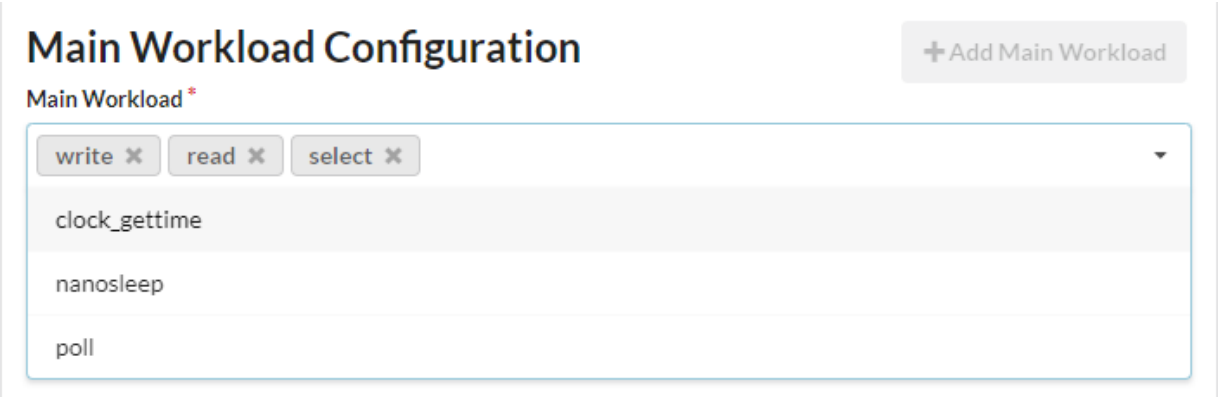

Figure 24: Main Workload Configuration

Similar to the Main Workload Configuration section, in the Background Workload Configuration section, users can add a custom background workload as shown in Figure 25. Background workloads cannot be benchmarked and will be running alongside the main workload in an attempt to interfere with the main workload. Users can add any number of custom background workloads by specifying sequences of system calls. Within each background workload, system calls are selected and run in the order that the user specified them in.

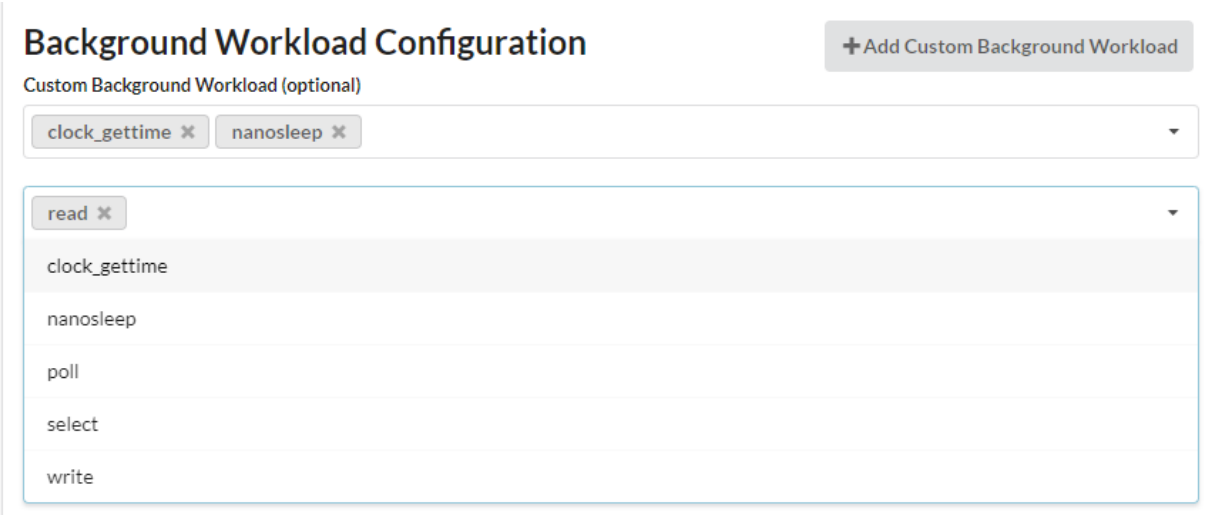

Figure 25: Background Workload Configuration

The subsequent component is the Core Configuration section. Here, users are required to pin the main workload to one specific CPU core and optionally pin custom or predefined background workloads to any number of CPU cores. Figure 26 demonstrates how users can pin main workloads and background workloads to cores. Additionally, users can select predefined background work to run on each core. nvPlayfair supports the following predefined background workloads: CPU_HOG, FORK_BOMB, and MEMORY_HOG.

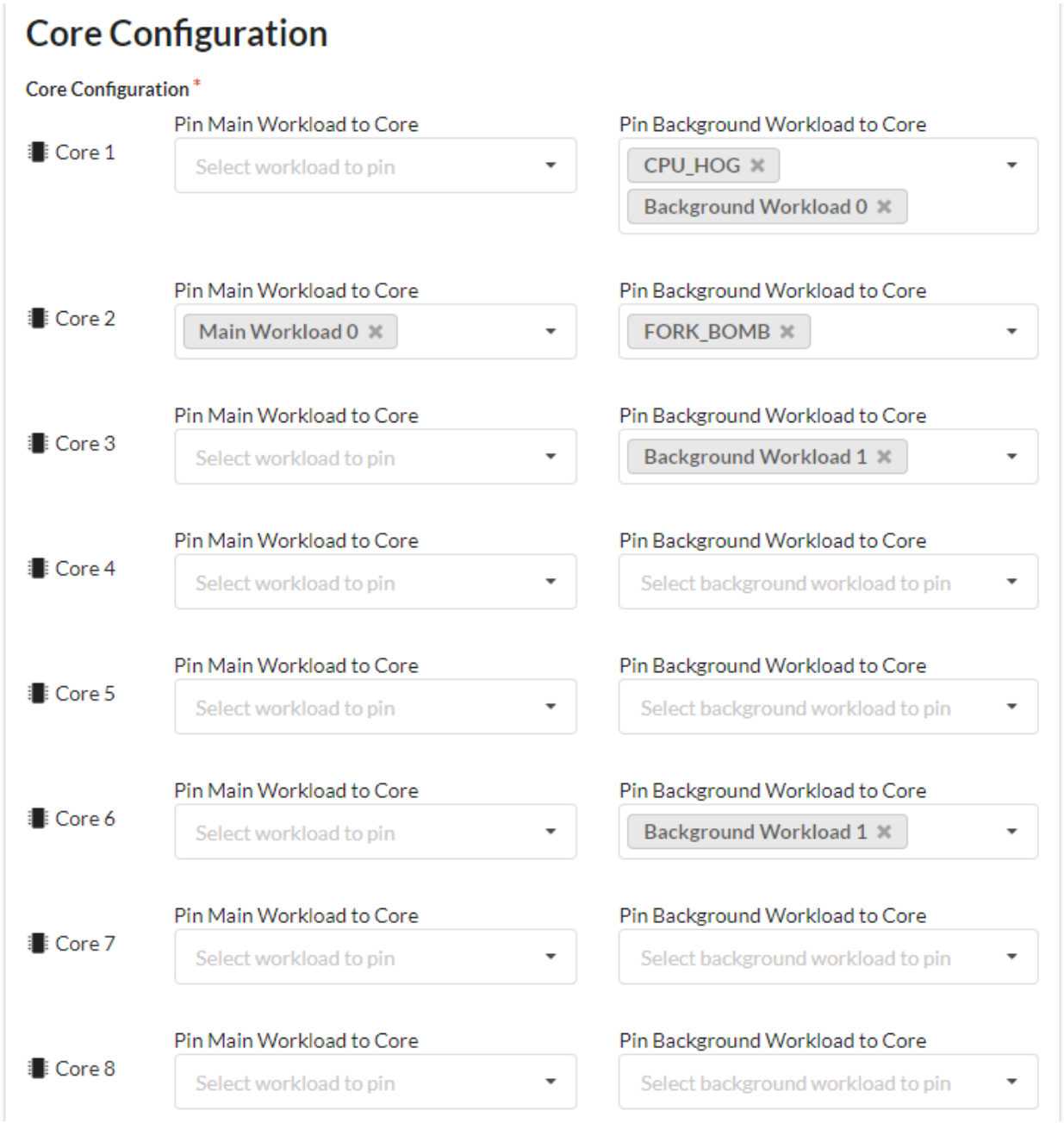

Figure 26: Core Configuration

The Scheduling Configuration section allows users to select a scheduling policy and a corresponding scheduling priority for each workload as shown in Figure 27. nvPlayfair supports three scheduling policies: SCHED_FIFO, SCHED_RR, and SCHED_OTHER. SCHED_FIFO

and SCHED_RR have priorities ranging from 1 to 99 inclusive, while SCHED_OTHER is set at 0.

