Software Design

January 29, 2024

Team

Kowalski

Sponser

Igor F Steinmacher

Team Mentor

Saisri Muttineni

Team Members

Erick Salazar, Bailey McCauslin, Jake Borneman, Nick Wiltshire

Revision

Version 1

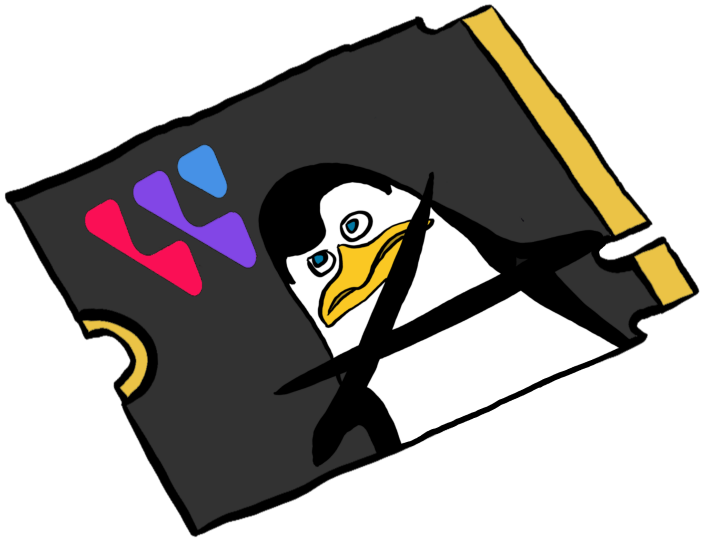


TABLE OF CONTENTS

Introduction

Project Overview

Currently many companies test SSDs for different uses, like desktops, appliances, and cloud computing to make sure the product quality meets the customers' product specification requirements. However, the industry currently lacks the following capabilities, significantly slowing down the research and development process:

- No data collection system, gathering and storing performance data from kernel-level.
- No centralized system for gathering collected data, analyzing it for thresholds, and visualizing results.
- Due to the lack of a centralized system, there is no way to automate the entire process, from data collection to displaying results.

Our framework would enable the collection, storage, analysis, and provision of visibility and monitoring through visual dashboards. The absence of such a framework hinders the automation of proactive failure detection in real-time and near-real-time scenarios. This delay in issue detection not only incurs time and financial costs for the companies but also adversely affects customer satisfaction metrics.

By implementing our observability framework, validation teams at these companies will be proactively able to detect issues in the SSD through the eye of the Kernel stack behaviors. On top of that, the data they collect will be presented in an organized, efficient, and effective manner. They will be able to see their data graphically represented in a comprehensive dashboard, equipped to alert when certain performance thresholds are not met. This will have a great impact on the company's ability to improve their products, and gain a competitive edge over their competitors.

Key User-Level Requirements:

- Efficient observability framework for capturing insights into system kernel behaviors across various stack levels.
- Collection, storage, analysis, and provision of visibility and monitoring through visual dashboards.

- Proactive detection of SSD issues in real-time and near-real-time scenarios.
- Organized, efficient, and effective presentation of collected data through comprehensive dashboards.
- Ability to alert validation teams when performance thresholds are not met.

Functional/Performance Requirements:

- Ability to capture system kernel behaviors at various stack levels.
- Efficient data collection, storage, and analysis.
- Real-time and near-real-time proactive detection of SSD issues.
- User-friendly and comprehensive visualization through visual dashboards.
- Reliable alerting mechanism for performance threshold deviations.

Environmental Constraints:

- Compatibility with existing testing environments and infrastructure.
- Scalability to accommodate varying testing scenarios and workloads.
- Minimal impact on system performance and resource utilization.

Implementation Overview

Solution Vision Recap

To tackle this issue we will be building a platform that condenses and automates a company's existing data analytics workflow. With our platform in place, validation teams will be able to easily view important performance data and alerts, without having to individually collect, store, or query any data manually. Data will also be organized and stored in a way that facilitates temporal analysis, allowing validation teams to identify issues that are hidden in real time.

Overall Approach (Technologies/Tools)

Technologies/Tools

- MySQL database: used to store kernel-level data collected from drives
- MongoDB (noSQL): used to store master data of tested drives (e.g. name, serial number, capacity, etc.)

