NORTHWEST NAZARENE UNIVERSITY

Quick Triage Tool

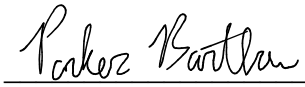
THESIS
Submitted to Department of Math and Computer Science
in partial fulfillment of the requirements
for the degree of
BACHELOR OF SCIENCE

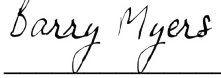
Parker Bartlow
2024

THESIS
Submitted to Department of Math and Computer Science
in partial fulfillment of the requirements
for the degree of
BACHELOR OF SCIENCE

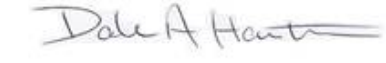
Parker Bartlow
2024

Quick Triage Tool

Author: 
Parker Daniel Bartlow

Approved: 
Barry Myers, Ph.D., Professor,
Department of Mathematics and Computer Science, Faculty Advisor

Approved: 
Mark Michaelson, M.A., M.Ed., Assistant Professor, Department of Academic Services, Second Reader

Approved: 
Dale Hamilton, Ph.D., Chair,
Department of Mathematics and Computer Science

Abstract

Quick Triage Tool.

BARTLOW, PARKER (Department of Mathematics and Computer Science),
MYERS, DR. BARRY (Department of Mathematics and Computer Science).

Hewlett-Packard (HP) LaserJet printers undergo rigorous quality testing to confirm they are reliable and efficient before they are sold to the public. Much of the LaserJet firmware testing is automated, producing thousands of failure logs daily. The process of analyzing logs to determine the failure type (triaging) was manual, leading the sheer magnitude of failures to outpace the capabilities of the people responsible for diagnosing them. This project aimed to automate parts of the triage process so that time and effort could be better allocated. The Quick Triage Tool is a full-stack tool that consists of a MongoDB database, C# middle logic, and a Vue.js front-end website for database management. From the front-end, users create objects called rules which contain symptoms and represent failure types. Firmware test failure log files are automatically searched against the database of rules, finding failure types when all rule symptoms are matched and displaying results on a common internal website. The automation of the triage process saves time by eliminating manual triaging and by preventing redundant triage efforts. Automation also increases confidence in quality metrics. Lastly, the aggregation of failure types allows failure prioritization, which increases test passing rates and ultimately firmware quality.

Acknowledgements

I would like to thank my project mentor for his guidance, expertise, patience, and encouragement throughout the time I worked on this project. I would also like to thank my technical mentor for his help learning new programming frameworks and technologies. I would like to thank my manager for guidance and perspective in navigating the professional workforce of my field for the first time. I would also like to thank the other intern that helped in this project for her teamwork and leadership during my first summer. Lastly, I would like to thank my dad for helping me navigate the HP Inc. culture and helping me process the amount of new information I was learning and the challenges I was facing.

Table of Contents

List of Figures

Introduction

Project Overview

The Quick Triage Tool provides functionality to automatically determine the reasons for firmware test failures once those reasons have been previously discovered. The tool is based on the idea of custom rule objects and consists of a Vue.js front-end website, a MongoDB database, and custom C# searching logic. Rules contain the information necessary to match failed test logs with failure types and are created, edited, tested, and deleted from the Vue.js front-end website. The tool is integrated with two sources of test failures, outputting results to two different locations for the user to see.

Background

One of the many lines of products that HP (Hewlett Packard) Inc. makes is the printer. HP Inc.'s Boise site is the primary location for the business behind and design of the company's LaserJet Printers. Both *The New York Times* and *U.S. News* has an HP LaserJet printer as their top laser printer of 2024 (John, 2024; Keough & Wells, 2024). One of the ways HP Inc. upholds this standard of quality is by rigorous automated testing.

Firmware, the code that runs on the printer's specialized computer, goes through daily automated testing, and produces three thousand to five thousand failed tests each day. Engineers and programmers use this failure information to fix issues before products ever reach customers, ensuring the delivery of high-quality, reliable printers. However, each test failure produces nearly 30 text fail logs, which are files full of obscure and highly technical text strings. These files constitute one failed test bundle and are stored in a .zip file. Specialized programmers, who, for the remainder of this paper will be referred

to as triagers, know what areas of these files typically contain the type of test failure and manually search through and analyze the fail logs to determine why the tests fail.

With thousands of daily test failures, there is no team large enough to manually triage every test failure. This old system leads to several problems. The first is that many test failures are not analyzed each day, meaning if the test's reason for failure is unique, it will not be caught by triagers. A second problem is that since many tests fail for the same reasons, redundant effort is spent rediscovering the same failure types. A third problem arises when a test starts failing for a different, more serious reason than it previously was failing with. In this case, it will not be caught as the test will be assumed to be failing for the same reason it previously was. All of these problems lead to inefficiency in the triaging process and ultimately lower firmware quality than what is possible.

Requirements

To solve this problem, a tool, later called the Quick Triage Tool, was proposed that would automate the triaging process for firmware test failures. For automation to work, there needed to be a way to create and store symptoms of test failure types that have been discovered by triagers. Then, if symptoms of a test failure can be matched in the test failure log files, the failure type can be known without redundant manual effort.

This tool also needed to work with test failures produced from the two different firmware testing sources HP Inc. uses. Both test sources organize fail logs differently and provide unique challenges to integrate with the Quick Triage Tool. Lastly, results from the automated triaging must be displayed in a meaningful way to triagers and other relevant employees.

Constraints

HP Inc. employees in firmware lacked the bandwidth to devote significant time to developing a tool such as the Quick Triage Tool, so the responsibility was given to interns. This meant that work must be completed within the summer internship window, from late May to early August. Additionally, there were cross-team dependencies with an HP Inc. team located in India. This meant that there were small windows of time that both parties were working at the same time, and that much email communication passed through half-day buffers, delaying the speed of progress.

Quick Triage Tool Design

Requirements Gathering

The project mentor had the best understanding of the requirements and goals of the tool. A technical mentor was also available that had more knowledge of the specific frameworks and technologies that were being used. Weekly meetings with these two individuals provided ample space for updates, questions, and additional requirements. The project mentor served as a bridge between the users of the tool, triagers, and the developers of the tool, the interns.

The Rule

A rule is a custom object that represents a test failure type. Rules contain symptoms which are used by the Quick Triage Tool to match to the contents of the fail logs from a test failure. The structure of the rule object is as follows:

name (String)
dateAdded (DateTime)
description (String)
lowPriority (Bool)
symptom (List<custom Symptom object>)
    logfile (String)
    searchString (List<String>)

Figure 1. Rule Object Structure

The name of each rule should sufficiently name the test failure type that rule represents, usually in the form of the error code. The dateAdded attribute is automatically populated upon the creation of the rule using the built in DateTime.Now constructor. The

description of each rule should provide any additional information about the test failure type past just the name of it.

The lowPriority attribute is used only when two or more rules are matched to a test failure's log files. This, in practice, means that the tool found multiple possible reasons that the firmware test failed. However, if one matched rule has a true value for lowPriority, then the other matched rule is deemed more important and chosen as the true failure type. This is most often used for failure types that involve the testing framework itself, such as a loss of emulator power.

Symptoms are an additional custom object that are nested within the greater rule object. To find the type of test failure, certain error codes or other text strings must be found in specific log files from that failed test. Each symptom contains the name of a logfile and a list of the text strings that should be found in that logfile. Each rule has a list of symptoms, as sometimes multiple matched symptoms are required to determine the type of test failure.