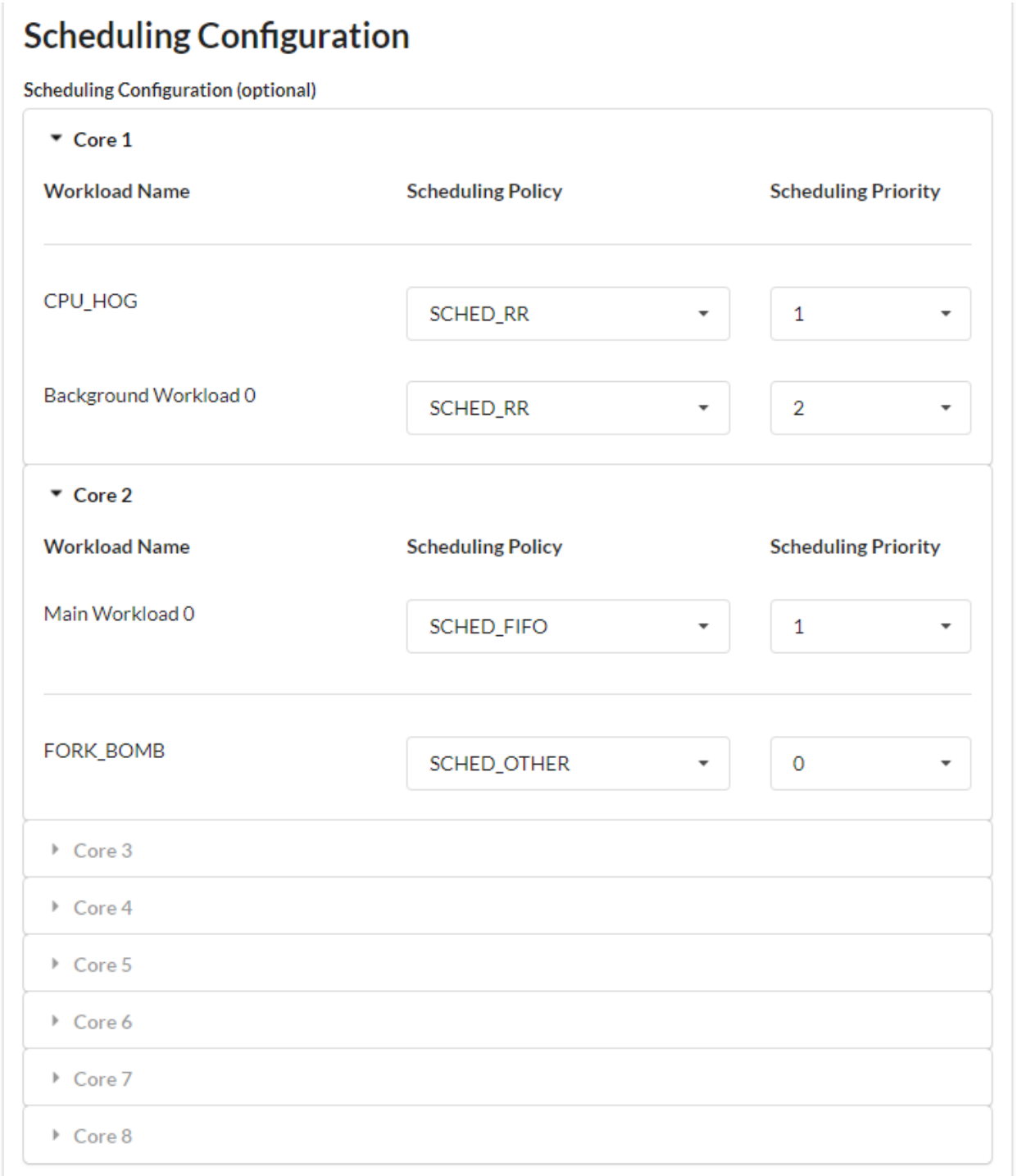

Figure 27: Scheduling Configuration

Lastly, in the Perf Configuration section, users can select perf counters to collect during benchmark test as depicted in Figure 28. All listed perf counters are from the Linux manual page from the perf_event_open API.

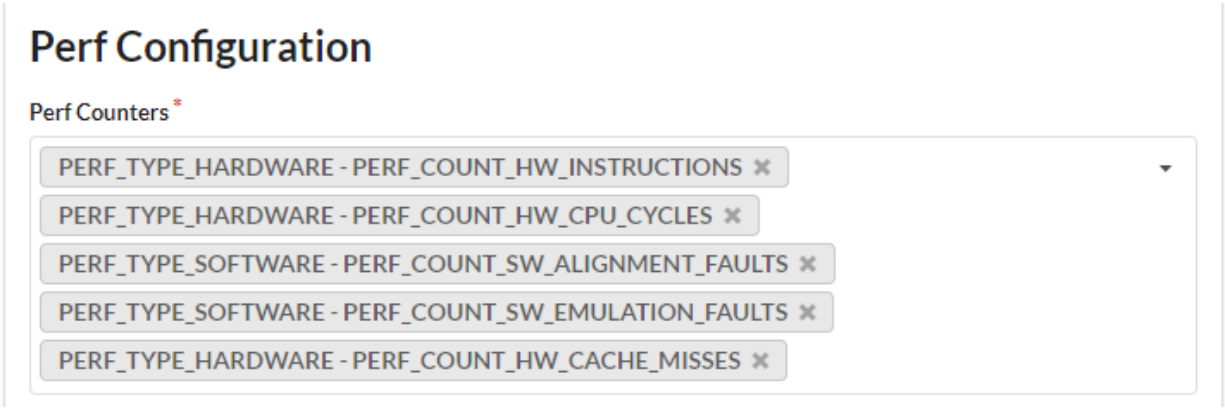

Figure 28: Perf Configuration

In total, nvPlayfair offers 17 perf counter options to record during a benchmark test run. Perf types are either hardware or software as denoted by “HW” or “SW”, respectively. nvPlayfair supports the following perf counter options:

- PERF_COUNT_HW_CPU_CYCLES
- PERF_COUNT_HW_INSTRUCTIONS
- PERF_COUNT_HW_CACHE_REFERENCES
- PERF_COUNT_HW_CACHE_MISSES
- PERF_COUNT_HW_STALLED_CYCLES_FRONTEND
- PERF_COUNT_HW_STALLED_CYCLES_BACKEND
- PERF_COUNT_SW_CPU_CLOCK
- PERF_COUNT_SW_TASK_CLOCK

- PERF_COUNT_SW_PAGE_FAULTS
- PERF_COUNT_SW_CONTEXT_SWITCHES
- PERF_COUNT_SW_CPU_MIGRATIONS
- PERF_COUNT_SW_PAGE_FAULTS_MIN
- PERF_COUNT_SW_PAGE_FAULTS_MAJ
- PERF_COUNT_SW_ALIGNMENT_FAULTS
- PERF_COUNT_SW_EMULATION_FAULTS
- PERF_COUNT_SW_DUMMY
- PERF_COUNT_SW_BPF_OUTPUT

Once a user submits the form to execute the benchmark test, our front-end makes a POST request to the back-end API. The request body data schema is shown below in Figure 29.

```
{
    "test_name": "my_benchmark_test",
    "iterations": 5000,
    "platform": "E3550 Native Linux",
    "mainWorkloadSequence": [
        {
            "id": "main_workload_0",
            "syscalls": ["read", "nanosleep", "write"]
        }
    ],
    "customBackgroundWorkloadSequence": [
        {
            "id": "bg_workload_0",
            "syscalls": ["nanosleep"]
        }
    ],
    "coreMappings": [
        {
            "id": 1,
            "mainWorkload": ["main_workload_0"],
            "backgroundWorkload": ["IO_HOG", "bg_workload_0"],
            "detailedMainWorkloadSettings": [
                {
                    "wl_id": "main_workload_0",
                    "sched_policy": "SCHED_RR",
                    "sched_priority": 1
                }
            ],
            "detailedBGWorkloadSettings": [
                {
                    "wl_id": "IO_HOG",
                    "sched_policy": "SCHED_OTHER",
                    "sched_priority": 0
                },
                {
                    "wl_id": "bg_workload_0",
                    "sched_policy": "SCHED_OTHER",
                    "sched_priority": 0
                }
            ]
        }
    ],
    "perfCounters": [
        {
            "type": "PERF_TYPE_HARDWARE",
            "perf_counter": "PERF_COUNT_HW_CPU_CYCLES"
        },
        {
            "type": "PERF_TYPE_SOFTWARE",
            "perf_counter": "PERF_COUNT_SW_TASK_CLOCK"
        }
    ]
}
```

