VISUAL LOGGING FRAMEWORK USING ELK STACK

by

RAVI NISHANT

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

Dec 2017

Copyright © by Ravi Nishant, 2017

All Rights Reserved

Acknowledgements

I would like to thank Professor David Levine for the continued support during my thesis design and implementation. I appreciate the time he allocated throughout the last two semesters. I would also thank Professor Gergely Zaruba and Professor Vassilis Athitsos for being my committee members and providing valuable suggestions.

I thank all friends namely Gaurav, Amith, Adhavann and Anil, for support in discussions and implementation. Lastly, I would like to thank my family for the continuous support during my journey in UTA.

Nov 16th, 2017

Abstract

VISUAL LOGGING FRAMEWORK USING ELK STACK

Ravi Nishant, MS

The University of Texas at Arlington, 2017

Supervising Professor: David Levine

Logging is the process of storing information for future reference and audit purposes. In software applications, logging plays a very critical role as a development utility and ensures code quality. It acts as an enabler for developers and support professionals by providing them capability to see application's functionality and understand any issues with it. Data logging has a widespread use in scientific experiments and analytical systems. Major systems which heavily uses data logging are weather reporting services, digital advertisement, search engines, space exploration systems to name a few. Although, data logging increases the productivity and efficiency of a software system, the logging process itself needs to be an efficient one. A logging system should be highly reliable, should support easy scalability and must maintain high availability. Logging infrastructure should also be completely decoupled with the parent system to ensure non-blocking operation. Finally, it should be secure enough to meet the needs of businesses and government as required.

In the age of big data, logging systems themselves are a huge challenge for companies, so much so that few corporations have teams dedicated to providing data services such as data storage, analytics and security. They use logging frameworks of varying capabilities and as per their need. However, most of the logging utilities tend to be partially efficient or they face critical challenges like scalability, high throughput and performance. At present, we have few logging frameworks providing analytical capabilities along with traditional functionalities of data formatting and storage. As part of this thesis work, we come up with a logging framework which seeks to solve both functional challenges and problems related to its efficiency and performance. The system demonstrated here combines best features of multiple utilities such as messaging brokers like Kafka, event publishing through SQS and data management and analytics using ELK stack. The system implementation also utilizes efficient design patterns to tackle nonfunctional challenges such as scalability, performance and throughput.

Contents

Acknowledgements iii

Abstract iv

List of Illustrations vii

List of Tables ix

Chapter 1 Introduction 10

1.1 Motivation Behind the Thesis. 10

1.2 Challenges 15

Chapter 2 Related Work 17

2.1 Massive distributed and parallel log analysis 17

Chapter 3 Existing Problems Requirements 23

3.1 Unstructured Schema: 23

3.2 Strongly Coupled Storage: 24

3.3 Lack of Visualization Capabilities: 26

3.4 Handling Data Loss 27

3.5 Real Time Analytical Capabilities 28

3.6 Performance Scalability: 29

Chapter 4 Solution Design and Components 32

4.1 Base Logging Framework 32

4.2 Centralized Storage 34

4.3 Non Blocking Execution 36

4.4 Avoiding Large Number of Thread Spawns 37

4.5 Data Pipeline using Kafka 38

4.6 Elasticsearch DB 39

4.7 Kibana Dashboard 41

Chapter 5 Implementation Overview 42

Chapter 6 Technical Approach 51

6.1 Storing Logs in S3 Buckets: 51

6.1.1 AWS S3 Configuration 51

6.1.2 S3 Folder Structures 52

6.1.3 AWS SDK's S3 API 52

6.1.4 Maintaining Access Policies 53

6.2 Schema Structuring: 53

6.2.1. LogFormatter Class: 54

6.2.2. JSON / AVRO Data Format: 55

6.2.3 Backward Compatibility For PlainText Logs: 56

6.2.4 Resources Field 57

6.3 S3 Appender 57

6.3.1 AppenderSkeleton Class 58

6.3.2 Singleton Wrapper 60

6.3.3 Producer Consumer Pattern 61

6.3.4 Graceful Exiting of Logging Threads 61

Chapter 7 Experiments and Results 62

7.1 Structured Log Data 62

7.2 System Throughput Result 63

7.3 Demonstration of Debugging Efficiency 65

7.4 Visualization and Dashboards: 66

7.5 Backward Compatibility: 66

Chapter 8 Conclusion and Future work 68

References 70

Biographical Information 72

List of Illustrations

Figure 1-1 General Large-Scale Logging System [15] 11

Figure 1-2 Meaningful Information in form of Visualization 13

Figure 1-3 Log4J configuration for log appender 15

Figure 2-1 Distributed log analysis framework workflow [3] 19

Figure 2-2 Three-Level hash map [3] 21

Figure 2-3 No. of nodes by time [3] 21

Figure 3-1 Unstructured log message [17] 23

Figure 3-2 Storage Decoupling [18] 25

Figure 3-3 Sample Dashboard 26

Figure 3-4 Data Streaming [19] 28

Figure 4-1 Log4J Properties Configuration 33

Figure 5-1 Logging to S3 (Producing Logs) 44

Figure 5-2 Log Consumption and Visualization Pipeline 45

Figure 5-3 Design Class Diagram 47

Figure 6-1 S3 Configuration 52

Figure 6-2 S3 Bucket Policy 53

Figure 6-3 LogFormatter Wrapper 54

Figure 6-4 Sample XML 55

Figure 6-5 Sample JSON 56

Figure 6-6 Backward Compatibility 56

Figure 6-7 Append Method Implementation 59

Figure 6-8 Overriding Close Method 60

Figure 7-1 Structured Data Demonstration 62

Figure 7-2 Thread Count vs Execution Time 64

Figure 7-3 Message Size vs Execution Time 64

Figure 7-4 Log Stitching 65

Figure 7-5 Visualizations 66

Figure 7-6 Backward Compatibility Demonstration 67

List of Tables

Table 1 Throughput calculation of a messaging queue system [12] 31

Table 2 Test Cases for S3 Appender 49

Chapter 1

Introduction

1.1 Motivation Behind the Thesis

This thesis aims to cover various aspects of a data logs and logging process. Data forms the core component of logging which make managing them both important and challenging. This report covers various aspects of creating, managing and storing data logs. Other than being stored in a secure manner, logs should be served efficiently when required with high reliability. This is where data logging process comes into picture. Logging data should be secure, reliable, scalable and have high availability. We also cover challenges faced by existing logging systems and ways in which they can be improved in different aspects. Finally, this thesis demonstrates a proof of concept logging system which seeks to solve major problems existing logging application face.

Logging data is a critical process in any system development life cycle. From smaller mobile applications to large software running on mission critical systems, each one of them implement data logging. Log data are used by developers to analyze the functional problems in the application. Developers rely heavily on logs in development process and also during maintenance phase. In most cases, applications are deployed in places such as hospitals, airports, banks and other commercial sites where professional don’t have access or are not provided with debugging tools. In these cases, logs help to localize the problem and provide access to complete spectrum of problem’s root cause.

Logs also helps system admins and operators to oversee any network or computer problems in an organization. Any system error or network issues generate error logs which can help a system admin to locate the problem and take corrective actions.