Rule Database

Once a triager manually determines the failure type of a failed firmware test, a rule can be created to represent that failure type, and then used to automatically discover all recurrences of that same failure type. Naturally, this requires that rules be stored somewhere once created.

MongoDB is a NoSQL database, meaning it is less structured and rigorous than its SQL database counterparts (What is NoSQL?, 2024). MongoDB was chosen as the database to store rules because of the complexity of the rule object and the resulting

simplicity in database operations with those rules objects. HP Inc. also already had an existing MongoDB server that was easily able to be utilized for this tool. NoSQL databases handle complex objects well, as each item in the database retains the structure of the object stored in it without much additional programming overhead.

Rule Manager Website

Rules represent test failure types, and they can be stored in the MongoDB. However, there must be some way to perform operations on these rules, such as viewing them, creating, deleting, or editing them, and even testing them. To avoid triagers directly accessing the database through a MongoDB application, the creation of a website was chosen as the method of database management. This website is hosted on a local HP Inc. server, limiting access to employees only. The website, plainly called the Quick Triage Rule Manager, consisted of multiple tabs and tools that allowed users to manage the rule database.

Vue.js

The rule manager website was built utilizing the Vue.js framework, an open-source JavaScript user-interface framework that aids and helps organize web page design and functionality (Vue.js, 2024). Many other frameworks could have been used, but this framework was recommended by the project's technical mentor. The best and most utilized features that Vue.js provided were reactivity and component-based architecture. For reactivity, Vue.js uses data binding, meaning that when users changed underlying data such as table data, the table would reflect those changes without requiring a refresh

of the website. The component-based architecture was the most helpful Vue.js feature, as it made user interface (UI) creation and interaction much easier.

Custom components such as the “Test” component can be reused throughout the JavaScript front-end, preventing the need for duplicated and redundant code. For example, since the testing functionality can be accessed from either the testing page from the navigation bar or the individual rule itself in the “View and Manage Rules” page, the component is simply created once and imported to both of those web pages. This modularity increases readability, efficiency, and consistency between web pages. When components are nested and reused, there are child and parent components. Sometimes data must be passed from one component to the other, and Vue.js handles this through the prop object. This provided framework saves a significant amount of manual JavaScript labor. Functions of the rule manager website are listed below.

View

To view the current rules in the database, users select the “View and Manage Rules” page on the website’s navigation bar. This page contains a table that pulls the data from each rule in the database and displays the most basic information about the rules: name, data added, and description. Each row corresponds with one rule, and each has four action buttons: edit, delete, information, and test. See Figure 2 below.

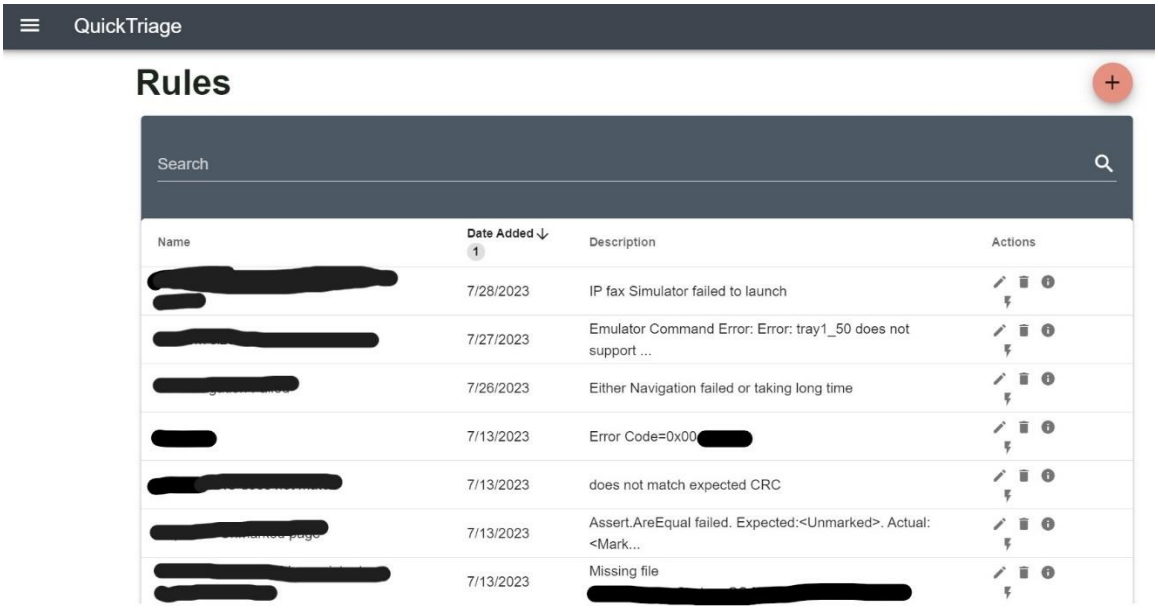


Figure 2. Rule Manager Website: View

The first action button, “edit”, allows you to edit individual attributes of any specific rule, and the second, “delete”, allows you to delete a rule completely. The latter may be done if an incorrect rule is found or if that failure type no longer applies to the current firmware testing methods. The third action button, “info”, shows a popup with all the rule’s attribute values, adding lowPriority and all symptom information. Lastly, the fourth action button, “test”, brings you to the test functionality of the website and automatically uses the rule whose “test” button the user clicked on. Testing will be described in greater detail in a following section.

Create

The “Create” web page is devoted to rule creation and can be accessed by either the navigation bar or the red plus from the “View and Manage Rules” page in Figure x.

From here, users can add any number of symptoms to their rule, but accuracy and correctness is heavily emphasized. See the empty “Create” page in Figure 3 below.