Figure 29: Final Input Data Schema

Descriptions:

test_name = user-specified test name

iterations = number of iterations to run the workload for benchmarking

platform = platform to run the tests on. Fixed to "E3550 Native Linux". This platform has 8 cores.

boardIP = the IP address of the Tegra E3550 board flashed with Native Linux to run benchmark tests on.

mainWorkloadSequence = list of syscall / libc sequences

id = auto-generated sequence id

syscalls = list of the system call or libc function names. Current options are limited to those that we have implemented.

customBackgroundWorkloadSequence = list of syscall / libc sequences

id = auto-generated sequence id

syscalls = list of the system call or libc function names. Current options are limited to those that we have implemented.

coreMappings = list of core settings to pin workload to cores

id = auto-generated core id, auto-increment, start from 1

sequences = list of ids of defined sequences

backgroundWork = list of strings of predefined background work. Options are ["CPU_HOG", "MEMORY_HOG", "FORK_BOMB"]

detailedMainWorkloadSettings = list of scheduling config per each main workload per core. Since we only support one main workload thread for now, this list will only have one element.

wl_id: main workload id

sched_policy: one of [“SCHED_OTHER”, “SCHED_FIFO”, “SCHED_RR”]

sched_priority: ranges of values depending on the selected schedule policy

detailedBGWorkloadSettings: = list of scheduling config per each background workload per core

wl_id: main workload id

sched_policy: one of [“SCHED_OTHER”, “SCHED_FIFO”, “SCHED_RR”]

sched_priority: ranges of values depending on the selected schedule policy

perfCounters = list of perf counters config pairs

type = type of perf counters

perf_counter = the exact perf counter

3.2.5 Back-End Implementation

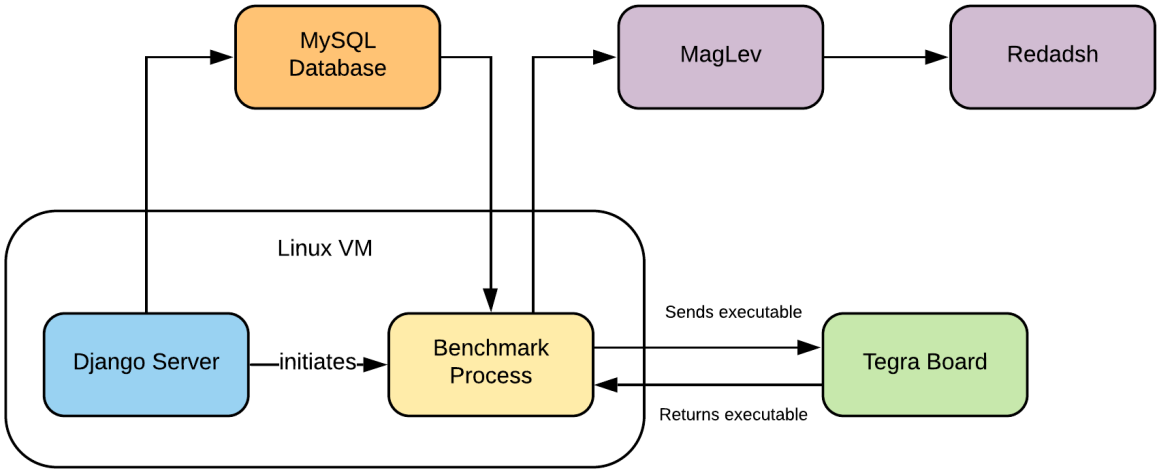

Figure 30: Back-End Workflow

Figure 30 depicts the process after users submit their test request. After the front-end sends an API request to the Django back-end, the request data is parsed appropriately and uploaded to the MySQL database. The Django back-end also initiates the process to benchmark on the Tegra board, starting with the script to generate the benchmark code. The benchmark process, which is written in Python, first retrieves the test input from the MySQL database. Next, the Python script dynamically generates a C file and cross-compiles it with the helper library files to get an executable to perform the benchmarking on the Tegra board. Subsequently, this executable gets copied over to the board environment and runs the benchmarking. The output for the benchmarking is written into a CSV file which then gets copied to the back-end server on the Linux VM. This output is processed and uploaded to MagLev.

3.3 Benchmarking On Tegra

nvPlayfair performs the benchmarking on a Tegra board by running an executable file in the Linux VM. The executable collects benchmark data and metadata about the Tegra board and transfers all data back to the Linux VM for further analysis.

3.3.1 Design Decisions

We originally proposed to use C++ as our programming language since it supports Google Benchmark which would help collecting system data. After gathering our project requirements and conducting more research on Google Benchmark, we found that Google Benchmark was not capable of collecting the data we wanted. It was also difficult to implement code that supported benchmarking several system calls. In our second sprint, we decided to switch to another approach, which was using Python to generate C code. The advantage of this approach was that we could generate benchmark code according to the user input in a flexible manner.

3.3.2 Data Collection

nvPlayfair collects two main pieces of data: the execution time of a main workload and perf counters of the system. We define execution time as the total run time of the main workload after the user-specified iterations; it is our metric to find and detect outliers of the benchmarked workload. Perf counters convey additional information about the system, such as hardware and software performance. In our project, we utilized the perf_event_open API call to collect metrics during the benchmarking phase. Perf counters help users understand what is happening under the hood in their system while looking at execution time outliers.

Metadata about the benchmarking system is also collected. nvPlayfair collects metadata, such as the total run time of the benchmark, the unix name of the system, and Linux standard base and distribution information.

Benchmark output and metadata is stored in two separate CSV files which are transferred back to the Linux VM for further processing to upload to MagLev.

3.3.3 Workload Generation

Our benchmarking code should be able to generate code that satisfies the user requirements, and at the same time open for extension for future use cases. Accordingly, we modularized our code structure and used a MySQL database to store all system call setup and cleanup code. It is separated into five sections which is shown in Figure 31:

1. One-time setup code
2. Individual run setup code
3. Code to be benchmarked
4. Individual run cleanup code
5. One-time cleanup code

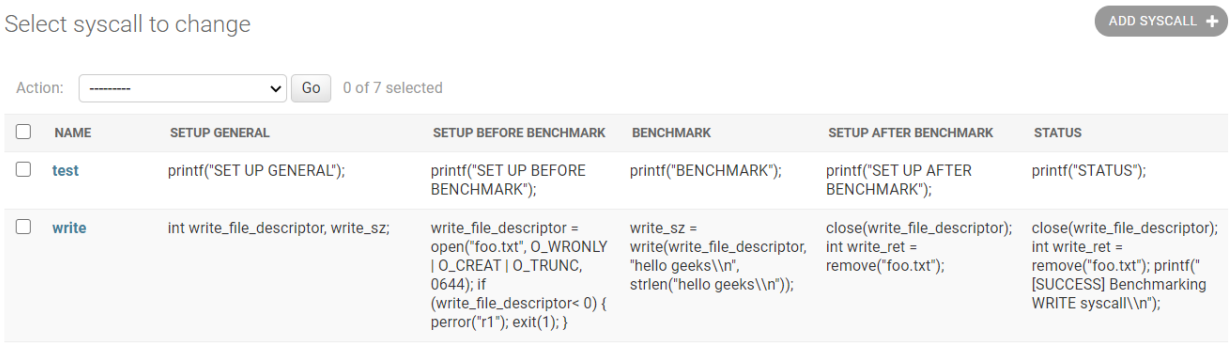

Figure 31: System Call Table

Breaking it down this way provided us with two major advantages. First, using this modular structure, we could virtually swap any system call, libc function, or higher-level function in and out of the benchmarking code easily. Second, we could combine multiple system calls, libc functions, and higher-level functions together for benchmarking by grouping together the portions of code for each of them respectively. Last but not least, this allows room for the application owner in the near future to add higher-level functions to benchmark easily.

The first column in the table is called “Setup General.” Certain system calls require one-time setup code. For instance, the read and write system calls require files to be generated before benchmarking. This section is outside of the iteration loop and is only run once. The second section is called “Setup Before Benchmark”. This section is in the iteration loop and runs before the benchmarked code snippet for every iteration. For example, when benchmarking the write system call, opening the file to write with a truncate flag is in this section. The third section is the actual code to be benchmarked. It will be wrapped around by perf counters and the timers to get the benchmark data. Next section is called “Setup After Benchmark”, which is responsible for closing files and cleaning up cache at the end of each iteration. And the last section called “Status” prints out the success message to the terminal and also removes all temporary files created by the program.