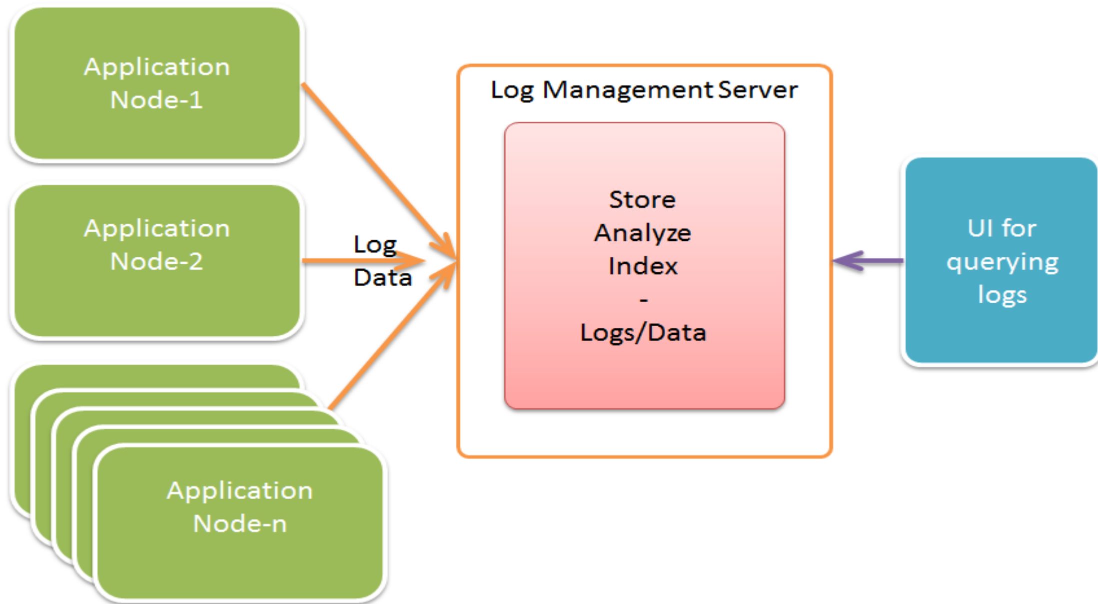

Figure 1-1 General Large-Scale Logging System [15]

Logs play a very important role in security domain. Large organizations have teams focused on IT security. Security logs helps deal with problems such as malware analysis, network traffic anomaly and unauthorized access monitoring. A fast-growing movement in IT called DevOps are also heavy users of logs. DevOps are generally a mix of developers and system administrators and therefore have specific use case for logging data. Finally, logs are used not just for troubleshooting issues and security. They are also used for forecasting weather, stock markets and consumer behavior using deep learning methods.

As a first step in search of an ideal logging system, it is important to characterize the important qualities it should possess. They can be broadly classified as follows:

1.1.1 Ease of Usability

Any system should be easy to use and same is true for logging systems too. An ideal logging system should be easily pluggable to parent application. A developer need not use his time and energy in configuring and maintaining a log system. To make it hassle free a log system should be self-sustainable.

1.1.2 Low Resource Usage

A log system almost always runs along with a main or parent application. It takes messages from parent application and forwards it to logging infrastructure. It is very important for it to not hinder with working of parent application. For instance, if we consider an online video streaming application, it is quite evident that the application will be a resource intensive one. In this case, the ideal supporting log system would be one which does not use much of resource for logging process and leave them available for parent application to perform business critical tasks.

1.1.3 Information from logs

A log is in essence, a pile of data generated by machine. It normally is huge in size and difficult to comprehend and analyze. It may not follow a standardized structure in many cases which further reduces its readiness. An ideal logging system should try to minimize these problems. Not being able to read and analyze data quickly defeats the purpose of using logging as an overhead to parent application. It also does not expose many critical information which developers can use to fix imminent bugs or an organization can use to make business development decisions. Analysis of logs can also boost team confidence by serving as an audit of how well the main application functions.

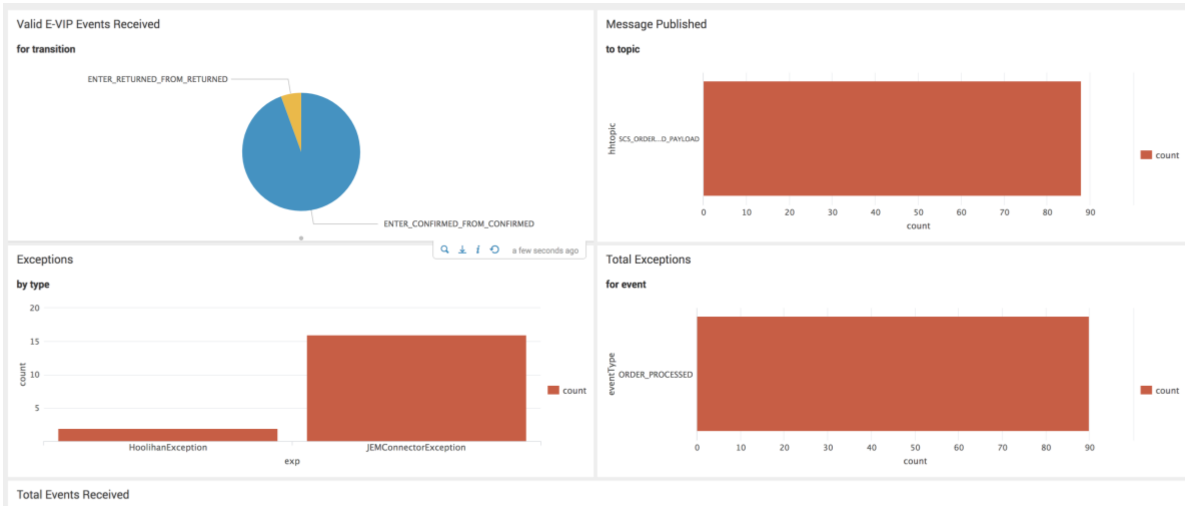

Figure 1-2 Meaningful Information in form of Visualization

1.1.4 Providing generalized solution

There are many logging frameworks available today. Most of them are used by large organizations and they serve the purpose well. However, there are only few frameworks which provide a generalized solution. Most of the times, logging frameworks needs to be tweaked for usage in different environments. They cannot adapt to different storage solution with ease. It is also quite difficult to adapt to a different data format with ease and with minimal configuration changes.

1.1.5 Ability to scale:

Software applications needs to scale and so is the case with logging frameworks. Logging systems in today's world often deals with large amount of data every day. Companies like Netflix and LinkedIn have reported over a million message points per second. Scaling logging infrastructure is so important that Kafka, the most resilient messaging system currently available was developed to solve the issue of logging big

data. Scaling a system also benefits with increased performance and lesser rework in refactoring efforts.

1.1.6 Security

Log data comprises of application logs as wells as user information. Application logs are critical information that no organization would want leaked. For example, an online video streaming company would not want to have its application logs leaked to competitors. This leak can provide a third-party access to their system architecture information. User information are another important data because they belong to customers. A person’s critical information such as bank account number, social security number, address if leaked can be used for identity fraud and financial fraud. A recent event of Equifax data breach is an example of how securing information is very critical for any software system.

1.2 Challenges

1.2.1 Varying Logging Requirements:

Logging has various use case in current software ecosystem. Its usage can vary widely from collecting information for storing purpose to analyzing user information for business development. Companies providing streaming services such as YouTube and Netflix use log information to enhance user experience depending on user's location around the world. Since there are many ways and requirement in which these systems need to fit in, it is very difficult to have a generic solution to logging problems. Often, they require extensive configuration for different environments and systems they are used with.

```
log4j.rootCategory=DEBUG, S3Appender
log4j.logger.com.amazonaws=ERROR
log4j.logger.org.apache.http.wire = ERROR
log4j.appender.S3Appender=org.uta.s3appender.S3Appender
log4j.appender.S3Appender.layout=org.apache.log4j.PatternLayout
log4j.appender.S3Appender.maxsize=1000000
log4j.appender.S3Appender.maxduration=1000
log4j.appender.S3Appender.microservice="TestMicroservice"
log4j.appender.S3Appender.threadsize=4
log4j.appender.S3Appender.fileextension=".log"
log4j.appender.S3Appender.s3bucketpath="test"
log4j.appender.S3Appender.s3accesskey="AKIAIBJOMJ5XPQP4VGCA"
log4j.appender.S3Appender.s3secretkey="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"
log4j.appender.S3Appender.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n

log4j.rootLogger = DEBUG, FILE
log4j.appender.FILE=org.apache.log4j.RollingFileAppender
log4j.appender.FILE.File=${log}/log.out
log4j.appender.FILE.ImmediateFlush=true
log4j.appender.FILE.Threshold=debug
log4j.appender.FILE.Append=true
log4j.appender.FILE.MaxFileSize=5KB
log4j.appender.FILE.MaxBackupIndex=2
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.conversionPattern=%m%n
```

Figure 1-3 Log4J configuration for log appender

1.2.2 Resource Intensive Operation

Almost all log data needs to be stored at some centralized location as it is not feasible to store large data in a single computer system. This is because logs grow in

size linearly with time and with scale of parent system. It is not difficult to imagine that larger the log data size, more time it will be needed to upload log file to a centralized storage. Not only size, but the number of log fragments also determine the performance of the logging system. Larger the number of log fragment files, lower will be the performance as cpu will be used inefficiently for creating and initializing new files on disk.