- Python: used to script APIs and connect all system portions into a single file
    - Pymongo: MongoDB API for scripting in Python
    - Boto3: API for AWS service to script OpenSearch
    - bcc: Module to run eBPF scripts
    - Tkinter: Module used to create GUI to run program
- Open Search: used to create visual dashboard to graph collected performance data
- eBPF: used to probe drives at kernel-level and produce performance statistics

The technologies will be used to create a Friendly GUI where a developer can enter several inputs resembling Linux command line arguments. Once all inputs are set, the program will start running, collecting driver performance data and its master data. These will then be uploaded to their respective sql databases and the program will link the respective driver performance data before it gets displayed into a visual dashboard for analysis.

Architectural Overview

Architectural Diagram

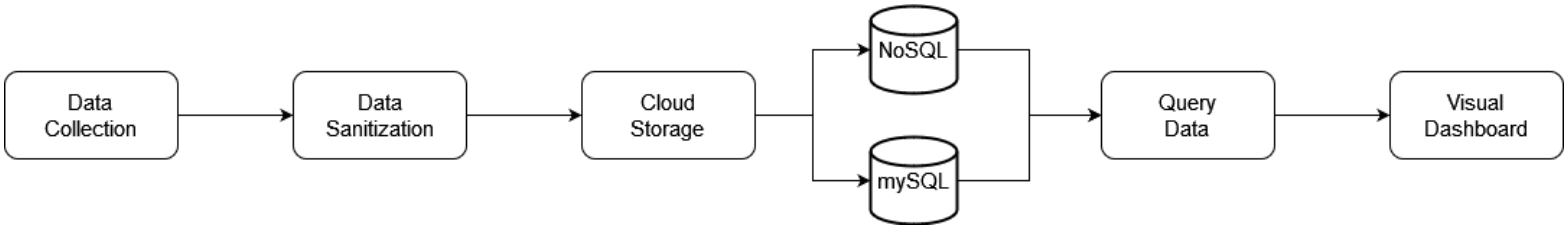


Fig. 01 High level overview of project architecture.

Discussion of Architecture

Key Responsibilities and Features of Components

Data Collection

- Driver performance data (Transactional Data) and device information (Master Data) are collected in this stage of the architecture.
- Transactional Data is collected using eBPF scripts run in Python, which allows access to kernel-level operations in Linux

- eBPF provides tools to stress test specific block devices, and the architecture focuses on the use of tools such as bioerror, biopattern, biolatency, and nvmelatency
- All transactional data is collected and stored into a CSV file for upload into a MySQL database
- Master Data is collected using Linux command line tools "lsblk" to list all block devices connected to the computer and "smartctl" to display information of connected devices such as capacity, serial number, and model
- The output of "smartctl" is parsed, saved into a text file, and then turned into a JSON file for upload to a noSQL database

Cloud Storage

- All data from the data collection stage is uploaded into their respective databases through Python scripts for storage and analysis in the next stage.
- Master Data is uploaded into a noSQL database, while Transactional Data is uploaded into a mySQL database
- The Python scripts handle the connection between both datasets through keys to ensure the data in each database corresponds with each other

Query Data

- Data is queried from databases based on specific thresholds.
- The queried data is saved and analyzed on a visual dashboard.
- A local crawler differentiates between master data and transactional data.
- The separated data is relocated for use in the visual display.
- Users can integrate their own databases with the querying and storage process.

Visual Dashboard

- Data collected, ranging from 1 dataset to an infinite amount, is compiled using open source for analytics display.
- The display allows users to view the most recent collected data and the overall collected data.
- Users can set the duration for which the data will exist before manual deletion or setting a benchmark for deletion.

Communication Mechanisms and Information/Control Flows

Data Collection

- **Communication Mechanisms**: eBPF scripts and Linux command-line tools (lsblk, smartctl) are used for collecting data. eBPF scripts enable kernel-level data collection, while command-line tools gather device information.
- **Information Flow**: Transactional data collected via eBPF scripts and Master Data via command-line tools flow into CSV and JSON files, respectively, before being uploaded to databases.

Cloud Storage

- **Communication Mechanisms**: Python scripts facilitate the upload of data to MySQL (for Transactional Data) and noSQL databases (for Master Data). These scripts manage database connections and data integrity.
- **Information Flow**: Data from the collection phase is transferred to respective databases for storage, ensuring that Transactional and Master Data are correctly associated through keys.

Query Data

- **Communication Mechanisms**: Database queries based on specific thresholds and a local crawler differentiate and relocate data for analysis.
- **Information Flow**: Queried data is processed for visualization, allowing integration with external databases and ensuring data is ready for display on the dashboard.