Since our program supports multiple system calls in one workload, it is important to make the variable in each system call unique. This is to prevent redeclaration errors when multiple system calls are added to the same workload.

One benefit of keeping benchmarking code in a database is that it is convenient to manage. In the future, the application owner can easily add or delete any system call without

touching any code in the back-end. It is also capable of adding custom code fragments. For example, any safety function that would like to be tested.

3.4 Data Visualization

nvPlayfair generates a dashboard that provides users with digestible visualizations to aid in understanding and analyzing their data. The dashboard is hosted on Redash with data tied to MagLev. Originally, MySQL and Grafana, a popular data visualization tool, were considered for use in the pipeline. However, our team chose to use Redash and MagLev because they are well-supported and used within NVIDIA. This allowed our team to reuse existing infrastructure to set up instead of using new resources.

3.4.1 Design Decisions

Our team initially planned to utilize MySQL as the framework's output data storage solution. MySQL is a common open-source relational database management system. Since there is a wide support for MySQL as well as its programming query clients, along with our experience with SQL, it seemed to be a good solution for our use case. However, there were two main disadvantages with MySQL. First, since MySQL does not have a strong memory-focused search engine, it can sometimes cause high overhead and cannot deliver optimal speed, especially when data is written in bulk. This raised an issue for our use case, since the output data can often get very large, given that the users can choose to benchmark thousands, or even hundreds of thousands of workload iterations. Second, since SQL is structured and requires a defined data schema, it can be an overhead for maintaining the framework. Our data schema can

change relatively quickly, and the time it takes to migrate the data could definitely be used on other higher priorities.

Later into the project, we discovered another internal data storage solution, MagLev. MagLev is widely used among NVIDIA's artificial intelligence and machine learning engineers as their big data storage solution, due to its ease-of-use, efficiency, and scalability.

Accordingly, we ran a comparison between these two solutions, shown in Table 2 below.

Feature	**MagLev**	**MySQL**
Python client	Yes	Yes
Data format (how does the data get stored?)	Structured Data, Tables, Partitions, etc. [15]	Tables / Relational
Volume of data	Flexible to scale	Set at 10-15 GB
IT Support	Yes, with existing infrastructure in place to reference	Yes, but need to do own research for reference to get support
Data store elasticity	Yes, MagLev "relies on a combination of object storage (Swift and S3), Presto, Redash, and a service called the Universal Data Catalog (UDC) to act as one such implementation of a data lake" [19].	Yes, with IT Ticket
Data manipulation	Can quickly use Python to read into Panda dataframe, process it, and write back to data lake. Data schema is flexible and can take in any format	Data writing in bulk causes overhead. Changing the data schema would require doing data migrations.

Table 2: Comparison Chart Between MySQL and MagLev

Ultimately, our team decided to use MagLev as the framework's output data storage solution. MagLev is flexible enough to allow the applications to scale in the future, but also is easy to set up and use. MagLev is also an internal tool that is extensively used within NVIDIA, which provides reliability and high level of support from IT.

As for our data visualization, initially our team planned to utilize Grafana. Grafana is an open-source analytics and interactive visualization web application, which can be configured to connect to most of the common datasources. We planned to install and deploy a Grafana server on an internal Linux VM. There were two major issues with Grafana, however. First, it required our team to spend time to set up and configure Grafana on the Linux VM. The setup and maintenance work would take away resources which could slow us down on developing new features. Second, there is a low level of support from IT to help scale this service in future use cases where we need to expand our framework to more users.

During the second sprint of the project, at the same time that we found MagLev as our data storage solution, we also found Redash as a potential data visualisation solution. Accordingly, we ran a comparison between these two solutions. Table 3 shows our findings:

Feature	**Redash**	**Grafana**
Efficiency of querying data	1 minute minimum automatic refresh frequency and 12 hours maximum query execution time [20]	Huge amount of writes to SQL can be an issue with performance and query time
Types of visualizations	No official support for histograms; support for scatter plots [21]	Support for histograms and scatter plot [22]
Axis configuration	Yes	Yes

Auto plot scaling	Yes, the plot can auto-scale and the user can manually zoom in	Yes, the plot can auto-scale and the user can manually zoom in
Level of support from IT	Yes	Limited. In case, where we need the capacity to allow more users to access the dashboard at once, we might need to increase the hardware requirement of the Linux VM. This can be done manually by creating a ticket.
Scalability → users	Yes, easily scale	No, required licenses to upgrade usage for more users
Scalability → storing the query & results of query	Unlimited saved queries [20]	Yes
Interactivity	Able to scroll / zoom / etc. but lacks granularity	Yes, able to scroll, zoom, etc.

Table 3: Comparison Chart Between Redash and Grafana

The decision to use Redash is tied to the usage of MagLev. Additionally, Redash also aligns more with our requirements and usage. More information can be found in the dashboard section.

3.4.2 Dashboard

The purpose of nvPlayfair's dashboard is to showcase the results of the benchmarking test and visualize potential outliers. This will help performance engineers better understand what is happening in the underlying system when outliers occur. Our dashboard contains seven widgets, including five core visualizations.

Our visualization is strictly tied to the SQL queries, also created and configured on Redash. Users can make custom SQL queries on Redash by retrieving data from the tables in MagLev. These queries can be used to create a visualization. Afterwards, on the dashboard view, users can select which queries and visualization to add to their custom dashboard. This provides users the flexibility to add and create graphs depending on their analysis. We also made use of parameterized queries to allow users to quickly change the bin size to bucketize in histogram, or to change the minimum and maximum range of the execution time graph. This breadth of customization ensures that users can always view basic visualizations to make their own analysis without the troubles to create new queries.

Starting from the top of the dashboard, the first widget is the “Test Name” widget, displayed in Figure 33. This test name auto-populates based on how the user named their test in the user interface of the control application. Additionally, we parameterized the test name to allow users to find their specified benchmark test results directly from the dashboard without making complicated queries on their own. The test name is responsible for loading the corresponding data into each visualization for the specific test.

Figure 32: Test Name

Underneath the Test Name widget is the Metadata widget, which describes the project version, the total run time of the test, and Linux system information provided by “uname” and “lsb”. Including the metadata widget helps the user recall which test they ran along with the system details. Below, Figure 34 is an example of the Metadata widget.

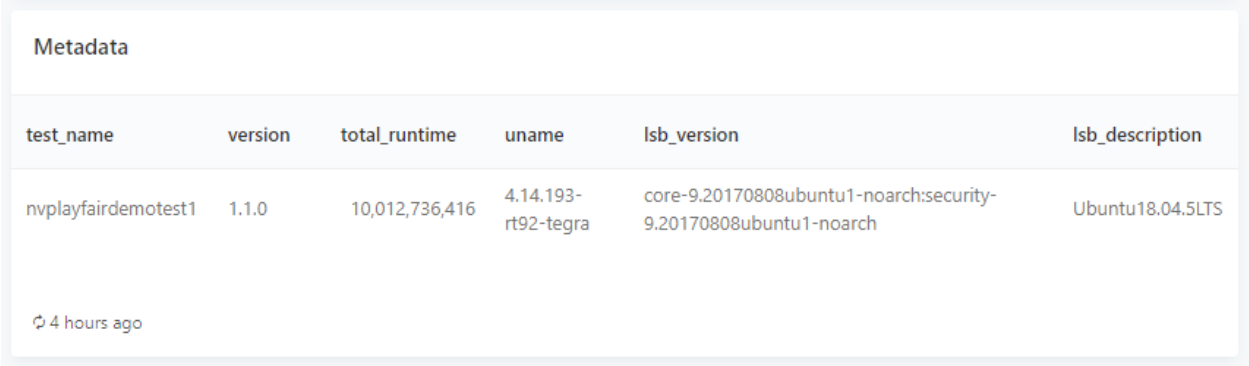

Figure 33: Metadata

Next, there are five core visualizations to help users understand the results of their benchmark test: (1) Frequency vs. Execution Time Histogram, (2) Execution Time Boxplot, (3) Outliers Table, (4) Time Series Scatter Plot, and (5) Execution Time Range Table. The Frequency vs. Execution Time Histogram in Figure 35 showcases data for the main workload. The chart is interactive and allows the user to zoom in and out on a data point, along with changing the bin size to get more or less granularity.