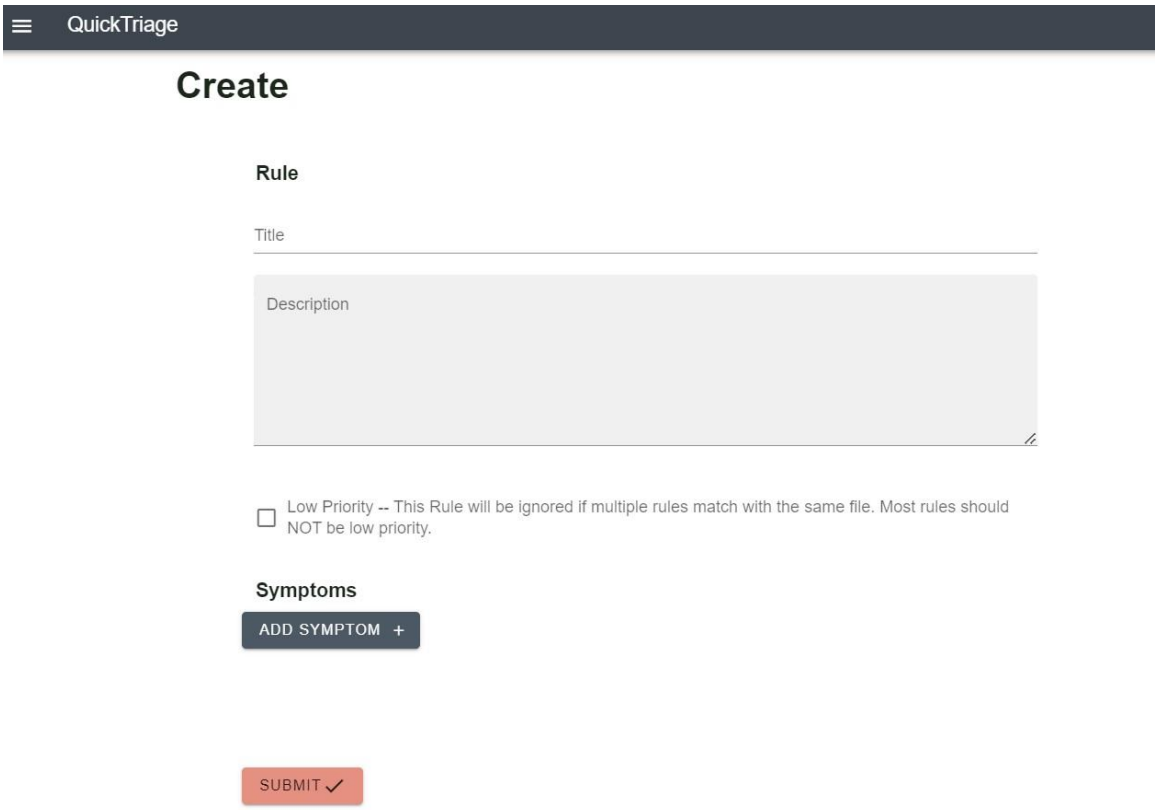


Figure 3. Rule Manager Website: Create

Test

Rather than relying on each user’s comprehensive and perfect knowledge of all failure types, a rule testing functionality is provided to ensure rule correctness and accuracy. The “Test” web page can be accessed either from the navigation bar or from the “View and Manage Rules” page. The testing functionality requires two inputs from the user: the path of a failed test bundle and the ruleID of the rule that is being tested. Ideally, once a triager discovers a new failure type, they would create a rule to catch that failure,

then test it here. They would input both the ruleID of their newly created rule and the path of a failed test bundle that they have manually confirmed to fail for the reason stated by their rule. See Figure 4 below for the “Test” page.

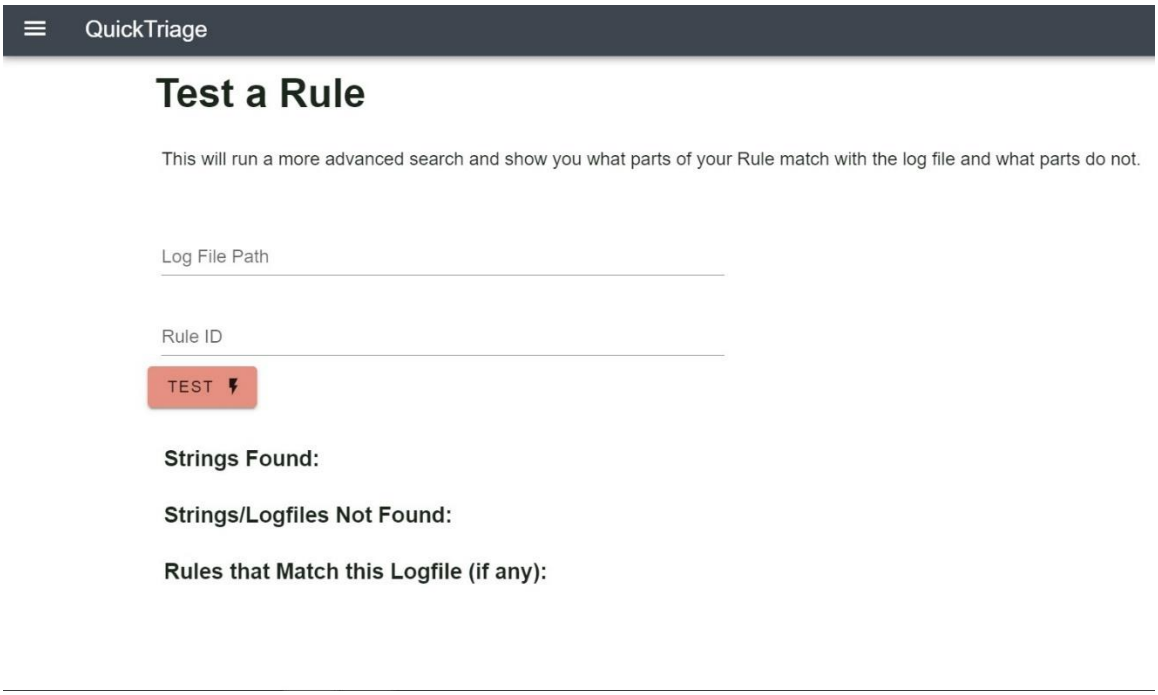


Figure 4. Rule Manager Website: Test

Below the test button, there are three categories of results from testing a rule. If a rule is not correct, it is because not all of its search strings within its symptoms were found in the log files. The first category, “Strings Found:” provides which of the tested rule’s search strings were correctly found in the log files. The second category, “Strings/Logfiles Not Found:”, provides which of the search strings were not found in the log files. Lastly, the third category, “Rules that Match this Logfile (if any):” provides the names of any additional rules that successfully fully matched with the failed test bundle.

This last result is helpful because the goal for the Quick Triage Tool is for each test failure to only match with a single rule. This testing information provides the user with enough information to see which parts of their rule are incorrect and if there are other rules that match, either because another user created a rule for the failure type already or other rules in the database are incorrect themselves.

Quick Triage Tool API

The Vue.js front-end JavaScript website on its own does nothing, as something needs to operate in between it and the rule database to handle the commands and transfer of data. This middleware is called an application programming interface (API), which connects the front-end and the back-end (Frye, n.d.). The Quick Triage Tool's implementation, the Triage Rules API, uses the REST API (Representational State Transfer Application API) architecture provided by ASP.NET and is written in C# (What is ASP.NET?, 2024). This API also handles the searching logic that matches rules to the log files found in failed test bundles.

Back-End Functionality

The REST API architecture is based on standard HTTP (Hypertext Transfer Protocol) methods GET, POST, PUT, and DELETE. Like other architectures, REST API uses unique URL (Uniform Resource Locator) endpoints for API functions. Additionally, REST API is a client-server architecture, which allows users to connect to the website (server) from their own devices (clients) over a network. This API is hosted on a local HP Inc. server that is on their private network.

Vue.js front-end operations discussed previously, such as the test a rule functionality, send an HTTP request to the API through a button press. A client module called Axios was installed to the front-end and allowed for easy HTTP requests that used the API URL endpoints to connect (Getting Started. Axios, 2024). For example, when a user presses “Test”, a JavaScript event-handler is fired off and sends an axios.get() request to the API’s test endpoint, which is roughly of the form <IP address>/TriageRulesAPI/rules/test. In code, this may look like Figure 5 below.

```
return axios.get(`${IPAddress}/TriageRulesAPI/rules/test
```

Figure 5. Axios Call

The axios client sends the packaged HTTP GET request to that API endpoint, and as long as the HTTP method matches the GET request (API method is a HTTP GET method), the request will successfully start up the correct API method.

The Triage Rules API also needs to be able to communicate directly with the MongoDB that the rules are stored in. For example, when users try to delete a rule on the rule manager website, the Axios request connects to the corresponding Triage Rules API endpoint, which then needs to contact the MongoDB and delete the specified rule. MongoDB provides a driver that can be installed to .NET applications to contact a MongoDB. The Triage Rules API uses the Rules MongoDB connection string to connect and verify the validity of the connection request.

Searching Logic

The Triage Rules API is not only responsible for handling requests between the rule manager website and the Rules MongoDB, but also houses the searching logic that matches rules to failed test bundles. The hierarchy and logic behind the searching operation is relatively complex, so this next section will go into it in depth.