1.2.3 Backward Compatibility

Another big challenge is providing backward compatibility to existing client systems. This is important as not every system may want to upgrade to use the new logging framework. In such a case, the proposed logging system should not break any functionality of the parent application. Providing backward compatibility sometimes require lot of unnecessary junk code which clutters the system.

Chapter 2

Related Work

2.1 Massive distributed and parallel log analysis

2.1.1 Introduction

Security log analysis is necessary for detecting anomalies and intrusions. But with the great volume of log data, new frameworks and security, computing techniques are required. The evaluation of the massive distributed and parallel log analysis for organizational security demonstrates the efficiency of a cloud based framework in analysis of large scale logs.

2.1.2 Why log analysis?

Computing systems, networks and other processes have various rich data for security analysis. This data is used to detect deviations, errors and to reveal previous and current anomalies. Unexpected network actions are identified by the system administrators using the network log analysis. Unauthorized file creation is detected by file system log analysis.

2.1.3 Challenges faced and Issue to be addressed

The above-mentioned logs are large in number and they become a burden to corporations and organizations when they come from various security systems. Organizations face a big challenge when it comes to economically and efficiently store and analyze the enormously growing logs. The existing security log analysis solutions assume centralized computing and information storage which is not scalable. The current systems are based on distributed file system which minimizes performance of the system mainly when the environment is realized on Network File System with a single pool. This

paper addresses the question of how to analyze the huge number of logs in a timely manner.

2.1.4 Approaches followed

To speed up the analysis, the workload has to be distributed to multiple nodes and then harvest and sum up the results. Distributed log analysis model is considered to process a large number of logs in this paper. This model varies from MapReduce. MapReduce assumes a specific file system for information storage whereas this model elaborates a common cloud storage. The logs are created and stored in the cloud with the services being outsourced to the cloud. This framework supports counting, pattern matching of security analysis and correlation activities. Below are the design goals from this lightweight framework,

1. Usability- For streaming log and task support, ease in use.
2. Lightweight- Framework's minimal overhead.
3. Cloud Compatibility- Cloud computing benefits.
4. Scalability- Efficient performance with numerous cloud instances.

2.1.5 Model Design

The design comprises of a two-level master slave model, appending file feature, computation on demand cloud environment and Amazon EC2 and Simple Storage Service (S3). The two assumptions in the environment are that logs are created and stored in cloud and the deployment environment enables a complete interface to handle computation resources.

- Entities and Components: The basic unit of the distributed framework is called a node which is a physical or virtual machine. There are three types of nodes namely the master (nm), storage slave (nss) and computation slave node (ncs)

1. Master node is the control and command node. It provides interface to the user, programs log segmentation of each computation slave node and retrieves log data from storage slave nodes.
2. Computation slave nodes are required for parallel log analysis activities. Master nodes send tasks to these nodes and later the log segments are retrieved from storage nodes based on the tasks and then the logs are analyzed.
3. Storage slave nodes are used for logs storage. This node is used to read the latest log from the native file system and respond to the network request.

- Framework operation and architecture:

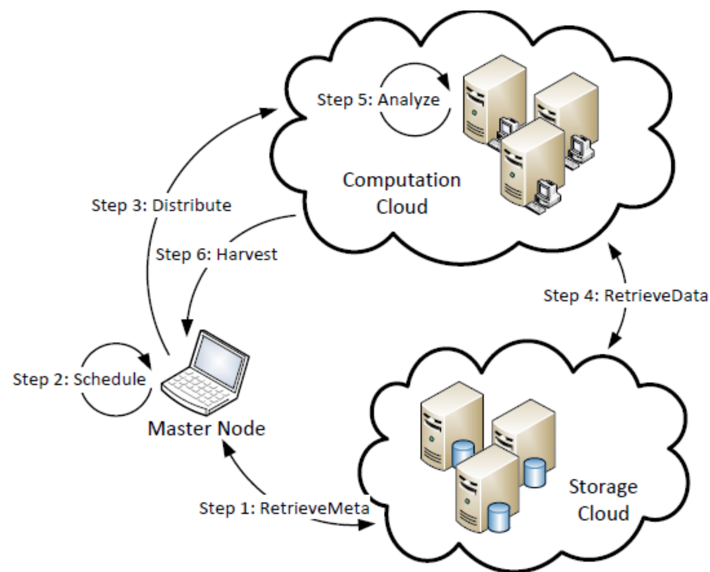

Figure 2-1 Distributed log analysis framework workflow [3]

RETRIEVEMETA (nss; lm)- Communication step in which the metadata of logs are retrieved from storage slave nodes by the master node. Metadata contains the URL of the stored log, count, range and average size of log entries,

1. nm SCHEDULE (tu; lm; ncs)- Task separation into task units and task assignment to the computation slave node by the master node
2. nm DISTRIBUTE tu - Responsible for creation of computation slave nodes and distribute cloud instances to them.
3. ncs RETRIEVEDATA d(tu) - Computation slave node retrieves log segments stored at the storage slave nodes
4. ncs ANALYZE (l; tu) - The logs are analyzed using the analysis executable. Multithreading is implemented and the thread result is then merged with the result of the node.
5. nm HARVEST ncs – Master node harvests results from the computation slave node. The global result is then given to the user.

2.1.6 Analysis App

The first step in identifying the security status of a system is to count the security events. The HTTP connection logs are analyzed and a group of connections are specified with the destination IP addresses as target. The algorithm is parallelized by separating the count onto various computation slave nodes. A three level hash map design is used in this model, master hash map, slave and thread hash map. This helps to effectively count the event occurrences with large tasks. Every hash map consists of key and record pair. Key contains the IP address of the destination and record consists of data needed to summarize the entries. Each key is mapped to a record. After the thread finishes the job, the slave hash map gets the merged result.

The image below shows the pictorial representation of the three-level hash map.

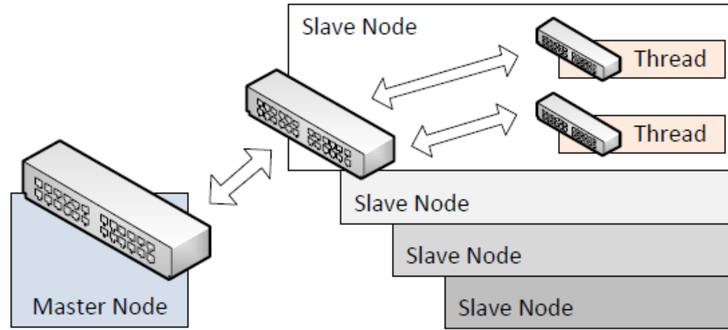

Figure 2-2 Three-Level hash map [3]

2.1.7 Evaluation

This prototype is easily usable. The user has to prepare the master node with access key pair creation and a security group created in the Amazon cloud. Then the master node creates the computation slave node. Three sizes of HTTP logs, 100M, 200M and 300M are used in the evaluations. The evaluation results are shown below.

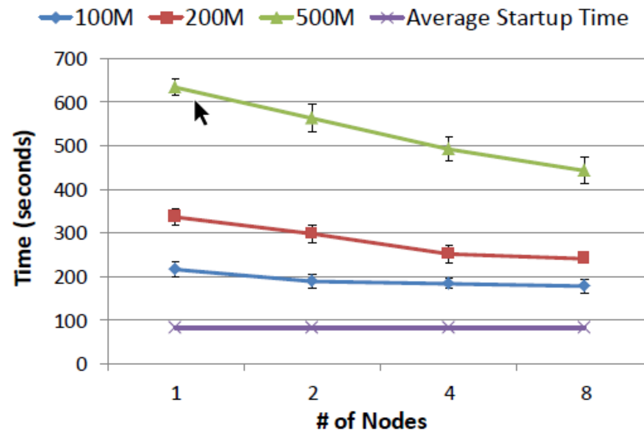

Figure 2-3 No. of nodes by time [3]

The speedup advantage is apparent on large dataset as expected. This system is designed to analyze large count of logs. The instance initialization is measured on the master node and hence this overhead is mitigated. The downloading overhead can be mitigated by using multiple storage slave nodes.

2.1.8 How this model differs from others?

This work differs from other antivirus systems with respect to the analyzing methods and objects. Elastic stream cloud identifies the gap between an application's streaming property and the standard cloud services. This design addresses the issue that occurs in streaming logs. As compared to the other distributed computing frameworks, this lightweight framework provides an easy and empirical way to deploy security frameworks with less distributing overhead.