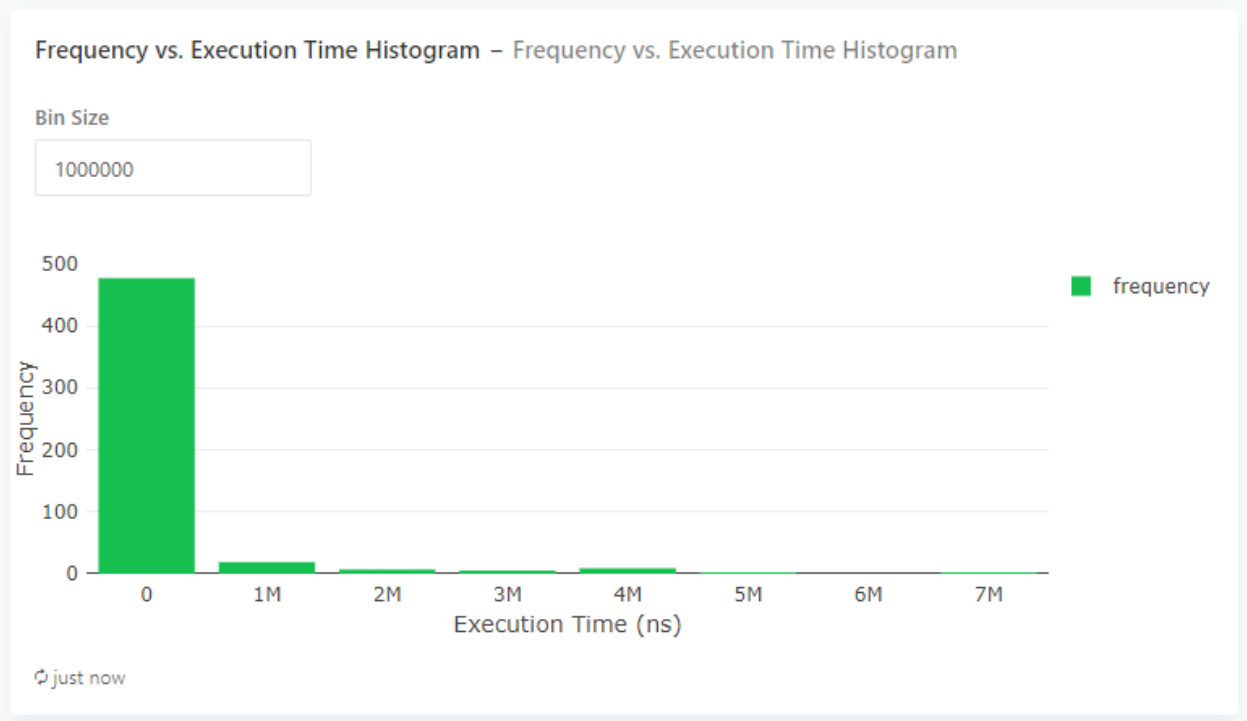

Figure 34: Frequency vs. Execution Time Histogram

In the second row of our dashboard, we display two visualizations that showcase data about the outliers. Figure 36 shows the first visualization, which is the Execution Time Boxplot. In this graph, users can see a box and whiskers plot, along with a five-number summary which includes the sample minimum, first quartile, median, third quartile, and sample maximum values.

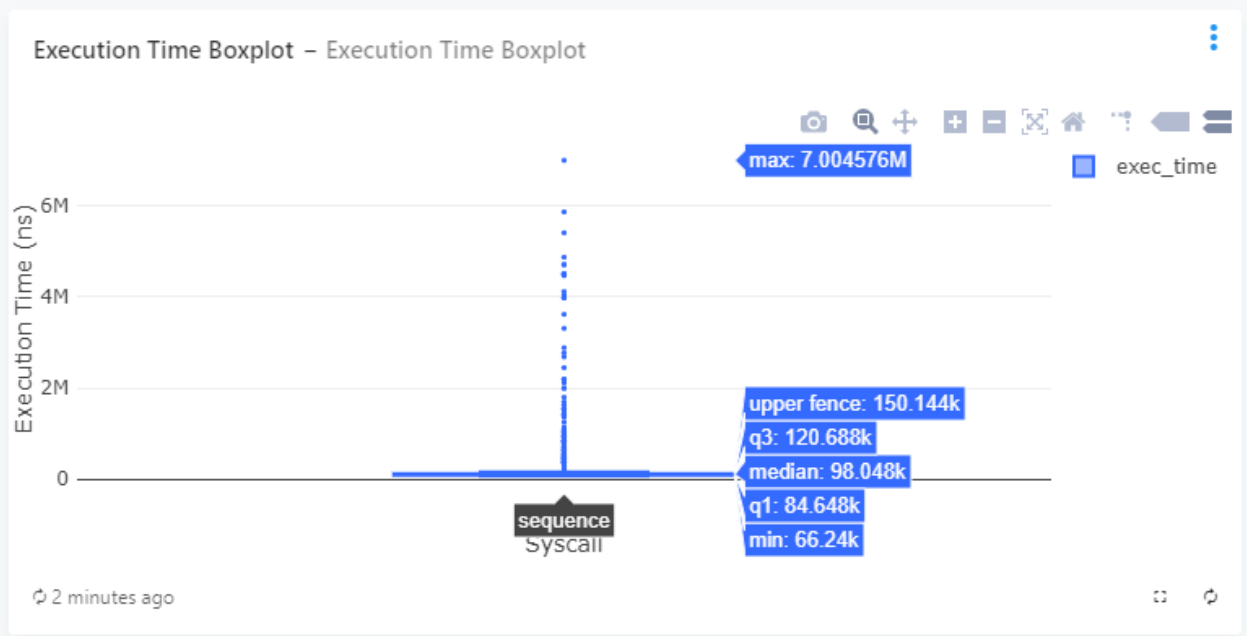

Figure 35: Execution Time Boxplot

The second visualization in this row is the Outliers Table. In the Outliers Table, users can view execution time outliers along with the corresponding user-specified perf counters. This table provides a more readable view for the user to look into what else is happening in the underlying system when outliers occur.

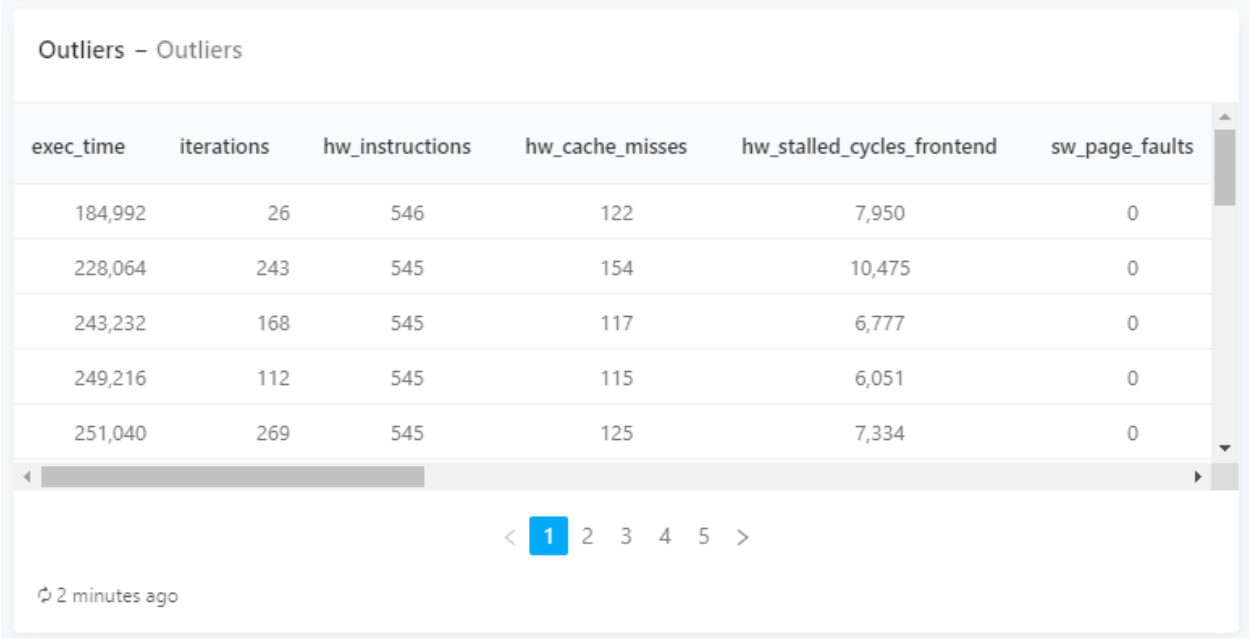

Figure 36: Outliers Table

The Time Series Graph shows the start time of the test versus the execution time as depicted in Figure 38. This visualization is intended to help users identify outliers that occur due to the system taking time to warm up.