Visual Dashboard

- **Communication Mechanisms**: Open-source analytics tools compile and display data from various datasets.
- **Information Flow**: Users interact with the dashboard to view, analyze, and manage data, including setting data retention policies.

Architectural Styles Embodied

Data Collection

- **Event-Driven Architecture**: The use of eBPF scripts for real-time data collection from kernel operations embodies an event-driven approach, reacting to system events.
- **Microservices Architecture**: Utilizing specific tools for different types of data collection (eBPF, lsblk, smartctl) reflects a microservices approach, with each tool serving a unique purpose.

Cloud Storage

- **Service-Oriented Architecture (SOA)**: Python scripts acting as services to upload data to MySQL and noSQL databases embody SOA, with services designed to handle specific data storage tasks.
- **Cloud-Native Architecture**: Storing data in cloud-based databases (MySQL and noSQL) reflects a cloud-native approach, leveraging cloud technologies for scalability and flexibility.

Query Data

- **Data-Oriented Architecture**: The focus on querying data based on thresholds and preparing it for analysis and visualization reflects a data-oriented architecture, prioritizing data accessibility and usability.
- **Microservices Architecture**: The use of a local crawler to differentiate and manage data types for visualization purposes can be seen as a microservices approach, with each service handling a specific data processing task.

Visual Dashboard

- **Component-Based Architecture**: The dashboard's use of open-source analytics tools to compile and display data from various sources embodies a component-based approach, integrating different components for a cohesive user experience.
- **Service-Oriented Architecture (SOA)**: Providing users with the ability to interact with and manage data through the dashboard reflects SOA principles, with the dashboard serving as a user-facing service layer.

Module and Interface Descriptions

Module 1: Data Collection

When starting up the program, we would request a JSON file containing the different types of arguments the program would use. This JSON file would contain the arguments “Process Time” and “Operation”. There will be expansion when going further in the project. Once this is inserted the python file would call the data collection files to activate. These data collection files are called BT files and they are from the program eBPF. Upon activating these files, the process of kernel information collection would happen and keep running for however long the user requests. All information that was gathered would be stored in a JSON file (for device information) and CSV files (processes and kernel data). This process can repeat as many times as the user would like to. This is the object that all users who use this program would like to see but need some form of collecting and transmitting to a more readable and understandable format. Once this action is completed, the module is done and the data stored would be transferred to the next module for sanitization.

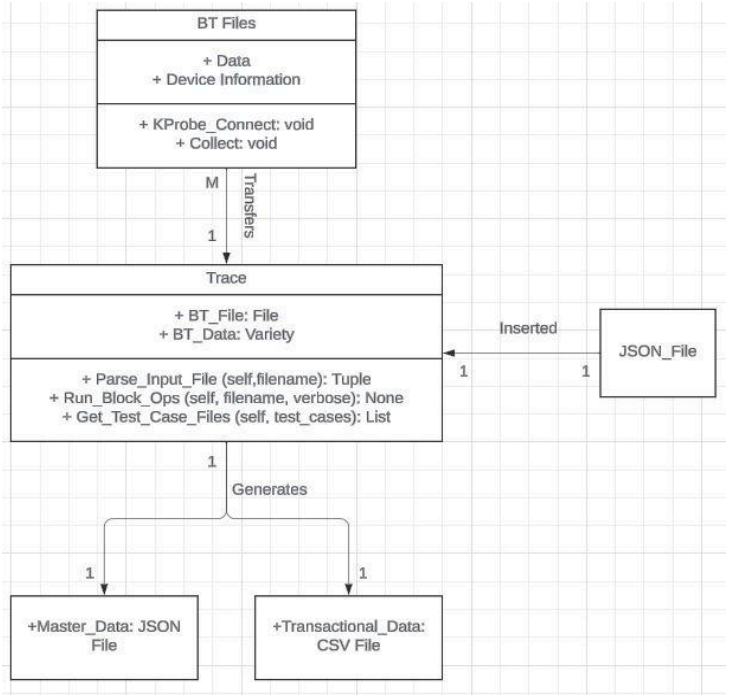


Fig. 02 Class diagram representing the Data Collection Module.

Inserted JSON File: The file is a collected request for the user to activate the program. The file so far contains “Process Time” and “Operation” but can be expandable.

