Automated Error Log Resolution: A Case Study

Mark Fuller
Baylor University
Waco, Texas, USA
Mark_Fuller1@baylor.edu

Elizabeth Brighton
Baylor University
Waco, Texas, USA
Elizabeth_Brighton@Baylor.edu

Micah Schiewe
Baylor University
Waco, Texas, USA
Micah_Schiewe1@baylor.edu

Dipta Das
Baylor University
Waco, Texas, USA
dipta_das1@baylor.edu

Tomas Cerny
Baylor University
Waco, Texas, USA
Tomas_Cerny@baylor.edu

Pavel Tisnovsky
Red Hat
Brno, Czechia
ptisnovs@redhat.com

ABSTRACT

Debugging and error resolution has become increasingly time-consuming and difficult for all domains of software development. Error logs have become very important when it comes to debugging and error resolution. To remedy the problems presented in the logs, typically, a search on online forums would shed light on the solution. We present a novel approach to utilizing these logs in conjunction with external Question and Answer forums to compute and expedite resolution by suggesting a solution to runtime errors. Since log format is non-standard and use cases can vary widely, our architecture allows for extreme customization for the intended ecosystem as well as a great degree of fine-tuning. We evaluated our solution in a case study and made our implementation open-source for the community.

CCS CONCEPTS

• **Software and its engineering** → **Data flow architectures**; • **Computer systems organization** → *Reliability*; *Availability*; *Maintainability and maintenance.*

KEYWORDS

Defect Resolution, Error Logs, Log Analysis, StackOverflow

ACM Reference Format:
Mark Fuller, Elizabeth Brighton, Micah Schiewe, Dipta Das, Tomas Cerny, and Pavel Tisnovsky. 2020. Automated Error Log Resolution: A Case Study. In *Proceedings of ACM SAC (SAC'21).* ACM, New York, NY, USA, Article 4, 7 pages. https://doi.org/xx.xxx/xxx_x

1 INTRODUCTION

While much research has been done on the topic of predicting errors from log files [7], monitoring errors of distributed services [23], and presenting Question and Answer (Q+A) websites in the assistance of developers [13, 14]. Very little has been done to design and implement a system to suggest solutions from Q+A forums to both compile-time and runtime errors based on error logs [15].

SAC'21, March 22-26, 2021, Gwangju, Korea

ACM ISBN 978-1-4503-6843-8/20/10.
https://doi.org/xx.xxx/xxx_x

In this study, we suggest a new architecture for connecting these two domains of research. Based on our research, only one other study has suggested architecture for connecting these two domains before parsing the errors for information and using that data to suggest solutions to developers and software management teams [15]. However, this paper had some shortcomings, mainly the system's time to suggest fixes for errors. One of our architecture's main improvements is its use of a dynamic corpus, stored in a database, which minimizes queries to external sites and speeds up the matching process. Our implementation also improves on its scope; our method can be run on any log files, while the previous research is dependent on a specific IDE console. Our implementation also allows for more flexibility in the matching of error logs to Q+A sites.

In our design of such a system, we suggest three components: first, a log parser that converts some system's output into a structured format. This part of the implementation depends greatly on the use case and therefore is a non-essential detail in the implementation, however still an important component of the architecture. Second, a scraper that compiles a database of possible errors that could occur depending on the use case. Third and finally, a matcher which associates those structured logs with entries in the database. The use of a database is the main improvement in the work of Rahman et al. [15]. Pre-computing and storing information is a common software development solution, mainly used when speed is an important analytic. This design is easily extensible and customizable, depending on the use case.

This paper is organized as follows. Section 2 discusses related work. We describe our approach in Section 3 and implementation in Section 4. A case study is reported in Section 5. The last section concludes the paper with current limitations and future plans.

2 RELATED WORKS

Parsing Logs. Computer systems can fail for many different reasons, bugs, invalid input, administrator errors, etc. The parsing of these logs has thus become a hot topic for research. Since log format and content are highly subject to the system and are non-standard, log parsers must also be highly custom. SherLog, a system that uses error logs to diagnose runtime errors, is one such parser [24]. This software is capable of deriving much information from the logs with a few benefits such as 1. not having to re-run the application for more context information. 2. No assumptions on log semantics, meaning that it can work on any log format. 3. The ability to infer

control and data information. 4. Easily salable, they tested their system on a system with over one million code lines. 5. Most importantly, it can "automatically generate log parsers," meaning that the tool's user can easily customize the application to work for their systems log file syntax [24].