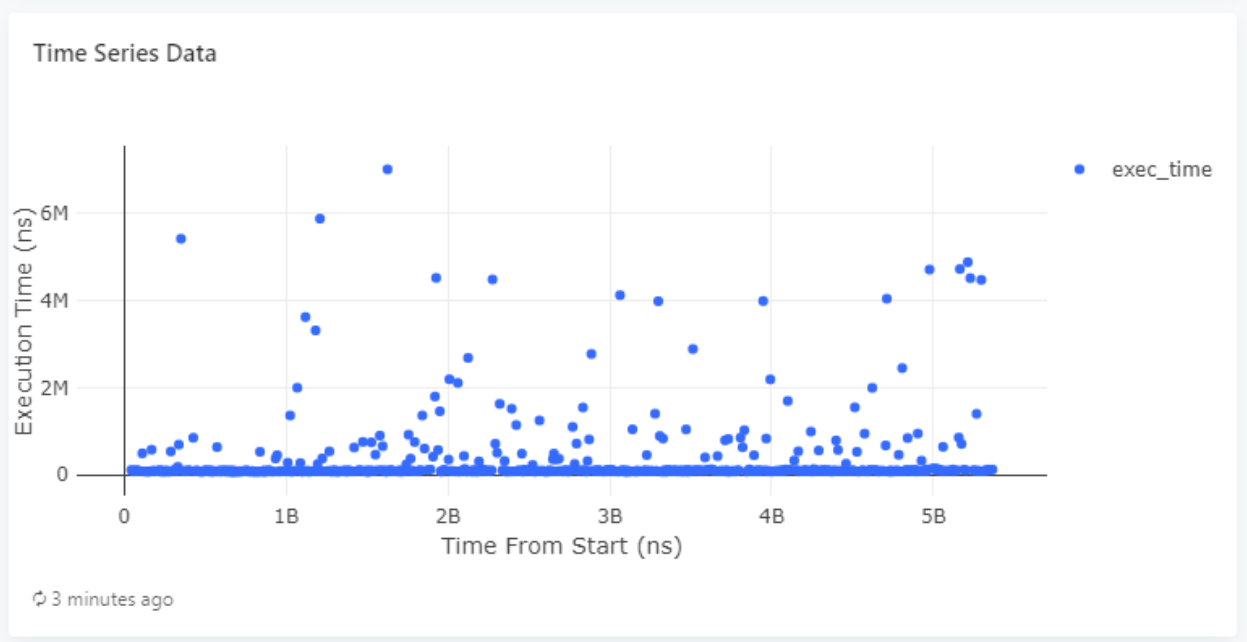

Figure 37: Time Series Graph

Our final visualization is the Execution Time Range Table shown in Figure 39. Users can utilize this table to search for benchmark data by specifying a minimum and maximum execution time to find the range for.

Execution Time Range – Execution Time Range

execution_time (ns)	iteration_start (ns)	iterations	perf_count_hw_cpu_cycles	perf_count_hw_instructions
5,412,960	350,870,816	28	43,417	553
5,872,256	1,209,504,960	114	40,991	553
7,004,576	1,626,755,584	156	40,817	553

2 minutes ago

Figure 38: Execution Time Range Table

4. Results and Analysis

To evaluate the result of nvPlayfair, we took two approaches: (1) record the number of features implemented versus features planned per sprint, and (2) collect user testing feedback. Reviewing the ticket completion rate per sprint provided us with a qualitative view of our team's progress. Examining user feedback provided us with a qualitative method to evaluate the functionality and ease of use of nvPlayfair.

4.1 Project Feature Requirements

Our project was divided into three sprints. For each sprint, we set a high-level goal along with a list of feature requirements that we aimed to implement by the end of the sprint. If our team was unable to implement a feature during the sprint, the task would be carried over to the subsequent sprint or moved to our backlog after re-evaluating the ticket. In this section, we revisit our requirements for each sprint and check off those that we were able to complete. This helps assess whether or not we met our sprint goal.

Below is the list of features for each sprint. Items preceded with a checkmark indicate that they were completed during the sprint, whereas items preceded with an empty check box signify that we were unable to implement them during the particular sprint.

4.1.1 Sprint 1 Requirements

Goal: Create a minimum viable product (MVP) to support benchmarking one system call at time.

- ✓ Write a shell script to automate flashing the Tegra board
- ✓ Define the test input data schema
- ✓ Define the test output data schema

- Benchmark - Calculate the run time
    - ✓ Research Google Benchmark
    - ✓ Research libc integration with Google Benchmark (decided to drop Google Benchmark since it did not fit our use case)
    - ✓ Write the benchmarking code using C++
- Benchmark - Perf counters
    - ✓ Research PAPI
    - ❑ Integrate perf counters into benchmarking code using PAPI
- Data source
    - ✓ Research NVIDIA's available database solutions
    - ✓ Setup MagLev
    - ✓ Write a script to populate the data source
- Data visualization
    - ✓ Research Grafana and pricing (dropped Grafana to use Redash)
    - ✓ Research Redash
    - ✓ Set up Redash
    - ✓ Create a frequency vs. execution time histogram for one system call
    - ✓ Create an outliers table for one system call
    - ✓ Create an execution time boxplot for one system call

4.1.2 Sprint 2 Requirements

Goal: Provide an end-to-end pipeline for a user to easily configure benchmark test specifications and visualize the results of the test.

- User Interface
    - ✓ Research user interface solutions
    - ✓ Create a UI mockup
    - ✓ Set up React front-end
    - ✓ Implement the following fields and functionality
        - ✓ Test Name
        - ✓ Number of Iterations
        - ✓ Platform Type
        - ✓ Add Main Workload
        - ✓ Add Background Workload
        - ✓ Pin Main Workload to Core
        - ✓ Pin Background Workload to Core
        - ✓ Perf Counters
        - ❑ Modal to display sequence labels
    - ✓ Show a loading screen and display success upon return 200; otherwise, display an error message
    - ✓ On success, display a dynamic link to Redash
    - ❑ Serve the front-end on a Linux VM
- API
    - ✓ Alter the API data schema to include test platform, sequences, core configurations, and selected perf counters
    - ✓ Set up a Django back-end

- ❑ Serve the back-end on a Linux VM

- Benchmarking
  - ✓ Evaluate benchmarking code options: C++ vs. Python-generated C
  - ✓ Refactor sprint 1 code and convert C++ to Python-generated C
  - ✓ Implement the following system calls and libc functions in addition to “read”
    - ✓ clock_gettime
    - ✓ nanosleep
    - ❑ poll
    - ✓ select
    - ✓ write
  - ✓ Implement workload generation
  - ✓ Write code to pin workload threads to user-specified CPU cores
- Data visualization
  - ✓ Parameterize the visualizations from Sprint 1 to display the correct test results to the users
- Automation
  - ✓ Research automating board reservations and flashing
  - ❑ Implement automating board reservations and flashing

✓ Reserve a Linux VM

4.1.3 Sprint 3 Requirements

Goal: Ensure that nvPlayfair is fully operational and maintainable.

- User Interface

- ✓ Add a Board IP field and validate the IP
- ✓ Label custom workload sequences
- ✓ Enable a user to pin multiple background workloads to a single core
- ✓ Add Scheduling Configuration section
- ✓ Add an NVIDIA favicon
- ✓ Make fields required
- ✓ Serve the front-end on a Linux VM
- ✓ Deploy the front-end to production and ensure it can continue running
- ✓ Write a README

- API
  - ✓ Ensure that the back-end can observe errors and return them to the front-end
  - ✓ Serve the back-end on a Linux VM
  - ❑ Deploy the back-end to production and ensure it can continue running
  - ✓ Write a README
- Benchmarking
  - ✓ Enable a user to pin multiple background workloads to a single core
  - ✓ Implement scheduling policies and priorities for threads
- Data visualization
  - ✓ Add a metadata widget to the dashboard
  - ✓ Add a time series graph to the dashboard
- Resolve bugs
  - ✓ Cross-Origin Resource Sharing (CORS)

✓ Change the name of the old Linux VM

✓ Write documentation on Confluence

4.2 User Testing

User testing was another method our team took to evaluate the success of our project. Our team asked two current NVIDIA engineers to test out our application from start to finish. One of them had never seen nor used our application before, while the other was familiar with the features our application provides as he worked closely with us. In this section, we explain our method for conducting the user testing and then analyze the feedback we received from our users.

4.2.1 User Testing Script

Below is the user testing script we followed to explain to our users what we expected them to do. We decided to follow a script so that we could standardize our results. We began by introducing our team and the framework we built. Next, we asked for their consent to use their responses and feedback in our report. After receiving consent, we began the user testing session.

- **Introduce our team and nvPlayfair.**
    - Hello [name], thank you for joining us today for this user testing session.
    - We are a group of interns working on a project with the Performance Architecture Team that focuses on developing nvPlayfair, a platform to measure and visualize system software performance on the Tegra chip.