Master_Data: This is a JSON file that would contain the device information. This is different from the Kernel information since the device information is more targeting what is being tested. Useful for companies when observing their own devices on the system

Transactional_Data: This is a CSV file that contains all the processes that are to occur at the kernel level. This would be associated with the device collected later in the modules.

BT Files

The BT files are files that eBPF operate with and the main purpose of the files are to collect the kernel level information and processes. Every file would have different information collected as well as the device information the activity is running on.

- KProbe_Connect: A method in eBPF where the program would connect to the kernel and trace the processes and information that is happening at that level.
- Collect: A method that rides off of the KProbe_Connect method where the data is collected at the level. The data collected is varied based on the BT File requirements and the kernel information.

Trace Class

The Trace class helps with collecting the information captured by the BT Files into the python program. Storing all the data so far collected into two different file types, JSON file for the Device Information and CSV File for the kernel information.

- Parse_Input_File(self, filename): This reads the inputted JSON file the user would provide for the BT processes. Once the file is read and gathered, it would return a tuple of all requests for the user.
- Run_Block_Ops(self, filename, verbose): This runs all the BT files that target block io operations. It traces the kernel operations until the specified process time ends.
- Get_Test_Case_Files(self, test_cases): This gets all BT files associated with specific operations and returns the list of filenames. Each to be ran within its corresponding method.

Module 2: Data Sanitization

Once the Data Collection module has completed its tasks, the data that was stored in the csv file and json file would be transferred to this module. This module is called Data Sanitization and the intent is for the clean up of the data that has been collected. Originally in the collection, there is extra information that would be seen as useless or white noise for the user so the sanitization would clear up the data either automatically such as the NULL column at the end of every BT collection or manually where the user filters certain data they would not like transferred, keeping data sizes manageable and easier to read. When at the end of the program, having white noise and useless information erased would help both the user and the program display information with ease. Upon completion of this action, the user can look at the data manually to observe their own cleaned data or they could load it into the next module, Data Storage.

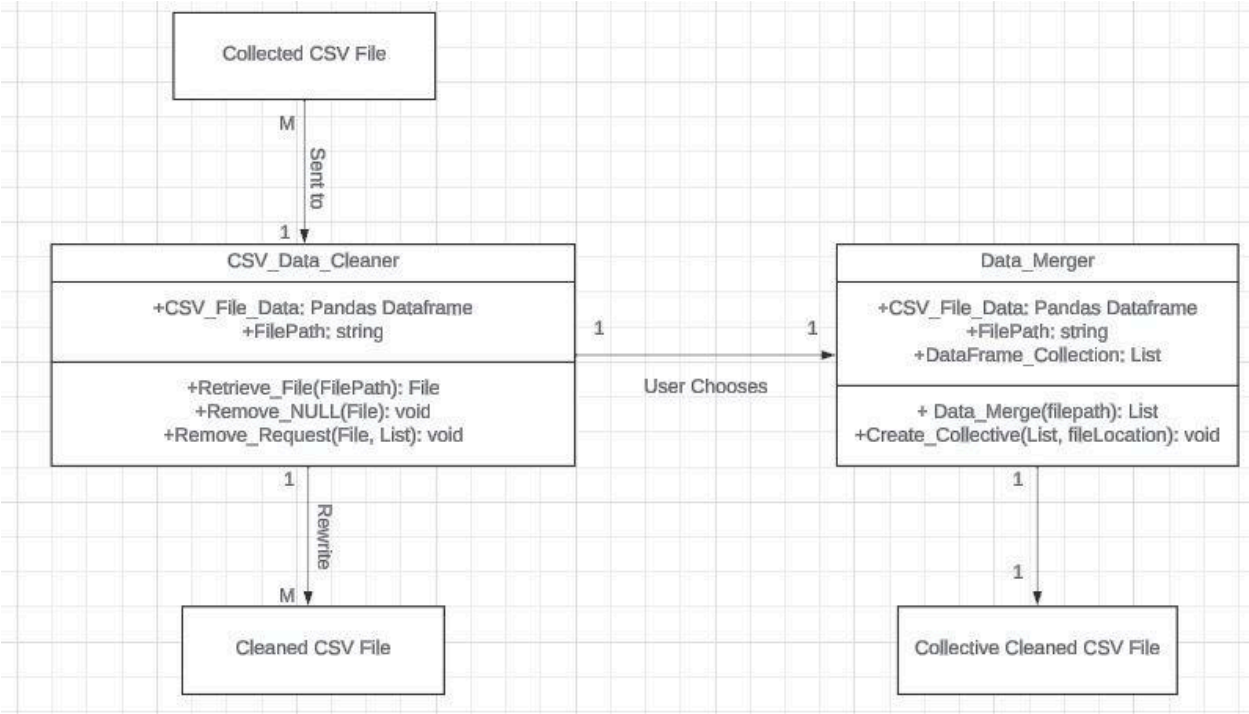


Fig. 03 Class diagram representing the Data Sanitization Module

Collected CSV File: The collected information so far collected from the previous module.