Since there is no universally decided format for logs, parsing and structuring error logs are essential to their use in automated systems [11]. Depending on the logs' format, though, this can be an arduous task. Research into this topic has been done, and many libraries have been developed for such a task [8, 25]. Zhu et al. implemented a series of pre-developed tools that help analysts parse and format a wide array of logs types. Of the data sets tested, their series was able to accomplish greater than 90% accuracy on each set [25].

Large distributed systems are becoming commonplace. Thus, parsing the logs from these systems is an equally important, however largely difficult task. He et al. completed a comprehensive study on a large set of publicly available log parsers and found that these parsers could not handle large-scale logs from these systems. They, therefore, designed POP, a large-scale architecture for distributed log parsing.

Scraping Forums. Debugging a piece of software is an arduous task. The use of forums has been widely known to help alleviate this shortcoming of programming for some time. Q+A sites are an excellent source of solutions to problems frequently addressed. Extracting data from websites is a domain full of solutions to every problem. Libraries like BeautifulSoup and lxml have been around for some time for the only purpose of extracting data from websites. [18, 19].

Extracting data from websites for use in error log resolution has also been attempted on many occasions. Blueprint, a web search interface, was created as a plugin to the Adobe Flex Builder environment to allow for direct searching of Adobe help forums from the IDE [2]. More recently, Ponzanelli et al. developed the Eclipse plugins Seahawk and Prompter to achieve this same end. Seahawk integrates a mock StackOverflow and search-engine within the IDE for fast access to reduce context switching between the IDE and search engine [1, 13, 22]. Prompter is an IDE plugin that auto-completes API and library documentation based on Stack Overflow [14]. Both of these tools are incredibly useful to developers in streamlining the development of complex libraries or APIs.

Searching. Search engines are used to return a set of values from a database of values given a seed value to search for. The most successful search engine Google's Search Engine is used to search the world wide web for matching sites based on some token [3]. However, search engines are used in other fields. For example, Lin et al. has developed pLink2, a successor to pLink1, which evaluates proteome-scale identification of cross-linked peptides [5].

Formal and complex search engines are not the only way to look for similar information in a database. Simple pattern matching also can suffice in small enough scenarios[10]. Fuzzy matching is a trendy way of quickly matching shorts strings together, and has seen use mining for data in web logs [9]. Regular expression has been used in this field to search for data within logs [6], and Levenshtein Distance has been used to analyze related search queries in logs [21]. Finally, recent studies have explored utilizing graph patterns for approximate string matching [4]; these graph solutions are generally limited by string size [20].

3 APPROACH AND UNIQUENESS

In this paper, we suggest an improvement in the architecture of the previously mentioned software Seahawk [13]. Seahawk's implementation for a solution to this problem can be optimized; every attempt made in an to solve a given error is made independently. We suggest that the pre-computing and caching of these attempts be made to speed up the suggestion process. By storing all suggestions in a database and not re-scraping and searching with each new error, the system will have to parse less HTML and make fewer network calls.

When an error needs to be resolved, a request is made to find that error in the database. Because of the system's architecture, how the error is searched for is highly customizable based on the language or scale of the implemented system. If the error has not been searched for before, then an actual network request is made, and data from Q+A sites are ranked, scraped, and stored in the database. This data is then returned to the user as suggestions for solutions to their error. By caching the data scraped from Q+A sites, the second network request is only made if the error is new to the system. Otherwise, the possible solutions are already stored in the database.

Apart from architectural improvement, we have also included several matching algorithms. Each of these algorithms has different levels of flexibility in terms of searching from the cache/database.

3.1 Parsing Logs

Before the error logs can be matched to the external solutions, the error logs must be put in a generic format that includes all the information that could be relevant in a matching algorithm. However, the main roadblock to a generic format is that program logs do not have a consistent format in all programming languages or projects. Therefore, the regex must be specially tailored to a specific log format in order to create the needed error log objects. Our parser is able to parse through a log file and find all error log entries, then analyze each log entry and store the following information:

Error Message. The most relevant piece of information from an error log is the error log's message. This generally indicates what the program was doing at the time of the error. We used the error message as our basis for matching with external sources.

Traceback And Nested Errors. The traceback and nested errors are useful for pinpointing the location of the error in the project code. We were able to find the relevant source code line if the error originated from the project's code.