2.1.9 Improvements and Conclusion

A lightweight distributed framework is implemented in this model. It contains minimum components and specifically designed for analyzing security logs. The framework is realized in Amazon EC2 and S3 environments for efficiency and easy use of the prototype. The improvements would be based on supporting complex log analysis and extending the framework.

Chapter 3

Existing Problem and Requirements

3.1 Unstructured Schema

Logs are more often viewed as a debug information useful in system maintenance. There was lack of sufficient use cases for logs to be properly structured and organized. For example, a log file which stores api session data just need to record the type of Http Method, Status Code, Error Message and any additional header parameter. Traditionally a raw log which above mentioned information served the purpose. In today's world, almost all systems produce lots of information. Not only that, now we also have sufficient capability to make sense of these data in order to benefit business operation or even improve the system performance or scale the system. Achieving this is not plausible if data chaos in form of unstructured data is not handled.

```
#Version: 1.5
#Software: Microsoft Windows Firewall
#Time Format: Local
#Fields: date time action protocol src-ip dst-ip src-port dst-port size tcpflags tcpsyn tcpack tcpwin icmptype icmpcode info path


2015-07-16 11:35:26 ALLOW TCP 10.40.4.182 10.40.1.11 63064 135 0 - 0 0 0 - - - SEND
2015-07-16 11:35:26 ALLOW TCP 10.40.4.182 10.40.1.14 63065 49156 0 - 0 0 0 - - - SEND
2015-07-16 11:35:26 ALLOW TCP 10.40.4.182 10.40.1.11 63066 65386 0 - 0 0 0 - - - SEND
2015-07-16 11:35:26 ALLOW TCP 10.40.4.182 10.40.1.11 63067 389 0 - 0 0 0 - - - SEND
2015-07-16 11:35:26 ALLOW UDP 10.40.4.182 10.40.1.14 62292 389 0 - - - - - - - SEND
2015-07-16 11:35:26 ALLOW TCP 10.40.4.182 10.40.1.11 63068 389 0 - 0 0 0 - - - SEND
2015-07-16 11:35:26 ALLOW TCP 10.40.4.182 10.40.1.11 63069 445 0 - 0 0 0 - - - SEND
2015-07-16 11:35:26 ALLOW UDP 10.40.4.182 10.40.1.13 62293 389 0 - - - - - - - SEND
2015-07-16 11:35:26 ALLOW TCP 10.40.4.182 10.40.1.13 63070 88 0 - 0 0 0 - - - SEND
2015-07-16 11:35:26 ALLOW TCP 10.40.4.182 10.40.1.11 63071 445 0 - 0 0 0 - - - SEND
2015-07-16 11:35:26 ALLOW TCP 10.40.4.182 10.40.1.11 63072 445 0 - 0 0 0 - - - SEND
2015-07-16 11:35:26 ALLOW TCP 10.40.4.182 10.40.1.11 63073 445 0 - 0 0 0 - - - SEND
2015-07-16 11:35:26 ALLOW TCP 10.40.4.182 10.40.1.13 63074 88 0 - 0 0 0 - - - SEND
2015-07-16 11:35:26 ALLOW TCP 10.40.4.182 10.40.1.13 63075 88 0 - 0 0 0 - - - SEND
2015-07-16 11:35:26 ALLOW TCP 10.40.4.182 10.40.1.13 63076 88 0 - 0 0 0 - - - SEND
2015-07-16 11:35:27 ALLOW UDP 10.40.4.182 10.40.1.11 55053 53 0 - - - - - - - SEND
2015-07-16 11:35:27 ALLOW UDP 10.40.4.182 10.40.1.11 50845 53 0 - - - - - - - SEND
2015-07-16 11:35:30 ALLOW UDP fe80::29ea:1a3c:24d6:fb49 ff02::1:3 57333 5355 0 - - - - - - - RECEIVE
2015-07-16 11:35:30 ALLOW UDP 10.40.4.252 224.0.0.252 59629 5355 0 - - - - - - - RECEIVE
2015-07-16 11:35:30 ALLOW UDP fe80::4c2e:505d:b3a7:caaf ff02::1:3 58846 5355 0 - - - - - - - SEND
2015-07-16 11:35:30 ALLOW UDP 10.40.4.182 224.0.0.252 58846 5355 0 - - - - - - - SEND
2015-07-16 11:35:31 ALLOW UDP 10.40.4.182 224.0.0.252 137 137 0 - - - - - - - SEND
2015-07-16 11:35:31 ALLOW UDP fe80::4c2e:505d:b3a7:caaf ff02::1:3 63504 5355 0 - - - - - - - SEND
2015-07-16 11:35:31 ALLOW UDP 10.40.4.182 224.0.0.252 63504 5355 0 - - - - - - - SEND
```

Figure 3-1 Unstructured log message [17]

Logs which are unorganized can be problems to both the parent application and application which consumes these logs. For example, it would be difficult to come up with a generic solution to process large amount to unstructured logs. It would also be a nightmare when trying to relate log messages to one another when debugging a code flow. Lastly, it may lead to degraded performance for application as logging libraries may strain in logging these large unstructured data.

3.2 Strongly Coupled Storage

Logs need to be stored for future reference. Type of storage used plays a big role in determining the usability and scalability of a logging system. Storage should provide better accessibility for logs. It is not worth while maintaining logs if they are not easily accessible. Fast retrieval and easy access is the primary purpose of having any logs. If logs cannot be retrieved fast, it will be almost impossible to have a real-time data analysis. Real time data analysis is in widespread use today. Almost all large and small companies use real time data analysis to create business growth strategies. E-Commerce companies such as Amazon and Alibaba uses real time data analysis to provide on spot offers and recommendation to customers. Weather forecast systems, mission critical in-flight system and many others use these data to perform critical core tasks. It is not difficult to imagine the problems an inflight system would face if weather data is not available to it in a timely manner.

From an implementation perspective, a logging system should have loosely coupled storage. Consider for instance, a system using relational database such as SQL as its storage option. If the same code base is used across different environment, the logging system would face incompatibility issues. For example, android does not have full-fledged support for SQL but uses a lighter version called SQLite. For embedded

systems database like Firebird and HQL works better. It is evident from this that portability of application would become a problem due to storage bottleneck.

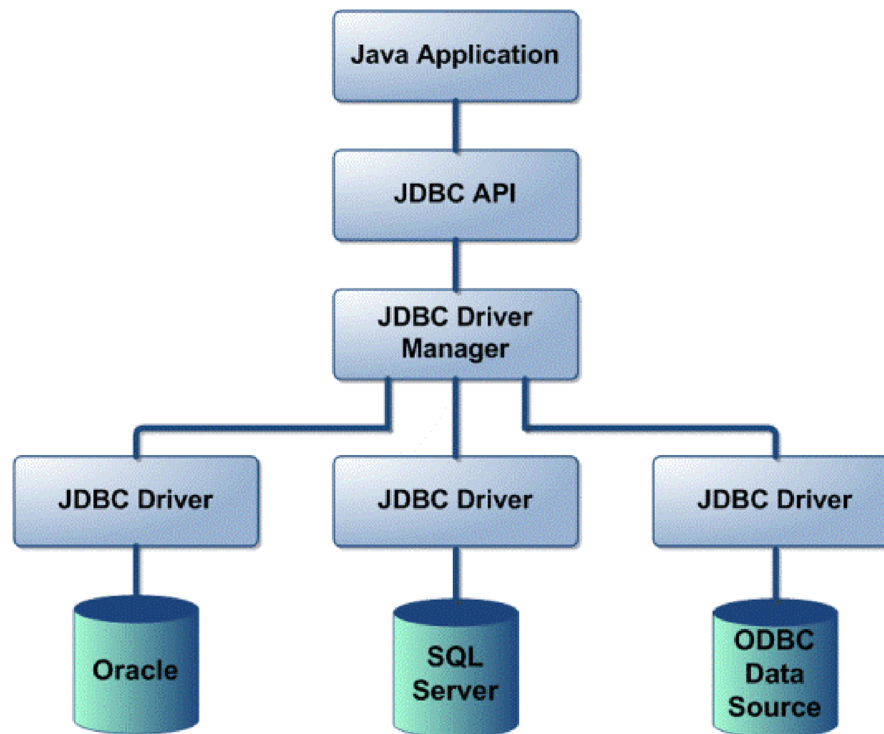

Figure 3-2 Storage Decoupling [18]

Lastly, decoupling storage from any system is imperative if the system needs to scale. A system with small number of user base may use a relational database for storage purpose. However, as system starts catering to larger number of users, a NoSQL storage might become a better option. A tightly coupled logging system would be difficult to scale in this case whereas a loosely coupled system would migrate easily from relational database to a NoSQL database. Speaking succinctly, a storage component for a logging system should be a pluggable component.