Cleaned CSV File: The collected data is cleaned from everything upon the NULL values and the requests taken by the user.

Collective Cleaned CSV File: Same as Cleaned CSV File except all files that were collected are merged into a single file. The user has to choose this option for it to work.

CSV_Data_Cleaner Class

This class is responsible for taking in the CSV files that have been collected and being cleaned of useless information and white noise of the user's selection. There will also be additional functions where there would be responsibility for merging all data so far collected and cleaned by the system into one CSV file. Only activates upon the user's request.

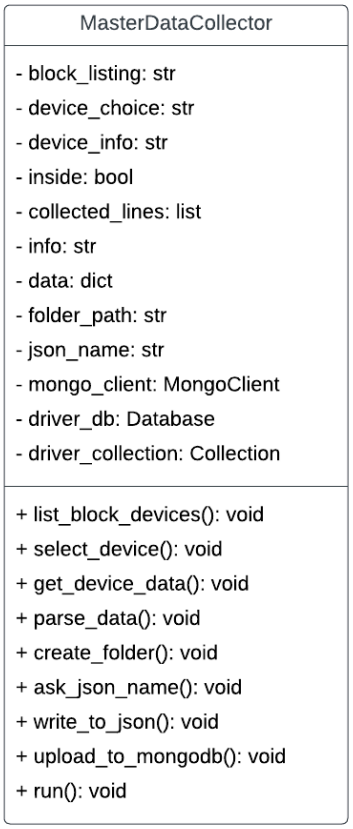
- Retrieve_File(FilePath): Retrieves the file stored in the specified directory and returns the information held inside the file. The file is a CSV.
- Remove_Null(): The function takes the File and removes the final column in the file that contains only NULL values as the system forces that empty column in the data collection.
- Remove_Request(File, List): Takes in the File and removes all requested information that is stored in the list. Preferably the List contains Labels. Ex: Idle, Processes, Error, etc.
- Data_Merge(filepath): Searches through the specified file path and grabs all related files through the use of glob, storing them into a List of Dataframes.
- Create_Collective: Takes the List of Dataframes and merges them into one, soon being written up into a CSV file based on the requested location.

Module 3: Data Storage

With the now cleaned and more readable data, the data is then transferred to this module where it would be stored into the respected databases, mongoDB (Device Information/Master Data) and mySQL (kernel data/Transactional data). This would be used as a standard for database storage on the local side. For whenever the user has their own database, they could implement the connection from the data files towards their databases. If not, they could use this for their storages. They must implement their information though to access their own database and not the project's. Once the data is stored, there will be scripts that would be readied prior to managing the data that exists in the database. The intention for the use of the

databases is when it would come to analyzing the data and displaying it which is in the upcoming modules, the databases are housing all the data collected and would use the openSearch platform at the near end of the program. With this though, the user can be led into the next module where all the data would be queried and analyzed.

MongoDB Class (Master Data Collector):



Services Provided

1. List all block devices connected to the computer.
2. Select a block device to fetch data from.
3. Get data of the selected device using smartctl.
4. Parse the obtained data to extract specific master data for the device.
5. Create a folder if it doesn't exist.
6. Prompt the user to enter a name for the JSON file.
7. Write the parsed data to a JSON file.
8. Upload the data to a MongoDB database.

Public Methods

- **list_block_devices()**: Lists all block devices connected to the computer.
- **select_device()**: Allows the user to select a block device.
- **get_device_data()**: Fetches data of the selected device using smartctl.
- **parse_data()**: Parses the obtained data to extract specific master data for the device.
- **create_folder()**: Creates a folder if it doesn't exist.
- **ask_json_name()**: Prompts the user to enter a name for the JSON file.
- **write_to_json()**: Writes the parsed data to a JSON file.
- **upload_to_mongodb()**: Uploads the data to a MongoDB database.
- **run()**: Executes all the above operations in sequence to collect, parse, store, and upload the data.