The overall order and hierarchy of searching starts with the .zip file that houses the failed test log files. These log files are iterated through one at a time. Each time a log file is opened, it needs to be searched against the rules in the database at the time to look for matches.

However, there are several considerations and optimizations that speed up the brute-force nature of this approach. First, once the rule data is pulled from the database, a new dictionary called LookupDictionary is created with log files as the key. For each dictionary entry and log file key, there is a list of SearchData objects, each containing a rule ID and the corresponding search strings that should be found. See Figure 6 for the LookupDictionary structure.

Key: logFile (string)
Value: List<SearchData>
SearchData
ruleID (string)
List<searchString (string)>

Figure 6. LookupDictionary Structure

This is a different organization than previous methods, where each item is a rule, and each of those rules could have multiple log files associated with them. Now each

item is a logfile with the possibility of multiple rules associated with them. This means that if one rule has two symptoms (search strings to be found in two separate log files), it would be present in two LookupDictionary entries.

The result of this reorganization of data is that the program knows every relevant symptom of the entire rules database that needs to be searched for in a given log file. Returning to the previous general order of the search, the program looks at one log file at a time. Upon looking at a log file, the LookupDictionary is searched for an entry with the key equal to that log file. If one is found, the log file is opened, and the SearchData objects are iterated through. Log files are not opened if there are no relevant symptoms to be searched for in it, which speeds to process up as file opening is slow. This way, every rule that has a symptom in this log file is being searched for.

This brings up the next two new data structures, called the RuleList and NoFlyList. The RuleList is a dictionary used to keep track of whether a rule has been matched to a test failure bundle or not. The RuleList structure is in Figure 7 below.

Dict<key: ruleID (string); value: bool>

Figure 7. RuleList Dictionary Structure

All rules in the rule database are listed here, and all values start with the value true. Once a search string is not found in a log file, the corresponding rule ID in RuleList for that search string (found from the SearchData object in that LookupDictionary entry) is set to false because if even one search string is not present where it should be, the rule

is not a match. The next step involves the use of the NoFlyList, which takes the form of Figure 8 below.

List<ruleID (string)>

Figure 8. NoFlyList Structure

Once a search string is not found and the RuleList entry for that rule is set to false, that ruleID is added to the NoFlyList. While the search iterates through SearchData entries, the search strings are only searched for in the log files if the correspond ruleID is not found in the NoFlyList. Once a rule's search string is not found, the corresponding symptom cannot be found, and there is no need to search for that rule's other symptoms. This increases the speed of the search and eliminates redundant searches.

RuleList contains the final results of a search, and a method called GetTopRule iterates through the RuleList entries that have values true. It will return the matched rule (with RuleList value true) that has a lowPriority value of false. If there is more than one matched rule with lowPriority false, a random matched rule is returned, but there is a mistake in the structure of the rules which can be fixed through rule testing on the rule manager website.

Integration with Test Source 1

The tool functionality is present, however, a pipeline or workflow needed to be created to automate the flow of test failures into the tool's searching logic. The first source that test failures were produced from was simpler to integrate with the Quick Triage Tool, so the pipeline that was created functioned completely automatically.

Pipeline

Because the production of test failures may outpace the speed of the Quick Triage Tool at any point, the consumption of test failure bundles by the tool had to be asynchronous, but also needed a queueing ability. RabbitMQ (Rabbit Message Queue) is software that performs just these functions (RabbitMQ: One Broker to Queue Them All, 2024). RabbitMQ was already utilized by HP Inc. and hosted on an internal server. A new queue called Triage was created for this project, and the project's technical mentor connected the first test source with this queue. When firmware tests failed from this first test source, the failure bundles were sent to the Triage RabbitMQ queue.

This queue required the creation of a separate program called QuickTriageListener, which was a simple C# script that is continuously ran on the same common server as the API and is accessed via a virtual machine (VM). This script uses the RabbitMQ.Client library to enqueue a failed test bundle from the Triage queue if it has received an acknowledgement from the Triage Rules API that its resources are unused and available. The QuickTriageListener sends failed test bundles to the Triage Rules API for processing and searching, and once the API has finished, it sends acknowledgement back to the Listener so that it can enqueue its next failure bundle. If there is an unusually large volume of failure bundles, this queueing system ensures that the API is not overloaded and that all failure bundles are processed accordingly. See Figure 9 below, which is a flowchart for the test source one pipeline.

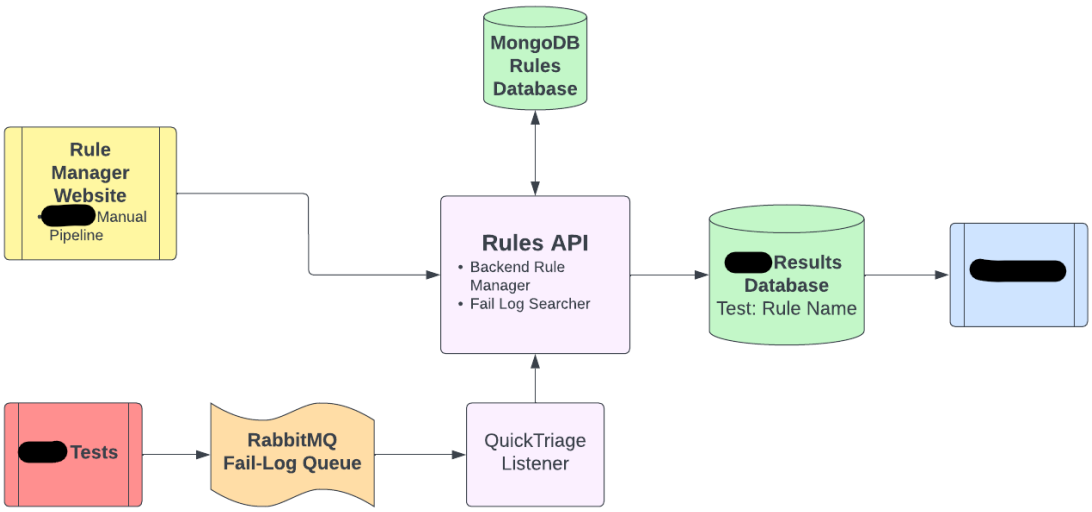
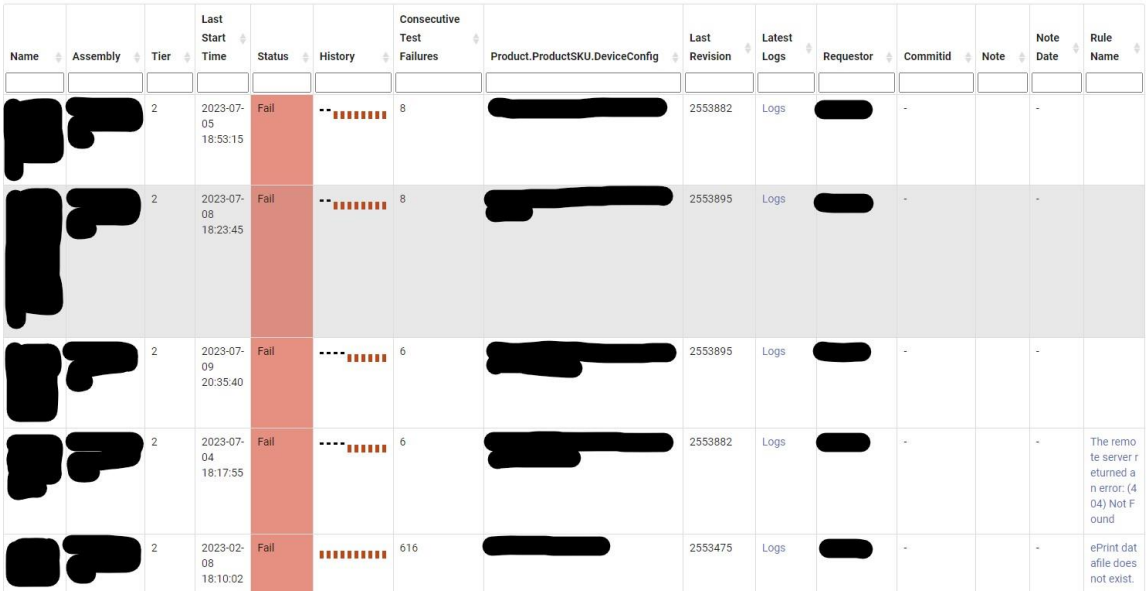