3.3 Lack of Visualization Capabilities

Logs are very important piece of information depending on how they are used. They can be used to scale system in terms of performance or business coverage. They can be used for devising business development strategies such as user retention, user recommendation, etc. However, raw data is of least significance, especially if these data are in terms of gigabytes or even more. From elementary mathematics level itself, it is clear how graph and charts prove to be more meaningful as compared to raw information or even data tables. It follows similarly for log analysis too. Logs are beneficial and a powerful tool only when they can be comprehended easily and effectively. It is nearly impossible to analyze huge logs in form of raw data.

Figure 3-3 Sample Dashboard

Visualization solves this problem by expressing raw data in terms of figures and graphs and charts. They can be easily understood and any decision can be taken much faster when referring to visualizations as compared to plain raw data. Another advantage

of visualization over raw data is that, they make sense to almost any one in the organization. A person need not be technically sound like a developer to understand and decode a visualization as is the case with raw log data which needs more technical expertise to decode and understand. Many companies have dedicated team to convert these raw log data into visualization which can then be delivered to higher management or marketing teams for their specific tasks.

Not every logging system comes with a visualization capability built in. In fact, currently visualization system such as Splunk are used together with logging frameworks like log4J to produce meaningful visualization. Other examples include loggly, Elasticsearch-Logstash-Kibana stack and grayLog2 as an alternative to Splunk.

3.4 Handling Data Loss

Logs are basically data that are produced at one site and consumed at another. Almost, all logging systems implement some kind of data flow mechanism to ensure that logs are transmitted to the required destination and stored properly. On the other hand, clients who consume these logs also rely on their transmission from storage site to current location. Most of the time logging is synchronous, which means as long as parent system works, data will be logged without any problem.

Figure 3-4 Data Streaming [19]

However, as data to be logged becomes huge, synchronous processing of logs no longer works. Many logging system, such as loggly uses asynchronous processing. This ensures that logging process does not degrades the performance of parent application. Asynchronous processing however is dangerous in terms of maintaining consistency in operation. Consider the example of video streams, almost all of which are asynchronous data streaming. It is not hard to notice the jitters and lags associated with any media streaming. Asynchronous log processing also comes with similar problems. There is a possibility of data loss for example when producing code produces data but consuming code delays consumption. Data may fall off the pipeline after delay reaches a threshold. Systems such as RabbitMQ are very efficient and fast in data transfer, however due to lack of their own temporary storage, they do not guarantee data consistency.

3.5 Real Time Analytical Capabilities

An organization having tools to analyze user and system data is definitely at higher advantage than organizations who do not have such capabilities. However, to be

at even higher advantage these capabilities should be near real time. An ecommerce website can run a batch job every day on users visiting their site and build recommendations to be shown in future. On observing closely this is not how large e-commerce organizations like amazon provide recommendation. Based on a user's current search he is provided with other product recommendations. Even YouTube provide recommendations just after user finishes watching a video. This would not be possible if user data is not processed almost instantaneously.

Real time analytical capabilities are difficult to achieve by using existing logging and visualization frameworks. This would need multiple systems working in sync with each other. This synchronization would cover stages such as logs transfer and storage, log retrieval and raw data processing, application of analytical algorithms and finally feeding data to visualization.

3.6 Performance Scalability

The amount of logging required for an application solely depends on business requirement. For example, an internal service needs to log only application specific logs. A client facing application may also log user data to study their behavior which may help in developing business strategies. Normally these clients facing systems need to scale themselves as their user base grows. This also means that the logging systems the application uses needs to scale too. Scaling all applications are quite a challenge for the teams working on it. It is possible only with futuristic design and early thought on how components of the system would need to be modified as user base grows. Scaling logging systems are also equally challenging as they deal with multiple components such as storage, messaging queues, flow algorithm and visualization service.

While scaling a logging system, the primary components to keep in mind are storage and messaging queues. Storage is responsible for storing the logs in various format as specified by the business requirement. Some storage medium is good at handling large volume of data while others are not. Some other storage solution provides better searching while cannot handle very large data. For example, relational databases such as MySQL are very efficient in data search using their easy to write queries. They also are good at fast data retrieval by indexing techniques. However, they are not preferred database solutions when storing user log data as this can proliferate easily and relational databases may not be able to support it. On the other hand, NoSQL databases such as mongo DB and Elastic search are very efficient in storing large amount of data. They can theoretically store infinite amount of data and since they store data in file format, their capacity is limited only by the machine's storage.

Messaging queues are responsible to act as a temporary in memory buffers which aids in transportation of logs from creation site to persistent storage. Not all messaging queues handles large data volume efficiently. The need to deal with high data in memory when the consumer is busy or is down. They also need to deal with concurrency problem when a data pulled by one consumer from the queue is no longer available to another consumer waiting for same data. Messaging queues have limited life time from an application perspective. Once they are past due this lifetime threshold, they drop data from temporary buffer which results in data loss. Finally, most messaging queue do not have data replay feature which can be used to minimize data loss.

Table 1 Throughput calculation of a messaging queue system [12]

Concurrent Users	Time (ms)	Throughput (tps)
1	10	100
50	500	100
100	1200	83.333
150	2200	68.182
200	4000	50

Chapter 4

Solution Design and Components

4.1 Base Logging Framework

Even if trying to build a logging system which expands over traditional logging feature to provide numerous other features as discussed before, logging data remain the core component of the system. Logging data seems to just print message in console or file. However, a lot of complexity goes on in building a logging framework. Instead of reinventing the wheel we can use existing frameworks such as log4j, Logback or java logging libraries. It is important to analyze however, which solution fits the purpose. It would be better to have a logging framework that serves exactly the data logging purpose without any overhead of additional features. We would be using Log4J for our purpose as it is the minimalistic framework and quite stable one.

Log4J is around for quite some time now. Its first version was released in 2005 as per maven central data and it has matured through last decade. It is one of the most trusted and most used logging framework in java ecosystem today. It provides all the required logging levels namely debug, info, error, critical and all. It supports multiple logging mechanism such as console logging where log messages are pushed to application console or ide console window. Another mechanism is file appender where it is possible to specify the file name where log messages needs to be pushed. The most popular one and which makes log usability most easy is the rotating file appender. In this mechanism logs are appender to file as in file appender mode but with a difference. Based on log4j configuration of maximum file size, log4j framework will create a new file automatically and start logging new messages here, once the previous file has reached the size limit. This is quite beneficial as it avoids the problem of having a single text file of

humungous size. Another feature provided by log4j is the layout configuration where user can provide message layout and the messages will be logged in the required format. Figure 4.1 shows example of a log4J configuration file.

```
log4j.rootCategory=DEBUG, rotatingFileAppender
log4j.appender.rotatingFileAppender.alayout=org.apache.log4j.PatternLayout
log4j.appender.rotatingFileAppender.maxsize=1000000
log4j.appender.rotatingFileAppender.customProperty="custom"
log4j.appender. rotatingFileAppender.fileextension=".log"
log4j.appender.rotatingFileAppender.layout.ConversionPattern=%d{yyyy-MM-dd
HH:mm:ss} %-5p %c{1}:%L - %m%n
```

Figure 4-1 Log4J Properties Configuration

- Root Category – This property determines the logging level of logs. In the above case, logging level is debug which means all the logs of type debug, info, error, critical will be logged. Another parameter to this property states that system will be using a rotating file mechanism.
- aLayout – This property determines the type of layout being used for logging. In this case it is pattern layout. Another type of layouts is AbstractStringLayout, CsvParameterLayout, HtmlLayout and JsonLayout among many others.
- Max Size – This property specifies after what size the log4j system need to switch to logging into a new file. This size is specified in bytes.
- Custom Property – User can also specify their own property which are not meant to be used by log4j framework directly. For example, it can be specified that which micro service is the current system under run.
- File Extension – This property can specify the extension of files that are created by log4j framework. These extensions can be “.txt”, “.log”, etc.