MySQL Class (ManageMySQL):

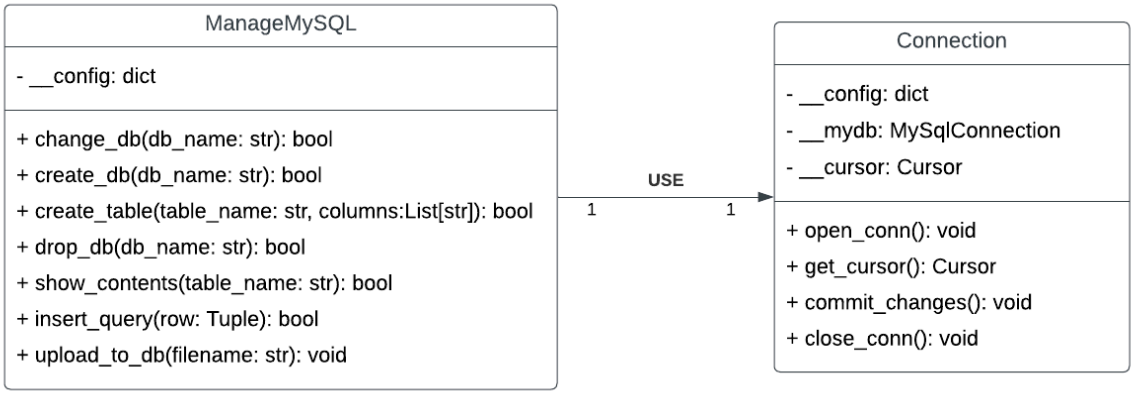


Fig. 04 Class diagram representing the Data Storage Module.

Connection Class

The *Connection* class provides services for connecting to a MySQL database and managing the connection.

- **open_conn():** Opens a connection to the MySQL database.
- **get_cursor():** Returns the cursor object for executing queries.
- **commit_changes():** Commits the changes to the database.
- **close_conn():** Closes the connection to the database.

ManageMySQL Class

The *ManageMySQL* class offers methods for managing a MySQL database, including creating, modifying, and dropping databases and tables, as well as uploading data from a CSV file.

- **change_db(db_name: str):** Changes the current database to the specified database.
- **create_db(db_name: str):** Creates a new database with the specified name.
- **create_table(table_name: str, columns: List[str]):** Creates a new table with the specified name and columns.
- **drop_db(db_name: str):** Drops the database with the specified name.
- **show_contents(table_name: str):** Shows the contents of the specified table.
- **insert_query(row: Tuple):** Inserts a new row into the database table.
- **upload_to_db(filename: str):** Uploads data from a CSV file to the database.

Module 4 and 5: Query Data and Visual Dashboard

Once all the data is stored in a database and cleaned up from prior activities, this module will be handling the querying, analysis, and displaying of the data. The program will utilize OpenSearch to request ALL files revolving around the query request and gather everything together meant for the user. Once the user obtains all related files in a specific location, the data would run through all the information stored under the related Master data. Once everything has been queried, there would be thresholds and flags called upon the data. The thresholds are the boundaries the device being tested are not supposed to pass. In the case of hard drives and SSDs, there should be a read and write speed of which should not be surpassed. If passed, a flag would be marked and only when a certain amount of flags have occurred, would send some form of notice towards the user, likely on the display portion. These repeated flags can only occur when there are more than five data files collected as only observing one data file would not identify these things called “Silent Failures''.

After the data is processed by OpenSearch, all data is organized and is ready to be displayed for the user to understand their SSDs’ performance visually. This will be done using OpenSearch Dashboards and display graphs and flags that have occurred. It would also show analytics on how many tests were run in each data file. Once the data is displayed properly, the module as well as the program is complete.