Figure 9. Test Source One Pipeline

Displaying Results

Firmware test data has been displayed on an internal HP Inc. website for quite some time, so the natural solution to where to display the matched rule results from our tool would be this website. The HP Inc. team in India runs this internal website, so the task required direct communication and cooperation with them, resulting in a new API endpoint they created on their website's API for our tool to send results to.

The internal firmware testing website already contained tables where each row corresponded to a firmware test. The HP Inc. India team's solution was to add an extra column to this table to list and link the rule that matched the test (if any). See Figure 10 below.



Name	Assembly	Tier	Last Start Time	Status	History	Consecutive Test Failures	Product.ProductSKU.DeviceConfig	Last Revision	Latest Logs	Requestor	Commitid	Note	Note Date	Rule Name
[redacted]	[redacted]	2	2023-07-05 18:53:15	Fail	[redacted]	8	[redacted]	2553882	Logs	[redacted]	-		-	
[redacted]	[redacted]	2	2023-07-08 18:23:45	Fail	[redacted]	8	[redacted]	2553895	Logs	[redacted]	-		-	
[redacted]	[redacted]	2	2023-07-09 20:35:40	Fail	[redacted]	6	[redacted]	2553895	Logs	[redacted]	-		-	
[redacted]	[redacted]	2	2023-07-04 18:17:55	Fail	[redacted]	6	[redacted]	2553882	Logs	[redacted]	-		-	The remote server returned an error: (404) Not Found
[redacted]	[redacted]	2	2023-02-08 18:10:02	Fail	[redacted]	616	[redacted]	2553475	Logs	[redacted]	-		-	ePrint datafile does not exist.

Figure 10. Test Source One: Display Individual Results

Triagers can use this table to find which firmware test failures do not match with rules, download the failed test bundle, and manually determine the reason for the test failure. Upon manually diagnosing the failure type, they can create a new rule to catch that failure, test it, and eventually see that result on the internal website.

Matched rule results from the Quick Triage Tool also contributed to a completely new table the HP India team created on the internal firmware testing website. In this table, each row represented a rule, which represents a test failure type. This table aggregates occurrences of rule matches, which provides crucial and invaluable information that will be discussed in the conclusion of this paper. See Figure 11 below for the aggregation table.

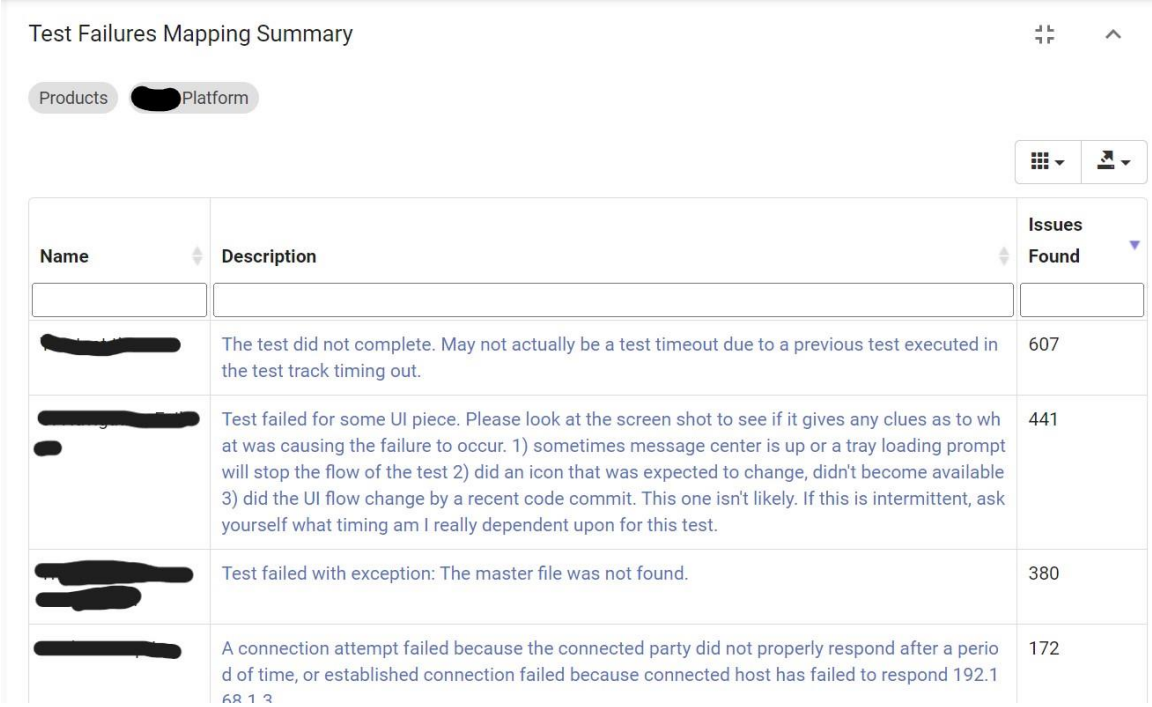


Figure 11. Test Source One: Display Aggregated Results

Retroactive Rule Testing

With this automated pipeline, there is one large problem that can occur. Automated firmware tests execute regularly, but at different intervals. Some tests may only execute once every two weeks, for example. Consider a two-week test that fails and doesn't match with a rule. A triager may analyze it the next day, determine the new failure, and add a new rule to the database to catch it. However, it will be nearly two weeks before the test runs again with the new rule matching it and getting displayed. This significantly delays the speed at which firmware problems are found and fixed.

To counteract this problem, retroactive rule testing was implemented. A short C# script was created that sent an HTTP request to the Triage Rules API's searching logic method to retrigger it. First, however, the most recent test failure bundles had to be pulled

from the network file share. Normally, failure bundles were sent to the Triage queue upon failure, then sent to the API. But, with this retriggering, the searching logic still needed access to the most recent failure bundles for each firmware test. The HP Inc. team in India provided a URL to access the most recent test failure bundles and their corresponding matched rule from our tool (if it exists). Next, this script was set to run twice a day by utilizing Windows Task Scheduler. This way, if there are any changes in failures for tests with a long interval between executions, they will be known within twelve hours at the most. New retroactive search results are only sent to the results website if the matched rule is different than the previously matched rule for a failed test bundle.

Integration With Test Source 2

The second test source for automated firmware testing is much more complicated. An automated integration of the Quick Triage Tool and this second test source would require much more time and resources than were available for this intern project. Instead, a manual workflow was implemented to leverage the Quick Triage Tool's capabilities with the second test source.