External Error. We created the external error boolean to determine if the error came from our project or from an external service. This tells us if the error found is project-specific, which may be harder to match with external solutions.

Project File Structure. Using the project being analyzed as input, we created a basic file structure object, which included the variables and imports used in each file.

Source Code Line and File. Using the traceback, it is possible to find the file and specific source code line that the error occurred

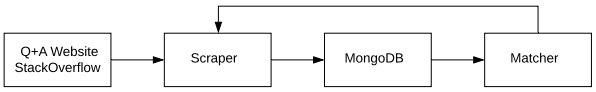

Figure 1: Scraping Architecture of Implementation

on. With this information, we can compare external solutions to the code that caused the error. We made use of the source code in our final algorithm through the error message weight system.

Error Message Character Weight Algorithm. In an error log message, it is important to identify the essential tokens. For example, matching 'the' to your error message from an external solution is much less important than matching 'language.' In order to give each word a proper level of importance, we created an algorithm that adds weight for each character in the error log message. The weight range is from zero to one, where zero means that the character is not important at all (for example, space), and one means that the character is very important. The first thing we decided to look at is variable names and values. Using the file structure object, we were able to determine if the variables in the source code line where the errors occurred were project-specific variables or imported variables. We were then able to assign different weights to those variables. The imported variables received a higher weight than the project-specific variables because project-specific variables are less likely to be found in external solutions data.

3.2 Web Scraping

Figure 1 gives us the architecture of implementation for scraping data from a website that can be matched with the error log being analyzed. In an ideal setting, all of the relevant external solutions would have already been added to the database. A database is used because it is much faster than runtime searches and bypasses websites access limits. However, in reality, new errors will occur that will have no relevant matches in a database, and more data must be scraped from the external Q+A website. To account for the constant possibility of new errors, we scrape data in two different ways. First, we scraped common error data into our database to create an initial data set for matching. It was followed by creating a system to automatically scrape data relevant to a log error if the log error did not match with any external solution already in the database. These two methods are described in detail below.

Initial Data Web Scraper. We created an initial web scraper that inputs external solution data into our database based on the programming language specified, it currently supports Python, C#, and Java. Once the scraper is started, it creates two threads that work together to produce data until the scraper is stopped. The first thread complies with a list of solution URLs for the programming language selected and places them inside a queue. As the first thread places each URL in the queue, the second thread dispatches a thread for each URL to begin the page scraping process. The data scraped is then put in the database.

Specific Data Web Scraper. While testing our matching algorithms, we discovered that one of the main reasons they failed is that we did not have an external solution in our database relevant to the error log being analyzed. Because of this, we created a

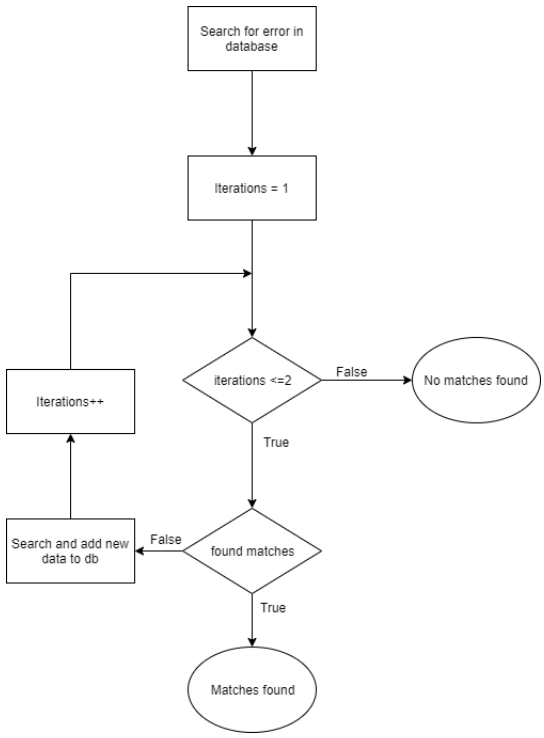

Figure 2: Flow of process on supplementation of database

scraping method that allowed us to scrape data that was actually relevant to the error log after the matching algorithm was run with no success. The matching algorithm would then be re-run with the newly scraped data to confirm our algorithm's ability to match the newly scraped data with the error log. The implementation of this scraper follows our initial data web scraper. Figure 2 outlines the flow of the process of the supplementation of data. We first query the external website is used for the ten most relevant solutions. We use the specific error log message as the search query topic to find the most relevant solutions. We then take the top ten solutions' URLs and scrape the data from those solutions directly into our database. Once all the data has been scraped, we re-run the matching algorithm to ensure our algorithm works correctly with the relevant data scraped and to find a relevant solution.