- Conversion Pattern – Here user can specify the formatter for log messages. The formatter specified in above example will print log messages in the format below. <Date> <Priority> <category of logging event> <Line Number> <Message>

4.2 Centralized Storage

Storage is an important component of a logging system as it plays a key role in determining the performance and efficiency of the system. It also determines the ease with which the system can be scaled when required. Having a good storage solution is important, but if the storage is centralized it adds a lot to the existing benefits. Having a centralized storage like AWS S3 removes the burden of maintaining in house storage solutions. Since, the servers are acquired from AWS, the business can save a lot in terms of maintenance, support staff, on site facilities and real estate. The major benefit however, is cost of usage and scaling. AWS S3 enables us to pay for what we use. Therefore, we end up not wasting resource as in case of in-house solution where even for low usage entire storage has to be setup. Second benefit is scalability of storage solution when required. Scaling an in-house storage solution is costly and required additional maintenance as oppose to S3 which can scale on demand.

We choose AWS S3 as our centralized storage solution for following reasons:

- Ease of Integration: One of the best feature of AWS S3 is its easy integration using Amazon's Restful API with any third-party application. AWS SDK is available in almost all modern programming languages such as Java, C#, Python, Ruby and others. The AWS documentation is quite self-explanatory and has an ever-evolving ecosystem of developers and maintainers.

- On-demand Scalability: AWS Services such as S3 can be easily scaled as demand increases. Not only that, the scaling of services do not even require manual intervention and they can be done automatically using alerts which notifies the system of increasing or decreasing load to tune the system accordingly. Normally an in-house solution can only be scaled up, however, AWS services can be scaled up and down continuously depending on the requirements.
- Trust and Security: AWS Services are built of a very secure model of IAM (Identity Access Management) roles and policies. The stringent security is maintained on S3 buckets using bucket policy. Also, the security is implemented in AWS S3 by a file by file model using SSL.
- Content Storage: AWS S3 stores data in file format which makes its model content independent. It is theoretically possible to store any type of file and of any size. This way AWS S3 can also form a basis for content distribution systems.
- Amazon QuickSight UI: AWS new service QuickSight offers big data analytics feature on top of data stored in AWS S3. In the world of big data analytics where organizations have to deal with multiple technology stack AWS provide the storage and analytics solution all under one hood.
- Backup and Disaster Recovery: Amazon provides great tool for backup and disaster recovery in event of any failures. This is achieved using amazon's CCR (Cross Region Replication) technology where S3 bucket. Can be replicated across any number of amazon's world-wide data centers.

- Free Usage and Pricing: In consistency with many other AWS Services, S3 also provides a flexible pricing model. A user can select a free S3 tier to get started and this is sufficient for most of the individual contributors. In the free tier AWS S3 provides 5 gigabytes of storage which includes 20000 GET request, 2000 PUT requests and 15 gigabytes of data transfer quota per month. The pricing for advanced usage is also quite reasonable and can be easily scaled.

4.3 Non-Blocking Execution

Logging is not itself a business requirement but it provides support to the business in various ways. It is therefore important to understand that logging should never become a bottleneck to the parent application. A code to log some data should not hinder the application run in any way. Consider the example of an exception thrown while trying to log a message. This exception will then propagate to the parent application entry point (normally main() method) and the application would terminate with an exception. It needs to be ensured that logging code runs in mutual exclusion with parent application code and any state modification of this system should be independent of parent application.

Generally, all asynchronous operations are thought as a non-blocking operation which is however a misconception. It is important to understand the difference between these two before a correct implementation can fall in place. Any asynchronous call made from the calling site is expected to return later at some point if time with some response data. This data is then consumed and application progresses. However, at this point the parent code is terminated and put on to the stack and the control is transferred to the calling site of asynchronous call. This in itself is a blocking scenario. In a non-blocking execution however, a call is issued and is expected to be taken care at its independent

pace and the parent application execution continues immediately. In this case, after a call is made it never returns with any response and the calling site is not aware of what is happening or has happened at the called site.

From the figure below, the application shows two threads, the producer thread which can be a main application thread, and the consumer thread which can be an independent thread on its own. Once the producer thread pushes a message onto the queue, it is the responsibility of consumer thread to listen to the queue and read the messages. However, it is clear from the figure that main code which is the producer

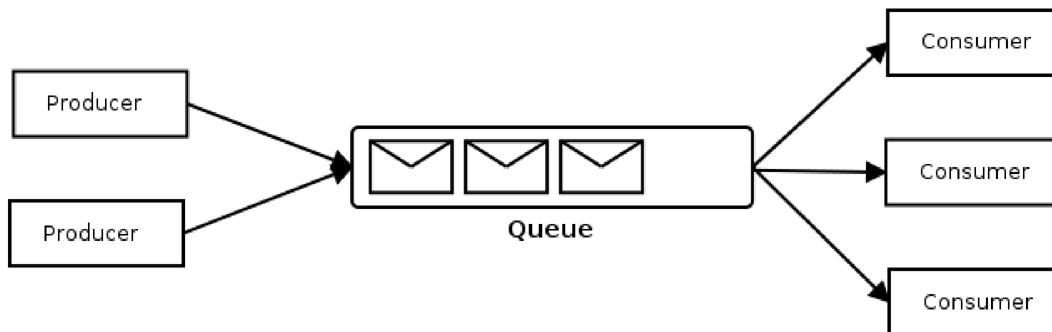

Figure 4-2 Producer Consumer Threads with Shared Queue [13]

thread continues at its own pace after pushing the message on the queue and is never aware of what has happened on the consumer side. The consumer never returns a value to the main thread which also means that it never become a reason to block execution go the main thread.

4.4 Avoiding Large Number of Thread Spawn

From point 4.3, it is clear that a non-blocking execution would require a separate thread on which logging code would run. From a broader view this approach looks quite efficient. A separate thread would never hinder the execution of main thread and

application would run perfectly as a whole. However, whether the application would run perfectly or not also depends on how many number of thread the application would be using. The application would have one main thread for sure. Apart from this thread, logging thread can have either one replica such as in case of a desktop application or a multiple replica in case of a web application.

A web application serves web pages to the client. The code behind for every web page contains application logic. If a logger object is instantiated at each of these web pages it would create a multiple thread spawn problem. The logging library defines the number of thread it would spawn for purpose of consuming logs. If this number is 'n' and there are 10 webpages served simultaneously by the web server, the webserver would be handling a total number of 10 * n threads at a given point in time. Since, all consuming threads would be doing similar jobs of consuming messages and pushing to a centralized storage, they need not be duplicated to such a high number. A singleton pattern can be used to ensure that at any point of time, the application just runs a single instance of logger class. This will ensure that the logger class spawn the define number (n) of thread just one time. So, if as per configuration we have a max number of simultaneous thread as 'n', at any point of time the application will have no more than 'n' running threads.

4.5 Data Pipeline using Kafka

Kafka is a modern system with a high degree of resilience. It is a distributed system which can be easily scaled horizontally by adding new clusters. The working of kafka is based on "topics". Topics are entities which stand independently of application code and any system can publish messages to these topics and other systems can consume messages from the same topic. For example, a topic named

“UserInformationChanged” can serve the purpose of dealing with user information changed functionality. If a user information is changed, the related payload can be pushed to this kafka topic. Any consumer who wants to receive information about what user information changed, can subscribe to this topic to consume data.

The resiliency of kafka is provided by notion of offsets. All messages pushed onto kafka queue are related to an offset which basically acts as an index for the message. To retrieve the message all a client needs to do is call GET with the required offset. Since kafka has its own configurable storage system (generally configured for 30 days data retention), if a consumer code failed and data is lost, data can be easily replayed just be going back to that offset.

4.6 Elasticsearch DB

The raw messages in AWS S3 are useful if they can be processed to a meaningful format like JSON and then can be converted into visualizations. SQL is a tabular structured database and does not support JSON inherently. Even though SQL has great query filtering such as aggregations, ordering and grouping, it does not have any visualization support. Any data in SQL if needs to be visualized in a chart or graph format will need to integrate with visualization tools such as tableau and powerBi. Elasticsearch is the database of our choice. It supports JSON as its primary data format and has a sister application called Kibana which is quite popular and efficient for creating rich visualizations and dashboards. In fact, Elasticsearch is marketed under the umbrella of ELK Stack (Elasticsearch, Logstash, Kibana).