Workflow

Since there was no realistic way to automate the pipeline of failed test bundles from the second test source to the Quick Triage Tool, it was decided to add a page to the Rule Manager website. This page contained a manual integration with the second test source, requiring. Failed test bundles from the second test source are also stored on the same network file share as the first test source, but the file hierarchy differs. The same

level of file path as the first test source results in a .zip file full of .zip files that each contain the log files that constitute a failed test bundle. This additional layer was handled through some simple file extension checking using both the Directory and Path classes in C#'s provided System.IO library.

The new web page on the rule manager website is dedicated to searching failed test bundles from the second test source manually. There is input for the user to enter the path of a .zip of failed test bundles, then a button to begin the searching process. The search uses the same searching logic from the Triage Rules API as the test source one pipeline. See Figure 12 below to see this additional web page along with its results from a search.

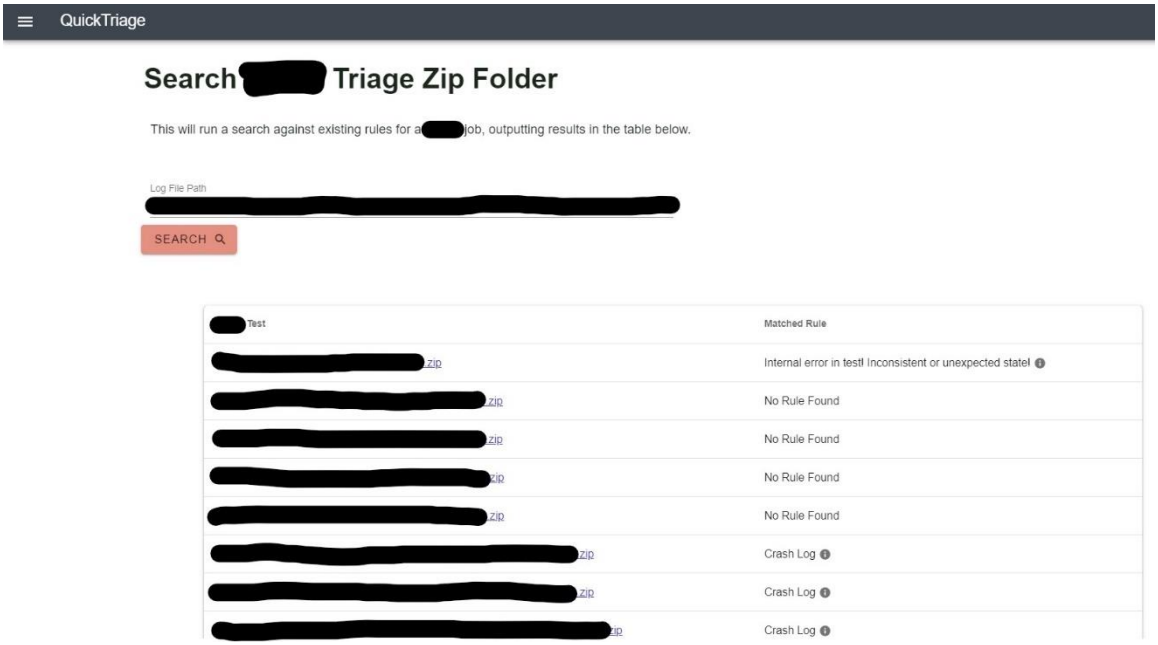


Figure 12. Test Source Two Integration Web Page

Firmware tests that execute from the second test source are not included on the same internal firmware testing website, so Quick Triage Tool results had to be displayed elsewhere. Since .zips from the second test source included several failed test bundle .zips, the search logic runs multiple times and finds a matching rule for each inner .zip failed test bundle. These results populate a table component on the web page. This table's first column has the path of each failed test bundle, and the second column contains the rule that matched, if any. Users can download a CSV containing the table data to their own machine since there is no existing common website to share this data.

By searching a .zip full of failed test bundles, a triager can see which bundles did not match with a rule. At that point, the triager can download the failed test bundle to their machine by clicking on the path in the left column. Once they have manually discovered the failure type of that failed test, they can add a rule to the database and re-run that search to see if a rule now matches it. While this workflow involves more manual work than the pipeline of test source one, it still greatly improves the efficiency of the triaging process.

Presenting and User Guide

Once the Quick Triage Tool was in a working state, the tool had to be marketed and taught to the triaging team, which is largely based in India. A user guide PowerPoint was created that walked through the new, refined triaging process step-by-step. This PowerPoint was explained to the team over Zoom but is comprehensive enough by itself to guide a triager to the correct use of the tool. The PowerPoint itself was put on the team's SharePoint site so that they could reference it as they learned the tool themselves. See Appendix A for a redacted version of this user guide.

Conclusion

Impact

The Quick Triage Tool is a comprehensive and multi-faceted tool, but its impacts can be summarized into three areas.

The first impact of this tool is that it eliminates the need to spend effort and time rediscovering previously known failure types. Once a failure type has been manually discovered, the Quick Triage Tool will discover every subsequent occurrence of the same failure type automatically.

The second impact of this tool is issue prioritization which is possible through the aggregation of rule match occurrences by the HP Inc. team in India. Employees responsible for fixing firmware issues can prioritize which issues to fix first by which rule has the most matches to firmware test failures. Once higher-prioritization issues are fixed, the number of failed firmware tests dramatically decreases.

The third impact of this tool is that it improves the company's confidence in their quality metrics. In the previous manual triaging workflow, once a test failure type was found, that was assumed to be the reason for failure for every subsequent failed execution of that test. However, the reason that a firmware test fails can change between test executions, especially to a more harmful failure type. The old system would not catch this change, however, through the automation of the Quick Triage Tool, it would either return a rule for the new test failure reason, or not return a rule at all because the failure type is new. This means that test passing and failing rates are now more accurate, and the

company's list of ongoing firmware issues and their prevalence should also be more accurate.

All of these impacts and features of the tool fall under two broad improvements. Because of the new, efficient, and expedited process, the Quick Triage Tool ultimately increases printer firmware quality. The effort of triagers can be better allocated to discovering new failures, and resources can be better spent on tackling the most prevalent and pressing issues first. Along with increasing printer firmware quality, the Quick Triage Tool also ushers in quality-of-life improvements for triagers, especially because redundant work is eliminated. The firmware testing triage process has been completely upgraded and refined.

Future Work

The Quick Triage Tool could be improved by adding automatic integration with the second test source. The current manual integration works well but is slower than the automatic pipeline that exists for the first test source. Additionally, the prioritization benefits of the aggregation of matched rules are not available in the manual second test source integration. This would be a major advantage if there was an automated pipeline from test source two to the Quick Triage Tool.

HP LaserJet printers are in the process of transitioning to a new codebase based in C++ instead of C#. With this change, the firmware testing process will change. The current Quick Triage Tool pipelines would no longer work, but the logic behind the tool can be universally applied, so a new version of the Quick Triage Tool could be made to work with the new codebase. The benefits of the tool are too good not to leverage.

HP Inc. uses bug-tracking software to track progress on firmware issues that are in progress of being fixed. Future work could include integration of this bug-tracking software with rules, so that employees can see how close the failure type a rule represents is to being fixed.

Lastly, more effort could be spent to market the Quick Triage Tool to encourage greater use amongst triagers and other departments. The tool works the best when everyone is using it and making rules to cover all new failure types. At the presentation of this tool, several separate departments outside of firmware took great interest, wondering if similar tools could be made for the testing they perform.