- We would like to get some feedback from you on our user interface so that we can write about it in the results section of a report published by our university (Worcester Polytechnic Institute).

- **Ask for their consent to use their responses in our report.**

- **Warm-up questions:**
  - What is your current role at NVIDIA?
  - What are some daily tasks you have?

- **Request the user to verbally share their thought process.**
  - In order to get constructive feedback on our application, it is important for us to understand the thought process of the user. So, please share your honest thoughts as we go along. There are no right or wrong answers.
  - Please think out loud as you do this. Share with us where you are going to click, why you are clicking there, and what you expect to see after you do so. Feel free to ask any questions and express any confusion or concerns you have, even if they can seem straightforward. We only aim to get feedback to improve the platform, so your input is valuable.

- **Explain the test.**
    - This session will take no longer than thirty minutes to complete. We will guide you through a series of tasks to complete on nvPlayfair's user interface and dashboard.

- **Ask the user if they have any questions before starting.**

- **Describe a scenario:**
    - Imagine you have a program consisting of running some system calls, specifically "read" and "write". Sometimes your program takes a really long time to run. Another engineer suggests you using nv-playfair app to test the benchmark for your program. You would then go into the platform and create a test to benchmark the "read" and "write" system calls.

- **Tasks:**
    - Test Name
        - How would you start if you want to create a new test with the name "User Test Benchmark"?
    - Number of Iterations
        - What do you think the purpose of the Number of Iterations field is?
        - Please specify 300 iterations.
    - Board IP

        - What do you think the purpose of the "board IP" field is?
        - What would you fill in for this field?
    - Platform Type
        - Please select Native Linux as the platform type.
    - Main Workload Configuration
        - What would you do if you wanted to benchmark the execution time of a program consisting of "read" and "write" system calls?
    - Background Workload Configuration
        - If you wanted to select a more advanced configuration in addition to just benchmarking the "read" and "write" system calls, where would you navigate to?
    - Core Configuration
        - What do you think the purpose of the Core Configuration section is?
        - Please pin the main workload to core 1 and pin the custom background workload to core 2. Next, select CPU_HOG and MEMORY_HOG and pin it to core 7.
    - Scheduling Configuration
        - What do you think the purpose of the Scheduling Configuration section is?
        - Please add a SCHED_FIFO scheduling policy to the main workload with priority 2. Next, add a SCHED_RR scheduling policy to your custom background workload with priority 98. Finally, add the SCHED_OTHER

scheduling policy to your remaining background workloads with priority 0.

- Perf Configuration
  - What do you think the purpose of the Perf Configuration section is?
  - Please select PERF_COUNT_HW_INSTRUCTIONS and PERF_COUNT_SW_CACHE_MISSES.
- Execute Benchmark
  - Now you are finished with configuring the test input. Can you submit the test?
  - What do you expect to show up after submitting the test?
- Redash
  - Open the link.
  - Looking at this dashboard, which visualizations could you see being helpful for your role?

- **Ask for feedback.**
  - Now that you have finished the tasks. What do you think about this process you just went through?
  - User Interface
    - Do you have suggestions for any improvement on the UI?
    - Is there anything that confused you when you were asked to complete the tasks?
    - Are there any functionalities you wish to see on this platform?

        - Rate 1-5 for the user interface [1= hard to use 2= slightly difficult 3=normal 4=slightly easy 5= super easy]
        - Why did you choose that answer?
    - Redash Dashboard
        - Do you have suggestions for any improvement on the dashboard?
        - Rate 1-5 for usability [1= hard to use 2= slightly difficult 3=normal 4=slightly easy 5= super easy]
        - Why did you choose that answer?
    - How helpful would nvPlayfair be given your daily tasks?

- **Thank the user for their time and feedback.**

4.2.2 User Feedback: Rhyland Klein

The first user testing session we conducted was with Rhyland Klein. Klein is part of the Drive Behavior team in Automotive within NVIDIA. Klein told us his daily tasks include test automation and running. Klein was unfamiliar with nvPlayfair and its functionality.

When asked to enter a benchmark test name on the user interface, Klein entered "User Test Benchmark." It is important to note that this is an invalid test name because there are spaces in between each word, and the test name corresponds to a table in MagLev. In MagLev, table names are not allowed to have spaces between words. This indicated to us that the test name field on the user interface needs to define what a valid test name is to the user. Klein was able to configure the rest of the benchmark test with accuracy and ease.

Next, we asked Klein to navigate to the Redash dashboard. Once on the dashboard, we asked him to examine all of the visualizations and explain whether he thinks they provide the user with relevant information regarding their test run, specifically if the user wants to identify why their program took a long time. Klein noted that the frequency vs. execution time histogram, outliers table, and time series graph looked straightforward and might be particularly useful. He explained that the execution time boxplot looked confusing to him.

After finishing the tasks, we asked Rhyland for overall feedback on nvPlayfair. Klein rated both the usability of our platform and the functionality of the user interface a four out of five. He commented that nvPlayfair's user interface was straightforward and had a logical progression. He liked our use of dropdowns for common items in the Main Workload Configuration, Background Workload Configuration, Core Configuration, and Scheduling Configuration sections.

Klein provided our team with a few suggestions regarding the user interface. First, Klein mentioned that the loading spinner that appears on form submission was confusing, especially if it takes a long time. He said that users might think their browser is frozen with no indication of when the benchmark test will finish. Another suggestion Klein had was to give the user the ability to configure a more custom and complex workload. He noticed the user must currently pick from a set list of system calls, which does not provide much flexibility. Additionally, he recommended implementing pre-set configurations for the background workloads. This way, the user does not have to click every single field. Klein's final piece of feedback was to incorporate more colors into the user interface design to visually break up our sections.

When asked if this platform would be helpful with his daily tasks, Klein said no. However, he mentioned that our tool could be helpful for other teams he works with who currently run RoadRunner on Tegra to stream and display data over different sockets.

4.2.3 User Feedback: Waqar Ali

Waqar Ali is a System Software Engineer on the PAT at NVIDIA. Although Ali took part in our development process, he was able to provide useful recommendations. Below are his recommendations and feedback on the UI:

- The user should not have to go through the entire form to know if the test name is valid.
- The platform type is confusing because the given options are a combination of two things. For instance, the option “E3550 Native Linux” includes an E3550 platform and a Native Linux operating system.
- It is a good design choice for the scheduling configuration to be optional. If a user does not want to specify the scheduling policy and priority for each thread, it will default to SCHED_OTHER.
- Creating a new workload is easy. Ali stated that selecting the name of a system call is easier to do as opposed to inserting the code for each system call.
- Ali also suggested that perf counters should not be a required field because there might be users who only want to record execution times without a need to collect perf counters
- Additionally, he pointed out that when putting in workload specification, users should be able to specify the workload name.

Next, we asked Ali to open the Redash page to examine all the test result visualizations. Below are Ali's recommendations and feedback for each visualization in the dashboard:

- Metadata
    - Helpful to know information about the system that ran the test, such as the exact release version and kernel version
- Frequency vs. Execution Time Histogram
    - Very helpful for a performance engineer to view a distribution of execution times
    - Configurable bin sizes is a nice feature to see the granularity of data
    - Visually appealing but suggested us to incorporate more colors
- Execution Time Boxplot
    - Five number summary is convenient
    - Helpful for detecting outliers
    - Thinks the boxplot is more useful than the histogram visualization
- Outliers Table
    - While the table is helpful for viewing an outlier's corresponding perf counters, Ali thinks that this information would be better inside of a visualization
- Time Series Graph
    - Could be more visually appealing by adding colors
    - Would be more useful if there was a way to overlay this time series graph with a time series graph from another run to help compare
- Execution Time Range Table
    - Likes the ability to zoom in on certain data points

- Helpful to look at performance stats

In terms of feedback, Ali gave the usability of nvPlayfair a rating of three out of five. He provided two reasons for such a rating. First, he noted that engineers who are not familiar with web development can be confused by the error received on the front-end, since our framework does not display a detailed error message upon encountering one. Second, he mentioned that our framework could be unintuitive for users who do not work daily with system software performance. However, he rated nvPlayfair's user interface a five out of five because the field names are straightforward and intuitive.

4.3 Summary

To assess the merits of nvPlayfair, our team reviewed the completed project feature requirements and gathered feedback from user testing sessions. We consistently achieved our sprint goals by implementing all of the basic requirements and the majority of our stretch goals into nvPlayfair. Furthermore, based on the feedback we received from two test users, we believe that nvPlayfair is a viable product that can help NVIDIA engineers benchmark user-specified workload, identify execution time outliers, and analyze their causes.