3.3 Matching Logs to Scraped Content

In order to tailor the matching algorithms outlined in the background section to match log file errors, we iteratively developed approximate string matching algorithms that could match error logs to external solutions scraped data. We created five algorithms, each building on the previous algorithm.

Fuzzy Title Matching Algorithm. In this approach, we used Fuzzy [12] string matching, which is an approximate string matching algorithm that uses the Levenshtein distance algorithm to determine how similar words or phrases are. Using the Fuzzy string matching, we were able to compare the message from the error log to the external solution's title. If the error log message and the external solution title were an 85% match, we considered the

external data to be applicable to the error log and returned that data to the user. This algorithm is effective when the title and error log message are almost exact but cannot find the external solutions that apply to the error log but have an unrelated title.

Basic Text Matching Algorithm. To solve the problem of off-topic titles, we implemented the second algorithm. This algorithm compares the external solution's title to the error log message as before; however, it goes a step forward and analyzes the text and code found in the external solutions answer. Because Fuzzy substring matching is only applicable for smaller strings, this matching algorithm was designed to go through the text and code using exact string matching. While exact string matching can be useful in many cases, it is not practical in log analysis because of the multitude of programming variable names, variable values, and method names specific to the program that are often included in the log messages. Our next algorithm improves upon the issue of specific variable values.

Advanced Text Matching Algorithm. This algorithm builds on the previous algorithms outlined while adding a specific feature based on the structure of the logs we analyzed. To get the most applicable solutions for our error log, we kept the exact matching as the preferred matching method. However, if no matches were found using the Basic Text Matching, we changed every value found in both the error log message and the external solution data to a generic value. We then compared the data again, using that generic value to find different results only by variable values. This allowed us to find the more generic solutions to the error logs received. However, if any exact matches were present, only the exact matches were returned, and not any of the generic solutions found. We were able to remedy this using a score-based matching system.

Score Text Matching Algorithm. In this algorithm, the system for matching remained the same. However, the value of each match was changed. Instead of just returning every potential match, we created a system that keeps track of how similar each external solution was to the error log. This was done by assigning different scores to different matches made. For example, if the error log message exactly matched the external solution title, that external solution would receive a higher score than if the error log message was a fuzzy match to the external solution title or if it was found in the external solution's text. A scale system was created for each of the matching methods outlined above, and the highest scored external solution was returned as the most relevant solution, followed by the next nine applicable solutions. With this, we were able to incorporate the Basic and Advanced Text Matching Algorithms together.

Weight Text Matching Algorithm. To improve our matching even more, we went back to the issue of variable names and values. We solved varying variable values in the Advanced Text Matching Algorithm; however, program-specific variable names could still impede our matching. To solve this issue, we came up with the Weight Text Matching Algorithm. This algorithm requires some preprocessing to identify all of the program-specific variable names. Once the variable names were found, we identified the variable names used in the source code line where the error occurred to assign less value to those words inside the error log message. Once each word in the error log message had a specific value assigned to it, we used the Levenshtein distance algorithm to determine the similarity of the more important words to the external solution's title, text, and code. This algorithm includes the most important data into the matching process, because of this, it had the highest correct matching rate in our testing.

4 IMPLEMENTATION

Parser. In our implementation, we created a generic interface for the parser. This interface can be extended for parsing specific log format. The parser tokenizes the log messages, including tracebacks, nested errors, and source code locations. Regardless of the raw log format, the parser produces a generic structure for error logs. This log structure is then able to be used as input into the matcher.

Scraper. Utilizing the Stack Overflow API along with a Python Flask framework, we developed an interface between Stack Overflow and a MongoDB database. The public Stack Overflow API allowed for an efficient and rather fast implementation. The scraper can run both in daemon mode or standalone mode. It keeps running in daemon mode and continues to scrape contents from StackOverflow for a given programming language. Using multiple threads, we are able to scrape many hundreds of forum posts per minute. These forum posts were then tokenized and stored into a MongoDB instance. In standalone mode, it takes a StackOverflow URL as an input and scrapes data for that specific URL.