Review

The Quick Triage Tool is a tool created to automate the process of discovering the failure types of failed firmware tests. The tool is based on a custom object called a rule that represents test failure types. It involves a Vue.js front-end website to manage the rules which are stored in a MongoDB database. The Triage Rules API connects the two, along with housing the searching logic to match rules to failed test bundles. An automated pipeline exists to pull failed test bundles from one test source, run them through the tool, and display results on an internal website. The second test source has manual integration with the tool. The Quick Triage Tool significantly improves the efficiency of the triaging process, ultimately leading to higher printer firmware quality.

References

Broadcom. (2024). *RabbitMQ: One broker to queue them all*. RabbitMQ. https://www.rabbitmq.com/

Frye, M.-K. (n.d.). *What is an API?*. MuleSoft. https://www.mulesoft.com/resources/api/what-is-an-api

Getting started. Axios. (2024). https://axios-http.com/docs/intro

John, S. (2024, February 1). *Best laser printers of 2024: Expert picked | U.S. news*. U.S. News & World Report. https://www.usnews.com/360-reviews/technology/best-laser-printers

Keough, B., & Wells, K. (2024, April 1). *The best laser printer*. The New York Times. https://www.nytimes.com/wirecutter/reviews/best-laser-printer/

Microsoft. (2024). *What is ASP.NET?*. Microsoft .NET. https://dotnet.microsoft.com/en-us/learn/aspnet/what-is-aspnet

MongoDB. (2024). *What is NoSQL?*. MongoDB. https://www.mongodb.com/nosql-explained

Vue.js. (2024). https://vuejs.org/guide/introduction.html

Appendix A: Quick Triage Tool User Guide

Quick Triage Tool User Guide

Outline

- Purpose
- Process Overview/Review
- (1) Rule Manager Website
- (2) Test Source 1 Process
- (3) Test Source 2 Process

Purpose

- The Quick Triage Tool (QTT) automates some of the triaging workflow, making your job easier!
- By keeping track of discovered failures and having an automated searching system, you don't need to spend time rediscovering known failures.
- Allows you to spend your time finding unique failures.

Process Overview

- The **Rule Manager** website allows you to manage a database of rules.
- Rules are custom objects that represent failures (ex. [redacted] crash)
- Failed firmware test logs can get sent through the QTT and searched against this rule database, returning the matched rule for the failed test run.
- For **Test Source 1**, the process is automated. You only create rules based on what you see from the results website, and the results are output on the results website.
- For **Test Source 2**, you'll need the path of a failed test run on the network share as input to make the search happen, and the results output as a table on the Rule Manager Website.

1. Rule Manager Website (1)

This section is a walkthrough of the QTT Rule Manager Website and its functionality. Please refer back to this section when reading about the Test Source 1 and Test Source 2 processes. Website URL: http:/[redacted]/RuleManager/#/

As a general note, when navigating popups, use the 'CLOSE' button in the bottom right-hand corner instead of the browser's back arrow.

Rule Manager Website (2)

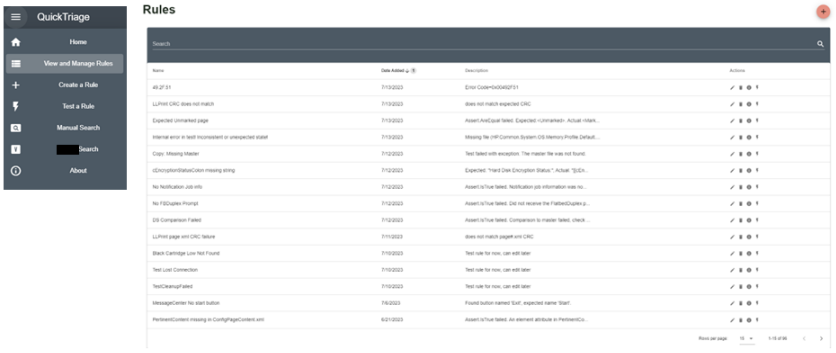
Rule: a custom object representing a failure type (ex. [redacted] crash, CRC does not match)

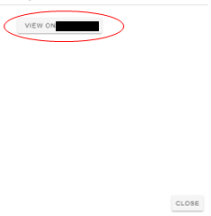
- -ruleID
- -name
- -description
- -date
- -lowPriority
- -symptoms

- LowPriority (true/false) allows you to prioritize rules. If multiple rules match a test failure, the rules with true LowPriority will be ignored. Rules with false LowPriority should be the most specific and exact match. Rules with true LowPriority may be symptoms rather than root test failure causes.
- A symptom contains a logfile and any associated search string(s) with that log file. Rules can have multiple symptoms.

Rule Manager Website (3)

- View rules in the database and edit, delete, get info on, and test them, along with creating new rules all from the View and Manage Rules Page (notice you can create and test rules from either this page or the taskbar).

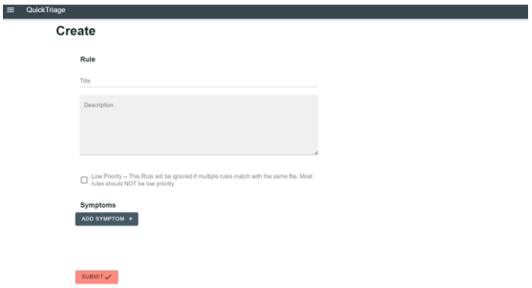




When viewing rule info, click 'VIEW ON [redacted]' to be taken to a summary page with how many instances of this rule have been found (Test Source 1 only).

Rule Manager Website (4)

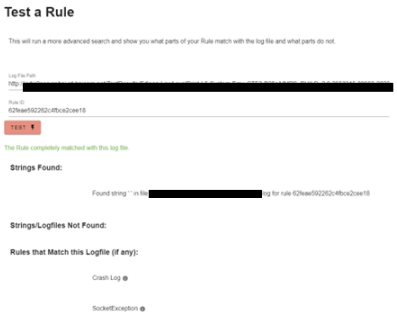
- Rules need to be as accurate and user-friendly as possible for the tool to work well.
  - Rules should be <u>unique.</u>
  - Rules should be specific (general rules should be lowPriority)
  - Rules should be tested to assure they work as expected.



Using the 'ADD SYMPTOM +' button, you can add multiple symptoms to one rule. You can also add multiple search strings to one symptom.

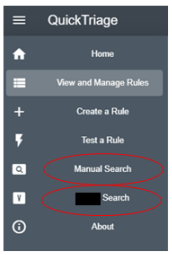
Rule Manager Website (5)

- Rules should be tested upon creation to make sure they function as expected.
- Rules should also be tested if they are suspected not to be working correctly.



Rule Manager Website (5)

- 'Manual Search' is used when you want to see what rule will match with a .zip.



- The '[redacted] Search' tab is only used in the Test Source 1 Process in section 3

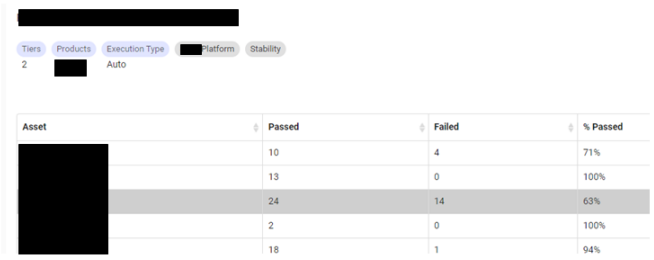
2. Test Source 1 Process (1)

- There is an automated pipeline for the Test Source 1. When Test Source 1 tests fail, the fail logs are sent through the QTT and displayed on the results website.
- To find what rules need to be created and added, you must go to the results website and see what failed tests aren't matching with a rule.