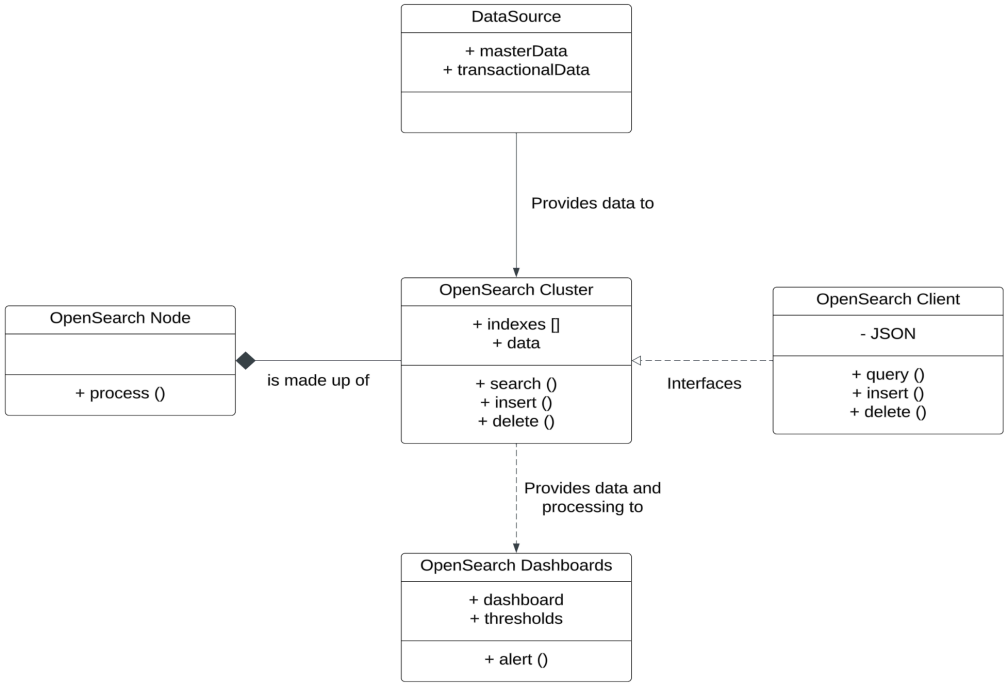


Fig. 05 Class diagram representing the Querying and Visual Dashboard Module

Data Source:

- Data is pulled from the database, in the form of csv files. JSON documents will provide OpenSearch with indexes for analyzing and querying data.
- Data can be updated and queried at any frequency, depending on how many OpenSearch Nodes the user is running.

OpenSearch Node:

- OpenSearch nodes can process queries, handle document status, and serve requests as needed.
- Any number of OpenSearch nodes can be run at any configuration the user needs, depending on the amount of data they are dealing with.

OpenSearch Cluster:

- The OpenSearch Cluster is made up of nodes and is the main processing unit for the user's data.
- Clusters serve data to the OpenSearch Client and to Dashboards.

OpenSearch Client:

- The client can be used to search through data, add data, or delete data manually.

Opensearch Dashboards:

- Dashboards will be the main interface, providing the user with a visual representation of their SSDs' performance.
- Alerts and thresholds can be set, and data can be observed for one time testing or long term data collection.
- Dashboards can be configured to the user's liking, but preset Dashboards will be created for the program's delivery.