We use Elasticsearch because of its following benefits:

- Lucene Base: Elasticsearch is built on top of Lucene. Lucene is a fully featured data retrieval kit. This way Elasticsearch leverages almost all information retrieval algorithms from Lucene. Since Lucene is familiar to many developers it is easier to pick up Elasticsearch usages and get up to speed fast.
- Full Text Search: Elasticsearch provides search features such as partial text searching, full text searching or facetted search. It provides multiple filters such as "bool", "must", "match", "should", "range", etc.
- Auto completion and Fuzzy Searching: Elasticsearch makes it very easy to build UI's for auto completion feature where Elasticsearch provides matches as user types. Using fuzzy search an application can leverage spell check algorithm as part of Elasticsearch SDK itself.
- Full Query Feature: Elasticsearch supports complex SQL like query with ease. It has support for select, inset, delete, group by, order by, aggregations and almost all SQL like query params.
- Document based storage: Elasticsearch stores real world entities in form of structured JSON data and uses indexing for efficient searching. It uses indexes on JSON fields(Keys) to provide maximum performance.
- Retrieval Speed: Since Elasticsearch uses structured data, it can find the document using indexes very fast. It also caches structured data for a particular query and they are returned from local cache if the query is exactly same.
- Scalability: Since Elasticsearch is inherently a distributed system, it is built from ground up keeping scalability in mind. It can be easily scaled horizontally by adding new clusters which reduces the load on existing clusters.
- Restful API: Elastic search driven by restful APIs, which makes it possible to perform action using a simple Restful Service.

- Fully Featured IDE: Elasticsearch has a mature ide to test Elasticsearch queries and view results. It can be used as a debugging tool to investigate any problems with the query. IT can also be used to create or delete static indexes. Finally, almost all admin related jobs require this IDE in some way or another.

4.7 Kibana Dashboard Visualization

As a final step of a logging system we need to ensure that we get proper visualizations from the logs that are gathered. There are many visualization tools in the market today. Some of them are Tableau, powerBi and Kibana. Tableau is a great visualization tool. Its development has mostly revolved around providing rich visuals to user without worrying much about underlying data storage. Tableau does not have storage on its own. However, there are multiple connectors which tableau makes use of. It has connectors ranging from CSV, MySQL, S3 and lots more. However, it does not as of now has a Elasticsearch connector. PowerBi is similar to Tableau in lot of ways. Our solution of choice is Kibana. Since Kibana is shipped as a part of ELK stack, it makes lot of sense to use it in coordination with Elasticsearch.

Kibana also provide following benefits:

- Search Highlighting
- Enhanced Query Aggregation
- Scripted Fields
- Automated Dashboards
- Client Server Separation

Chapter 5

Implementation Overview

The implementation can be efficient and scalable if it adheres specifically to problem statement. The outline of customer problem statement are as follows:

5.1 Customer Problem Statement

- Need ability to store application logs to a centralized location such as AWS S3.
- Unlike traditional way of viewing logs as a text file, should be able to view log data in a structured way.
- Should be able to filter logs by parameters such as datetime, component or any other key present as part of log data.
- Should be able to make meaningful analysis from logged data using visualizations.
- Visualization should reflect most recent data as per the use case. For example – Number of users logged in every hour can have a delay of an hour of data.
- The system should enable data to be accessed by visualization platform such as tableau.

5.2 System Requirements

The problem statement needs to be translated into concrete requirements. Since, logging systems may need to scale in the near future, both functional and non-functional requirements need to be addressed from the onset of development lifecycle.

5.2.1 Functional Requirement

- System should be able to log messages to S3 in real time.

- Log data should follow JSON format with “Message” as the mandatory fields and other supporting optional fields.

  For example,

  ```
  {
  "DateTime": "2017-05-11T14:23:45Z",
  "Message": "Payment Successful for user 5065188077",
  "Status": "Success",
  "ThreadId": "8",
  "Component": "AuthorizePayment",
  "SessionId": "as3es-ger3d-iq993-wqddw",
  "RequestId": "1033923",
  "ErrorCode": "0"
  }
  ```

- The size of log files should be configurable by the user. For example, a developer can specify that he wants all log files in size of 100KB each.
- The log files should be stored in structure as follows:

  <S3 Bucket Name>/Module/”MM/DD/YYYY”/log_mm_dd_yyyy_hh_mm_ss

- The storage structure should make it easy for client scripts to read log data based on timestamps. For example, I should be able to iterate over all logdata from now to now – 5 days.
- Filter should be provided as a dropdown which options such as by day, by month. Etc.
- The data pipeline should be properly compatible with ELK visualization stack.
- The visualization should be automatically updated if new data arrives.

5.2.2 Non-Functional Requirements:

- Data Should be logged real time to a centralized location.
- Logging Framework should have 100 percent yield to avoid data loss.
- Logging activity should not hinder the main execution of parent system.
- The centralized server should have high availability.
- A pipeline which pulls logged data and creates visualization should be in place.
- All the visualization should provide data which is near real time to make better sense.

5.3 High Level Solution and Code Flow

1. Client Application can use existing logging frameworks such as log4j for Java or log4net for .Net frameworks.

2. A custom appender class needs to be implemented which uses AWS libraries to commit data to S3 bucket. This appender class will be used by log4j/log4net framework.

3. This system can then push log messages to S3 bucket.

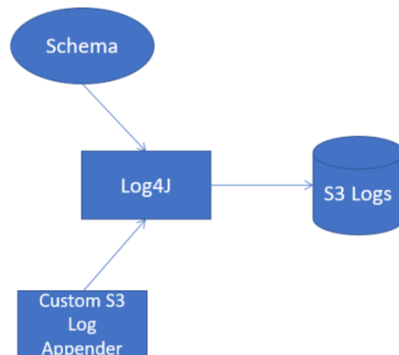

Figure 5-1 Logging to S3 (Producing Logs)

4. Once logs are in S3, a Jenkins/python pipeline can fetch the logs from S3 and push to an Elasticsearch index.

5. Then visualization can be built on top of data present in ES index.

6. Since Kibana visualization have automatic update features, any new data arrival in the ES index will be picked up by the visualization.

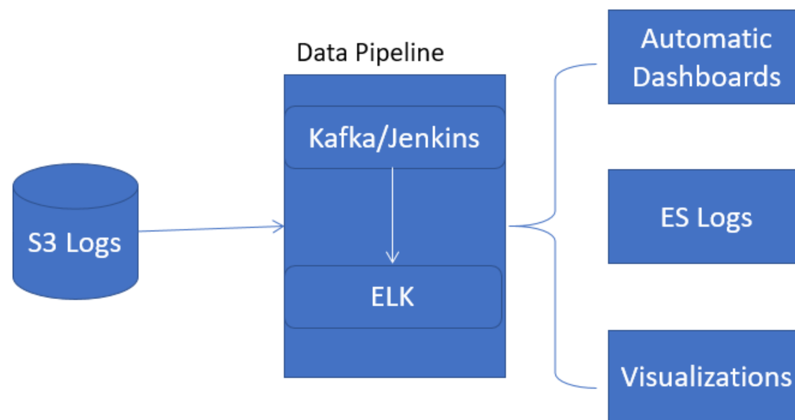

Figure 5-2 Log Consumption and Visualization Pipeline

5.4 Implementation

1. Custom appender may use multi-threading to increase performance. Instead of using raw threads, advanced frameworks such as "SharedThreadPoolExecutor" is being used.

2. The log file size is made configurable by keeping it as a configuration property in log4j.properties file.

3. The layout.conversionPattern property is also configurable and can be changed by the user depending on the format in which log message string is required.

4. A class UTA Logger is created as a wrapper class for logger object. This call takes care of formatting log message and any supporting fields such as datetime in a valid json object.

5. The primary purpose of this class is to act as a singleton. This is important because our solution uses multiple threads. Creating multiple objects, each spawning multiple threads is not a good design and can lead to performance issues in a multiple page web application scenario.

6. The S3Appender class extends from AppenderSkeleton Class. Therefore it needs to override methods activateOptions(), append(LoggingEvent) and close().