Matcher After parsing the error logs, scraping the data from Stack Overflow, and placing that data into our MongoDB, we then implemented our matching algorithms using the Java Spring framework. All algorithms were able to match the error log messages with the Stack Overflow titles. The Basic, Advanced, and Score Text Matching Algorithms were able to identify applicable Stack Overflow solutions through the use of the Stack Overflow scraped text and code sections as well. These three algorithms all use the JavaWuzzy Fuzzy string matching algorithm based on the FuzzyWuzzy python algorithm as their basis for string matching [12]. In the Weight Text Matching algorithm, we used the Levenshtein distance formula to implement our own version of the JavaWuzzy algorithm. Our Weight Text Matching Algorithm correctly pulled all internal and external variable information from the programs given and assign relevant weights to each word in the error log message, making the overall matching much more accurate. Every successive algorithm correctly resolved the issues from the previous algorithm that it was created to solve.

Our implementation is open-sourced and available as a reference for future improvements [1].

5 CASE STUDY

In our case study, we used logs from the Red Hat Insights Results Aggregator project. This particular project gives insight on OpenShift Container Platform (OCP) data within an OpenShift Cluster Manager [17]. These data contain information about clusters status, especially health, security, and performance [17]. The architecture of the project is shown in Fig. 3.

We collected row logs in static files from the cxx-data-pipeline and the aggregator. These raw log files are then inputted into our system. The parser processes the log files and filters the error logs

[1] github.com/cloudhubs/log-errors

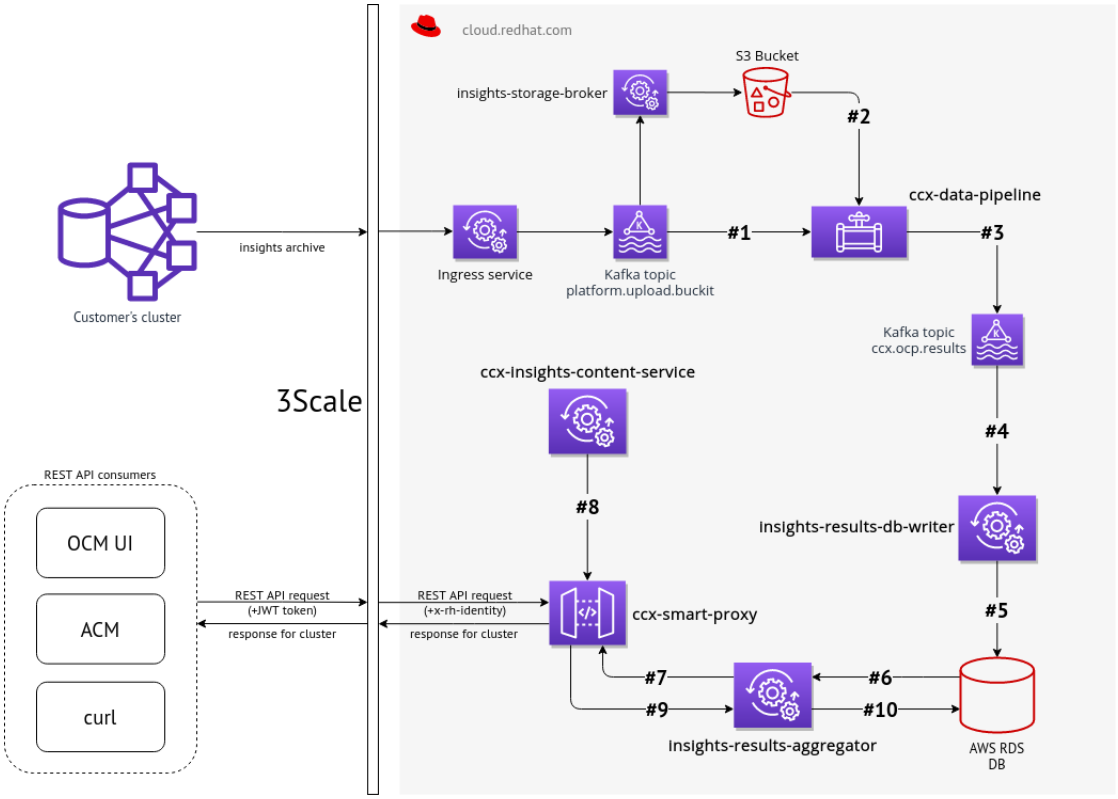

Figure 3: Architecture of the RedHat Insights Results Aggregator [16]

into a structured form for further analysis. Listing 1 shows a sample block of the log file. We implemented the parser interface for this specific log format. The Listing 2 shows the parsed error produced by the parser. The parsed errors are then fed into the matcher. The matcher first tries to utilize the cache for each error, which is a MongoDB collection of StackOverflow contents. If no relevant results are found in the cache, it performs a generic web search to find a list of StackOverflow URLs and scrapes the content from the URLs. The scraper stores each scraped page into the cache. Once the scraper finishes scraping contents and storing them into the cache, the matcher performs the matching again from the cache.