Test Source 1 Process (2)

- From the results website home page, navigate to Tests → [redacted] → <[redacted]> → <[redacted]> → click on a 'Failed' number (ex. 14)

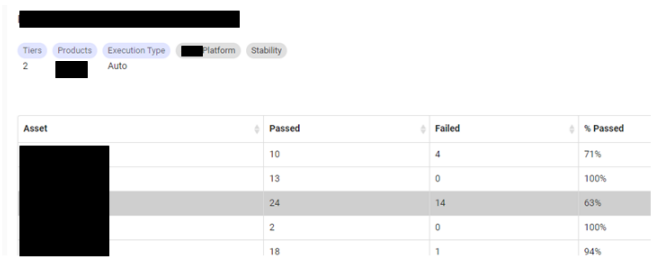
Results website link here:



Test Source 1 Process (2)

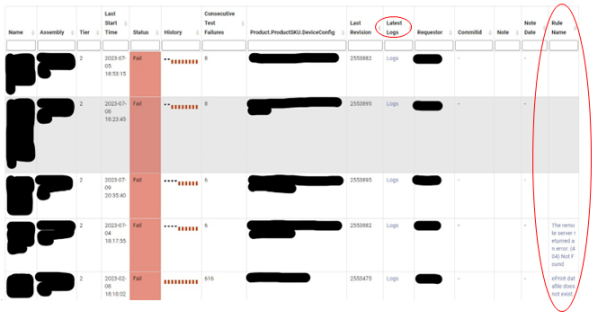
- From the results website home page, navigate to Tests → [redacted] → <[redacted]> → <[redacted]> → click on a 'Failed' number (ex. 14)

Results website link here:



Test Source Process (3)

- The far-right column contains the matching rule for the last test run. Clicking on it will bring you to the Rule Manager Website and show you the rule's info.
- For tests with no matched rule, download the fail logs using the Latest Logs → 'Logs' link to manually analyze for a new failure.
- Once found, create a new rule on the Rule Manager Website to represent that new failure.

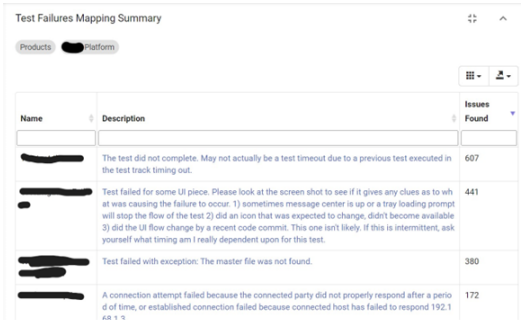


!

It is important to note that the new rule you create won't show here as matched until the test runs and fails again.

Test Source 1 Process (4)

- If you'd like, you can view an aggregation of the rules and how many times they have been found in test failures on the results website. This allows issue prioritization!
- Navigate to Tests → Test Failures Mapping



Test Source 1 Process (5)

Last Changes!/Next…

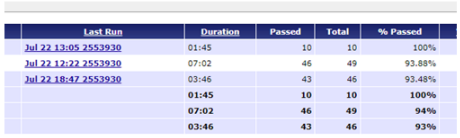
- As a note, we have added retroactive rule searching, meaning that every 12 hours or so, the QTT will search the rules database against all the recent test fail tips.
- This means that you should see the results of adding a new rule within 12 hours instead of waiting for the test to run and fail again.

3. Test Source 2 Process (1)

- You'll need the location of a triage bundle, which you can get from the Test Source 2 website
  - Website link here
- Filter on product, branch, and test pillar

*This is only one way to get the triage bundle location. Any other methods are okay as well.

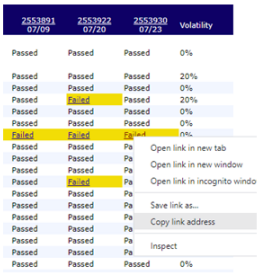
- Click on the linked "Last Run" you wish to triage



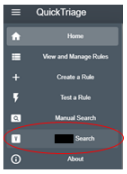
Last Run	Duration	Passed	Total	% Passed
Jul 22 13:05 2553930	01:45	10	10	100%
Jul 22 12:22 2553930	07:02	46	49	93.88%
Jul 22 18:47 2553930	03:46	43	46	93.48%
	01:45	**10**	**10**	**100%**
	07:02	**46**	**49**	**94%**
	03:46	**43**	**46**	**93%**

Test Source 2 Process (2)

- Go to Test History – Job
- Right click, "Copy link address" on a failure on the recent test run



Test Source 2 Process (3)



- On the Rule Manager website, navigate to the '[redacted]Search' page.
  - [Website link here:]
- In the text box, paste the link address you copied previously, but remove the last level of the path to reach the parent folder of the .zip.

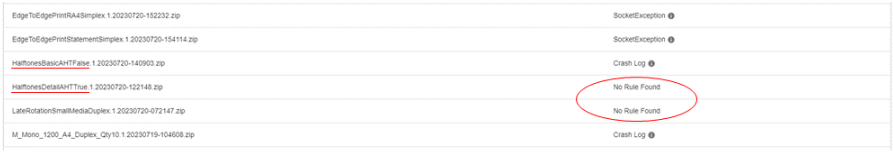
- Click "Search". This runs each .zip in the folder against the database of rules. Depending on the number of .zips, this could take some time. Look for the loading icon in the top right.

Test Source 2 Process (4)

- The resulting table shows the test name and the rule that matched.
- You can press the ⓘ icon to see the details of a matched rule.

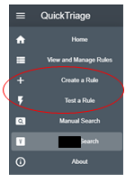
Test Source 2 Process (5)

- Find a test that did not match with a rule (“No Rule Found”). Go back to the test source 2 website and search for the test (excluding the date and run information, i.e. the red highlighted part) to download the fail logs of that test. Manually find this new failure so that you can add a new rule in the database to catch it.



Test Source 2 Process (6)

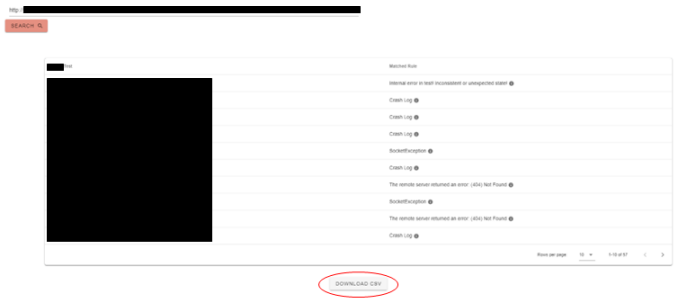
- Create a new rule for your failure. You can test the rule with the test failure .zip you found it from to make sure it successfully matches.



- Now that the new rule is in the database, try running the Test Source 2 search again. Not only should that failed test now match with your new rule, but other failed tests may also match with your new rule.

Test Source 2 Process (7)

- Continue this process until all test failures in the folder used as input match with a rule.
- Once you're done, you can download a csv to your machine that contains the data you see in the table.



Test Source 2 Process (8)

Last Changes! Next…

- We added:
  - Sort by 'Matched Rule' on the results table
  - Links on each .zip in the results table to make it easier to find the fail logs

Questions?

Contact [redacted]