5. Conclusion

In order for the PAT to satisfy safety requirements with deadlines, the PAT needs to understand the performance and determinism of the software involved. Additionally, in the case that a safety function does not execute within a given deadline, the PAT needs to detect the execution time outliers and analyze the reason behind it. Accordingly, the goal of our nvPlayfair project was to develop an automated benchmark test pipeline for NVIDIA engineers to benchmark system software performance on the Tegra SoC, with the focus on Linux system calls and basic libc functions as a proof of concept. In order to achieve that goal, we implemented three key components: (1) a user interface to configure benchmark tests, (2) a back-end service to automate workload generation and benchmark test deployment, and (3) a dashboard to help detect execution time outliers and analyze its causes.

During the course of the project, we met with our mentors and manager in order to better understand the end goal as well as requirements for this project. Given the number of feature requirements of the project, we analyzed and prioritized them, then created high-level goals during our sprint planning meetings, with help and guidance from our mentors. Furthermore, we added some investigation tickets along the way to help determine the best tool for a specific task as well as lower the maintenance effort. In addition, we re-evaluated our goals and tickets every week, which helped keep the project on track.

As a result, nvPlayfair provides a framework that can help the PAT better understand the safety function system software performances, by assisting them in detecting execution time outliers and analyzing their causes.

5.1 Future Work

This section discusses the limitations of our framework and potential solutions, additional features to add, and automating board reservations and flashing.

5.1.1 Limitations and Potential Solutions

First, nvPlayfair currently only allows the user to configure one main workload to benchmark and collect performance counter data. This was due to a concern that the perf_event_open API, which was used to collect perf counters, is single-threaded. Therefore, collecting perf counters this way across multiple threads would yield inaccurate results.

Second, nvPlayfair only supports running the benchmark tests on the E3550 Tegra board that had been flashed with Embedded Native Linux. This is because the embedded Linux, also known as Hypervisor + Linux, did not have the ARM Performance Monitoring Units (PMUs) enabled. Accordingly, Richard, an engineer from the PAT, helped enable the ARM PMUs in Embedded Native Linux, so that we could collect the hardware perf counters. In the future, when ARM PMUs is enabled across other systems (such as HV+L or QNX), our framework will be able to run the tests on them.

Third, our framework does not support the ability to run tests on QNX platforms yet. However, due to the extendable setup of the framework, we could potentially add the QNX cross compiler to our system, and conditionally select the correct cross compiler to generate the compatible executable to be run on QNX.

Fourth, nvPlayfair only supports a handful of Linux system calls. However, we were able to design our system in a way that can extend to higher-level workloads. Specifically, application owners can configure higher-level functions in the existing MySQL database.

Fifth, the time between submitting a benchmark test and receiving the link to Redash is long. As part of future work, we recommend that benchmark tests be run in the background, with a feature to send an email to the user once their test completes.

5.1.2 Additional Features

Below is a list of new features requested by other NVIDIA engineers that have not been implemented due to time limitations:

1. Provide an option for users to specify a higher-level workload.
2. Support comparing graphs between different test runs.
3. Add support for generating formal reports.
4. Enable replaying specific system calls / libc functions patterns. This can be based on the output of ftrace on a specific application (e.g. RoadRunner).
5. Enable users to specify the weight of each workload.

5.1.3 Automate Board Reservation and Flashing with Colossus

The benchmarking pipeline currently requires users to manually flash the board and pass their reserved board's IP address downstream to our pipeline, so the back-end can bring the benchmarking tests over to their board to run. Additionally, users are required to flash the board in a specific way in order for the perf counters collection to work correctly. Specifically, given that users perform a normal embedded Linux flashing process for the E3550, the benchmarking tests will not be able to collect the hardware perf counters, since the normal flashed embedded Linux - which is Hypervisor + Linux - does not enable the ARM performance monitoring units by default. Therefore, users would need to flash a Native Linux build to the E3550. Users will

not have to go through such a process if our framework could help them automate the board reservation and flash a custom Native Embedded Linux build.

One potential solution for automating board reservation and flashing is Colossus, an internal tool to reserve a wide variety of hardwares with automation support. NVIDIA system engineers utilize this service to automatically reserve a hardware resource, along with running environment setup scripts and customized workload on the reserved resource using Ansible. At the time of our research, the Colossus team was in the middle of enabling support for Tegra. Unfortunately, our use case required custom automatic flashing support, direct access to the reserved Tegra board, and internet connectivity on the board at that time. Additionally, there were a limited number of the E3550 boards available. Therefore, although automating the board reservation and flashing was a nice feature, we decided not to include it into our framework due to time limitations and unsupported features of Colossus.

6. References

[1] "NVIDIA Tegra: Next Generation Mobile Development". 2021. Tegra. [online] Available at: https://developer.nvidia.com/tegra-development. [Accessed 20- Mar- 2021].

[2] En.wikipedia.org. 2021. Tegra. [online] Available at: https://en.wikipedia.org/wiki/Tegra. [Accessed 20- Mar- 2021].

[3] R. Joshua Ho, "NVIDIA Tegra X1 Preview & Architecture Analysis", Anandtech.com, 2021. [Online]. Available: https://www.anandtech.com/show/8811/nvidia-tegra-x1-preview. [Accessed: 20- Mar- 2021].

[4] DRIVE Software, 2021. [Online]. Available: https://developer.nvidia.com/drive/drive-software. [Accessed: 20- Mar- 2021].

[5] RoadRunner, 2021. [Online]. Available: https://confluence.nvidia.com/display/AVQA/RoadRunner. [Accessed: 20- Mar- 2021].

[6] DRIVE Perception, 2021. [Online]. Available: https://developer.nvidia.com/drive/drive-perception. [Accessed: 20- Mar- 2021].

[7] DRIVE Mapping, 2021. [Online]. Available: https://developer.nvidia.com/drive/drive-mapping. [Accessed: 20- Mar- 2021].

[8] DRIVE Planning, 2021. [Online]. Available: https://developer.nvidia.com/drive/drive-planning. [Accessed: 20- Mar- 2021].

[9] S. Sood, "RoadRunner QNX Event Mining," presented to PAT, 2021. [Google Slides].

[10] "React – A JavaScript library for building user interfaces", Reactjs.org, 2021. [Online]. Available: https://reactjs.org/. [Accessed: 20- Mar- 2021].

[11] "Components and Props – React", Reactjs.org, 2021. [Online]. Available: https://reactjs.org/docs/components-and-props.html. [Accessed: 20- Mar- 2021].

[12] Code Studio, Understanding virtual DOM React JS. 2020.

[13] "Top 7 Backend Frameworks for Web Development in 2020", Kelltontech.com, 2021. [Online]. Available: https://www.kelltontech.com/kellton-tech-blog/7-most-popular-backend-web-development-frameworks-2020. [Accessed: 20- Mar- 2021].

[14] "Why We Use Django Framework & What Is Django Best Used For", Django Stars Blog, 2021. [Online]. Available: https://djangostars.com/blog/why-we-use-django-framework. [Accessed: 20- Mar- 2021].

[15] MagLev User Guide, 2021. [Online]. Available: https://maglev.nvda.ai/docs/platform/index.html. [Accessed: 20- Mar- 2021].

[16] "Presto | Distributed SQL Query Engine for Big Data", Prestodb.io, 2021. [Online]. Available: https://prestodb.io. [Accessed: 20- Mar- 2021].

[17] "Spark SQL and DataFrames - Spark 3.1.1 Documentation", Spark.apache.org, 2021. [Online]. Available: https://spark.apache.org/docs/latest/sql-programming-guide.html. [Accessed: 20- Mar- 2021].

[18] "Agile as a Software Development Process - Software Engineering Authority", Software Engineering Authority, 2021. [Online]. Available: https://ao.gl/agile-as-a-software-development-process. [Accessed: 20- Mar- 2021].

[19] MagLev Data Lake, 2021. [Online]. Available: https://maglev.nvda.ai/docs/workflows/2.0/recipes/data-lake.html?highlight=scalability. [Accessed: 20- Mar- 2021].

[20] Redash.io, 2021. [Online]. Available: https://redash.io/pricing. [Accessed: 20- Mar- 2021].

[21] Redash.io, 2021. [Online]. Available: https://redash.io/help/user-guide/visualizations/visualization-types. [Accessed: 20- Mar- 2021].

[22] "Visualizations Grafana Labs", 2021. [Online]. Available: https://grafana.com/docs/grafana/latest/panels/visualizations. [Accessed: 20- Mar- 2021].

Appendix A

Our manager, Mitch Luban, proposed the name nvPlayfair for our application. The name was inspired by William Playfair, who introduced the idea of graphical representation into statistics. Additionally a “play fair” is a double entendre for a “performance”, and our framework measures software performance.