We ran two separate benchmark tests - the first one examines the overall system's performance and the second one examines the performance of individual matching algorithms. For the first benchmark test, we cleared the cache beforehand to make sure the scraper was invoked. We ran the experiment five times for five different similar-sized log files with a fixed matching algorithm. Table 1 shows the average execution times of each module over five trials. The parser module took three seconds on average while the scraper and matcher module took 87 and 12 seconds respectively. The runtime of the matcher module involves both performing web search and execution of the matching algorithm. The scraper module consumed the highest execution time since it involves API calls and intensive HTML parsing. Also, it took a slightly different execution time in each trial. This could be associated with the network latency of the StackOverflow API calls.

Table 1: Average execution times of different modules

Module	Time (seconds)
Parser	3
Scraper	87
Matcher	12
Total	102

In the second benchmark test, we passed ten preformatted errors directly to the matcher instead of going through the parser and scraper since the goal was to evaluate matching algorithms. For each of those errors, we manually searched StackOverflow and prepared a list of the most relevant solutions. Also, we populated the cache beforehand with those most relevant pages along with some moderately relevant and completely irrelevant pages. Then we ran the matcher with different matching algorithms and counted how many of the returned results matched our most relevant solutions, how many are moderately relevant, and how they may be irrelevant. Table 2 shows the counts along with the execution times of each matching algorithm. Note that the results represent an average outcome for a single error event over ten trials.

From Table 2, we can see that Fuzzy Title Matching and Basic Text Matching did not return any irrelevant results; these two algorithms involve the most strict matching. Fuzzy Title Matching returned fewer results compared to others since it only considered the title while matching. In total, Fuzzy Title Matching returned

Listing 1: Sample block of raw log file

```
...
2020-06-03 12:42:52,653 WARNING - dr.py:974 - Traceback (most recent call last):
File "/opt/app-root/lib/python3.6/site-packages/insights/core/dr.py",
line 962, in run result = DELEGATES[component].process(broker)
File "/opt/app-root/lib/python3.6/site-packages/insights/core/dr.py",
line 681, in process return self.invoke(broker)
File "/opt/app-root/lib/python3.6/site-packages/insights/core/plugins.py",
line 64, in invoke return super(PluginType, self).invoke(broker)
File "/opt/app-root/lib/python3.6/site-packages/insights/core/dr.py",
line 661, in invoke return self.component(*args)
File "/opt/app-root/lib/python3.6/site-packages/ccx_ocp_core/models/nodes.py",
line 108, in Nodes int(node.q.status.capacity.memory.value.split("Ki")[0])
AttributeError: 'NoneType' object has no attribute 'split'
...
```

Listing 2: Parsed error log

```
{
    "source": "/logs/ccx_data_pipeline_1_anonymized.log",
    "lineNumber": 2666,
    "sourceCodeLine": "result = DELEGATES[component].process(broker)",
    "sourceCodeFile": "dr.py",
    "isExternal": false,
    "errorMessage": "AttributeError: 'NoneType' object has no attribute 'split'",
    "traceBacks": [
      "File '/opt/app-root/lib/python3.6/site-packages/insights/core/dr.py',
      line 962, in run result = DELEGATES[component].process(broker)",
      "File '/opt/app-root/lib/python3.6/site-packages/insights/core/dr.py',
      line 681, in process return self.invoke(broker)",
      "File '/opt/app-root/lib/python3.6/site-packages/insights/core/plugins.py',
      line 64, in invoke return super(PluginType, self).invoke(broker)",
      "File '/opt/app-root/lib/python3.6/site-packages/insights/core/dr.py',
      line 661, in invoke return self.component(*args)",
      "File '/opt/app-root/lib/python3.6/site-packages/ccx_ocp_core/models/nodes.py',
      line 108, in Nodes int(node.q.status.capacity.memory.value.split('Ki')[0])",
      "AttributeError: 'NoneType' object has no attribute 'split' "
    ],
    "nestedError": null
}
```