Implementation Plan

Design Implementation Timeline

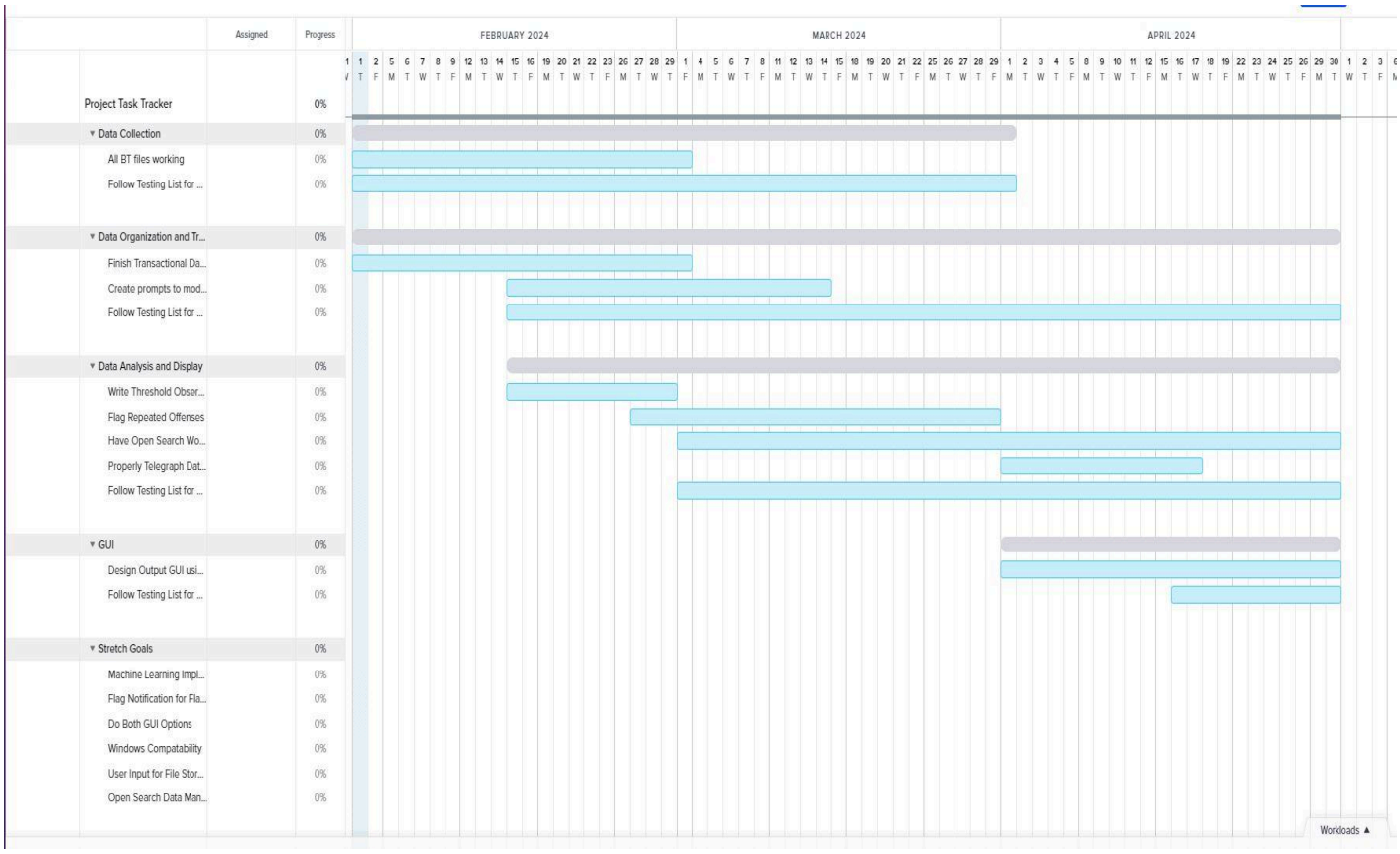


Fig. 06 Gantt Chart representing the project's Implementation Timeline

Explanation of Major Development Phases

As shown in the Gantt chart, there are four sections in the project that are expected to be finished before the semester ends. For the first section which is the Data Collection, there is only need of having the BT files that are used for data extraction to be completed. There are currently two files that have issues but the program for data collection has been completed. There would be testing done to ensure the program does not have gaps. In conjunction, Data Organization and Transmission is to also be done at the same time as finishing the BT files. Notably having the transactional data collected be stored properly in the mySQL database. These are of higher priority as the data collection of the kernel system is the focus of the project, especially when looking into the future applications companies would like to use. When the organization and transmission is mostly complete, the next section, Data Analysis and Display, is to be started on. This section is to analyze all the data collected and properly display onto openSearch which would be worked on once the data collection and Transactional data tasks are completed. The last section which is creating the output GUI is working in tandem with openSearch. This would be more focused on how the outputs of the analysis are to be seen and polishing of the GUI initially created from openSearch. The extra sections which are not given a time frame are the stretch goals of the projects. These goals are non-essential to the program but when the expected times are completed faster than listed for the essential sections, the stretch goals can be worked on.

Team Member Assignments

The assignments to the sections are as followed

- Section 1: Data Collection - Bailey, Erick
- Section 2: Data Organization and Transmission - Erick
- Section 3: Data Analysis and Display - Jake, Nick
- Section 4: GUI - Nick
- Testing - Jake

The stretch goals are a chosen list for a person to grab and complete but there is no requirement to attempt these in the first place.

Conclusion

Recap of Project's Value and Importance

In conclusion, the absence of an efficient observability framework to capture insights into system kernel behaviors at various stack levels poses significant challenges for companies testing SSDs across diverse applications. This lack inhibits proactive failure detection in real-time and near-real-time scenarios, resulting in increased time and financial costs as well as diminished customer satisfaction. By implementing our proposed observability framework, companies can revolutionize their validation processes, enabling proactive issue detection through comprehensive visualization of kernel stack behaviors. This not only enhances product quality but also empowers companies to gain a competitive edge by swiftly addressing performance issues and optimizing product offerings.