Table 2: Comparison of different matching algorithms

Algorithm	Total	Most Relevant	Moderately Relevant	Irrelevant	Time (seconds)
Fuzzy Title Matching	3	2	1	0	1
Basic Text Matching	5	4	1	0	2
Advanced Text Matching	8	4	3	1	3
Score Text Matching	8	4	3	1	3.5
Weight Text Matching	9	5	3	1	4

three results (two most relevant and one moderately relevant) while Basic Text Matching returned five results (four most relevant and one moderately relevant). Advanced Text Matching is more flexible compared to the basic one; however, it returned one irrelevant result along with the four most relevant and three moderately relevant results. Score Text Matching is similar to Advanced Text Matching with the additional benefit of sorted results based on scores. Weight Text Matching is the most flexible one while matching the code lines since it uses program-specific weights on variable names. In general, it performed better than the previous two in finding the most relevant solutions. It returned five most relevant and three moderately relevant results. However, due to flexibility it also returned one irrelevant result.

All of these matching algorithms operated on the same sized cache for a fair comparison. On average, Fuzzy Title Matching took one second and Basic Text Matching took seconds to execute. Advanced Text Matching and Score Text Matching took 3 and 3.5 seconds respectively. Score Text Matching involves additional sorting compared to Advanced Text Matching. Weight Text Matching took the highest execution time among all, consuming four seconds on average. Overall, the execution time of the algorithms increased gradually with the matching flexibility.

From the case study, we can see that automation can be applied in resolving error logs and the flexibility of the solutions can be adjusted with the choice of matching algorithms.

5.1 Threats to Validity

The runtime of the parser relies on the log format. Complex log format will take more time to parse. The runtime of the matcher depends on the amount of scraped data. It will grow linearly as the size of the cache increases.

In the second benchmark test, we manually identified the most relevant and moderately relevant solutions. It could be prone to human errors. Also, we assumed the cache already contains those solutions. In practice, this will require a large size of scraped content.

Since the matching algorithms were compared against the selections of humans, bias must be considered as a downfall of this technique. Depending on the selections of the baseline result set, the success of each of the demonstrated algorithms could change.

6 CONCLUSION

Error resolution is paramount to the development and maintenance of software. There has been little research done to combine the log analysis fields suggesting Q+A sites solutions. Our paper contributions are summarized as follows: (1) we proposed an architecture for the automatic analysis of runtime and compile-time error logs (2) we introduced several matching algorithms with different levels of flexibility (3) we performed a case study to show that the proposed architecture can automatically suggest solutions with little to no human interaction in the process. (4) We described a benchmark test in the case study to compare the matching algorithms.

We demonstrated on our prototype implementation that it is possible to automate error log resolution, and thus improve the quality assurance process, reduce the burden and turnaround related to bug resolution.

6.1 Shortcomings and Further Work

This architecture is heavily dependant on a central repository of pre-existing Q+A forum data. Therefore for new or uncommon errors, a reasonable recommended solution is unlikely. This can be improved by increasing the number of Q+A forums that contribute to the central repository. The individual improvement of the modular algorithms that are used in the architecture, such as the matching algorithm or the parser, will always need to be refined further. We will integrate our system into the software development lifecycle and package as an IDE plugin to enhance productivity.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1854049 and a grant from Red Hat Research https://research.redhat.com.

REFERENCES

[1] A. Bacchelli, L. Ponzanelli, and M. Lanza. 2012. Harnessing Stack Overflow for the IDE. In *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*. 26–30.

[2] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. 2010. Example-Centric Programming: Integrating Web Search into the Development Environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. Association for Computing Machinery, New York, NY, USA, 513–522. https://doi.org/10.1145/1753326.1753402

[3] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks* 30 (1998), 107–117. http://www-db.stanford.edu/~backrub/google.html

[4] Xiaoshuang Chen. 2020. Simulation-Based Approximate Graph Pattern Matching. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2825–2827. https://doi.org/10.1145/3318464.3384401

[5] Zhen-Lin Chen, Jia-Ming Meng, Yong Cao, Ji-Li Yin, Run-Qian Fang, Sheng-Bo Fan, Chao Liu, Wen-Feng Zeng, Yue-He Ding, Dan Tan, Long Wu, Wen-Jing Zhou, Hao Chi, Rui-Xiang Sun, Meng-Qiu Dong, and Si-Min He. 2019. A high-speed search engine pLink 2 with systematic evaluation for proteome-scale identification of cross-linked peptides. *Nature Communications* 10, 1 (July 2019). https://doi.org/10.1038/s41467-019-11337-z

[6] Minos N Garofalakis, Rajeev Rastogi, and Kyuseok Shim. 1999. SPIRIT: Sequential pattern mining with regular expression constraints. In *VLDB*, Vol. 99. 7–10.

[7] Amir Globerson, Terry Y. Koo, Xavier Carreras, and Michael Collins. 2007. Exponentiated Gradient Algorithms for Log-Linear Structured Prediction. In *Proceedings of the 24th International Conference on Machine Learning (ICML '07)*. Association for Computing Machinery, New York, NY, USA, 305–312. https://doi.org/10.1145/1273496.1273535

[8] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu. 2016. An Evaluation Study on Log Parsing and Its Use in Log Mining. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 654–661.

[9] Karuna P. Joshi, Anupam Joshi, Yelena Yesha, and Raghu Krishnapuram. 1999. Warehousing and Mining Web Logs. In *Proceedings of the 2nd International Workshop on Web Information and Data Management (WIDM '99)*. Association for Computing Machinery, New York, NY, USA, 63–68. https://doi.org/10.1145/319759.319792

[10] Nick Koudas, Amit Marathe, and Divesh Srivastava. 2004. Flexible String Matching against Large Databases in Practice. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30 (VLDB '04)*. VLDB Endowment, 1078–1086.

[11] S. Messaoudi, A. Panichella, D. Bianculli, L. Briand, and R. Sasnauskas. 2018. A Search-Based Approach for Accurate Identification of Log Message Formats. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. 167–16710.

[12] P.W.D. Panayiotis. 2020. fuzzywuzzy. https://github.com/xdrop/fuzzywuzzy. (2020).

[13] L. Ponzanelli, A. Bacchelli, and M. Lanza. 2013. Seahawk: Stack Overflow in the IDE. In *2013 35th International Conference on Software Engineering (ICSE)*. 1295–1298.

[14] L. Ponzanelli, G. Bavota, M. D. Penta, R. Oliveto, and M. Lanza. 2014. Prompter: A Self-Confident Recommender System. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 577–580.

[15] M. M. Rahman, S. Yeasmin, and C. K. Roy. 2014. Towards a context-aware IDE-based meta search engine for recommendation about programming errors and exceptions. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. 194–203.

[16] Red Hat. 2020. Customer facing services architecture. https://github.com/RedHatInsights/insights-results-aggregator/blob/master/docs/assets/customer-facing-services-architecture.png. (2020).

[17] Red Hat. 2020. Insights Results Aggregator. https://github.com/RedHatInsights/insights-results-aggregator. (2020).

[18] Leonard Richardson. 2020. Beautiful Soup. (2020). https://www.crummy.com/software/BeautifulSoup/

[19] Stephan Richter. 2020. XML and HTML with Python. (May 2020). https://lxml.de/

[20] Kaspar Riesen, Miquel Ferrer, and Horst Bunke. 2020. Approximate Graph Edit Distance in Quadratic Time. *IEEE/ACM Trans. Comput. Biol. Bioinformatics* 17, 2 (March 2020), 483–494. https://doi.org/10.1109/TCBB.2015.2478463

[21] Xiaodong Shi and Christopher C. Yang. 2006. Mining Related Queries from Search Engine Query Logs. In *Proceedings of the 15th International Conference on World Wide Web (WWW '06)*. Association for Computing Machinery, New York, NY, USA, 943–944. https://doi.org/10.1145/1135777.1135956

[22] Andrew Walker and Tomas Cerny. 2020. On Cloud Computing Infrastructure for Existing Code-Clone Detection Algorithms. *SIGAPP Appl. Comput. Rev.* 20, 1 (April 2020), 5–14. https://doi.org/10.1145/3392350.3392351

[23] Xiao Yu, Pallavi Joshi, Jianwu Xu, Guoliang Jin, Hui Zhang, and Guofei Jiang. 2016. CloudSeer: Workflow Monitoring of Cloud Infrastructures via Interleaved Logs. *SIGARCH Comput. Archit. News* 44, 2 (March 2016), 489–502. https://doi.org/10.1145/2980024.2872407

[24] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. SherLog: Error Diagnosis by Connecting Clues from Run-Time Logs. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1736020.1736038

[25] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R Lyu. 2019. Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 121–130.