7. activateOptions() is called when the appender class is instantiated using log4j manager object. Here, consumer threads are initialized and started. Any other initializations such as S3 authentication keys, etc. are done here.

8. Append() method is invoked automatically when a logging method such as debug() or info() is called. We get the rendered message from LoggingEvent parameter.

9. A StringBuilder is used as a buffer to append log messages and keep it in memory until the log file size specified by the user is reached. At this point all accumulated messages are flushed onto local disk.

10. The consumer thread which continuously checks for log data on the disk then picks up the file and reads its content.

11. Logformatter method is then called with the message content to add any additional fields and check for validity of JSON.

12. The returned data from Logformatter() method is pushed to S3.

13. The filename to which message will be pushed in S3 is constructed on the fly using current datetime and random string returned by the method getSaltString()

14. Close() method is implemented to take care of gracefully exiting all the worker threads when the application shuts down.

15. In close(), the code also checks for any un-flushed message and queues it for processing. If the size of buffer is zero, the threads are exited and resources freed.

16. The appender uses Google's Gson library for any JSON related manipulations and is the only dependency apart from native log4j.

5.5 Design Class Diagram

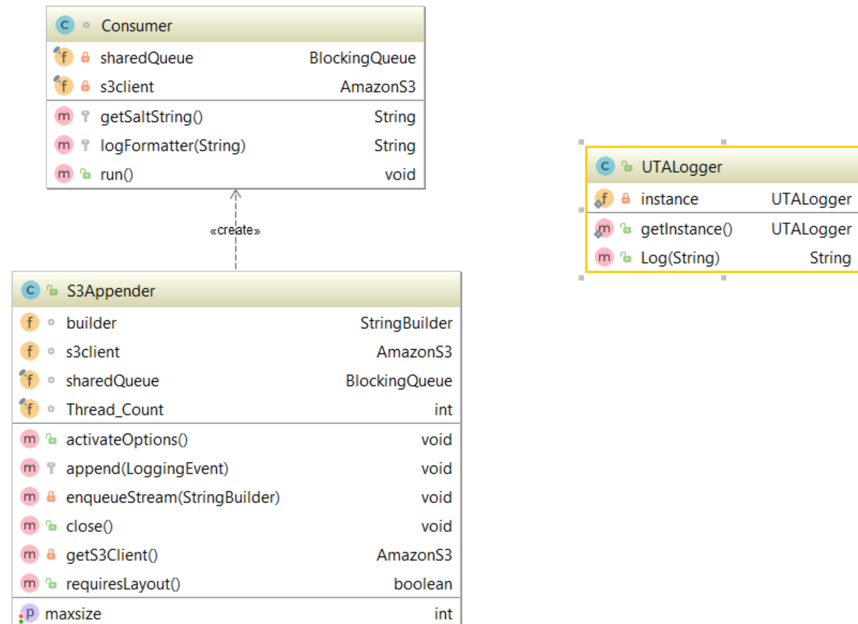

Figure 5-3 Design Class Diagram

5.6 Code Usage for Client

5.6.1 Properties File

```
log4j.rootCategory=DEBUG, S3Appender
log4j.appender.S3Appender=org.uta.s3appender.S3Appender
log4j.appender.S3Appender.layout=org.apache.log4j.PatternLayout
log4j.appender.S3Appender.maxsize=100000
log4j.appender.S3Appender.layout.ConversionPattern=%d{yyyy-MM-dd
HH:mm:ss} %-5p %c{1}:%L - %m%n
```

5.6.2 Sample Usage

```
package org.uta.s3appender;
import org.apache.log4j.Logger;
public class MainApp {
        final static Logger logger = Logger.getLogger("org.uta.s3appender");
        public static void main(String... args) throws Exception {
        logger.debug(“Test Debug Message”);
        logger.debug(UTALogger.LogFormatter(“Message”, “SessionId”, “RequestId”,
        new Resource() {{
        put(“key1”, “value1”);
        }};);
        }
}
```

5.7 Test Cases

Table 2 Test Cases for S3 Appender

Test Case Desc	Pre-Condition	Expected Output	Actual Output	Status
To verify if logger object is getting initialized or not.	S3 bucket should be configured.	On running the application, logger object should be initialized without any errors on console.	Logger object was initialized successfully and console did not show any errors.	
To verify if a simple plain message is not logged to S3 before application exits.	S3 bucket should be configured.	A short message should not be logged to S3 before application exits, to prove message is still in the buffer.	Message was not logged to S3.	
To verify if a simple plain message is logged to S3 after application exits.	Application should be in terminated mode. S3 bucket should be configured.	The buffer should be flushed and message should be logged once application exits	Message was displayed in S3.	
To verify if messages sent	S3 bucket should be configured.	The S3 bucket should receive all	The messages were dropped in	

continuously is received by S3.		the messages as soon as possible. (average < 2 seconds)	S3 averaging less than 2 seconds	
To verify the throughput when sending bulk of messages	S3 bucket should be configured.	There should be no message loss. All the data should be successfully stored in S3.	The data was stored in S3 without any message loss.	
To verify if size of file stored in S3 changes accordingly.	Change file size parameter in log4j properties config file. Test Data: Give file size as say 10000, 12000 and so on, S3 bucket should be configured.	The file size should be according to what is specified in config.	The file size was in accordance with config parameter.	
To verify that all the messages are properly flushed on application exit.	S3 bucket should be configured.	Any remaining message in buffer should be logged in S3 on application exit.	Messages were properly flushed and logged on application exit.	

Chapter 6

Technical Approach

6.1 Storing Logs in S3 Bucket

Choosing AWS S3 as the centralized storage solution provides a lot of benefits including cost optimization and easy scalability. All AWS services are tied to a region and it is recommended to select a region by keeping few things in mind. First, not all services are available in each region and it is better to choose a region with more number of datacenters. Second, it is always beneficial to spin up a cluster in the region which also have the application data.

Though easy to configure, S3 requires few configurations that are important to leverage benefits that it can provide. The S3 configuration uses a concept called AWS profile. Use of AWS profile is optional, but recommended.

6.1.1 AWS S3 Configuration

There are four important S3 config properties. They are Bucket Name, S3 Access Key, S3 Secret Key and AWS Region. There are multiple ways to provide these information to the AWS SDK. First by creating an AWS profile which can be done by creating a file .aws/credentials in the root directory in case of Unix based system and mac. On windows this file can exist in the directory containing windows installation. This file can contain the properties specified above in form of key value pair.

Another way is to provide it as part of application properties file and use then in s3_client api as shown below. The properties can be fetched using default getter and setters for specified property name. In Java, for getter setter to work properly camel casing needs to be followed for variable names corresponding to properties in the properties file.

```
s3bucketpath="S3-Logs"
s3accesskey="AKIAIBM4DS6XPQP4VGCA"
s3secretkey="JqAso8pSF8GJSnudwPqsid4RdsdfN8FjyajS7fF"
```

Figure 6-1 S3 Configuration

6.1.2 S3 Folder Structure

In a scenario where the system is intended to collect lots of data on a daily basis, it is good have a folder level separation for daily logs. For example, folder naming can follow pattern as /Module/<MM/DD/YYYY>/. This makes sure that any consumer will have capability to segregate logs based on daily filters. The files in a particular folder can also follow certain pattern. Though this is not important, this will make implementation easier and we will have more parameters to validate. One such pattern for log file can be "application_log_mm_dd_yyyy_hh_mm_ss.txt". In this way, even without any other parameters given to us, it will be easy to make out some pattern among the logs just by using their file names.

6.1.3 AWS SDK S3 API

AWS provides api for S3 in almost all modern programming language. We use Java api for our purpose. The connection is established by calling AmazonS3Client constructor with user profile information which returns an S3 Client object. The S3 client object provides access to various functionalities such as copying objects from one bucket to another and deleting objects. It provides usages such as create bucket, delete bucket, delete bucket configuration, delete object and get object from bucket. We use S3 Client's list api which gives us keys for all the logs present in the bucket. This key acts as a handle which can be used to retrieve log file content and load it into memory for various uses. Since, we also need to upload log files to S3, S3 client also provides support for put object using which a log file can be uploaded to S3 bucket.

6.1.4 Maintaining S3 Policy

S3 bucket policy specifies what actions are allowed on a certain principle inside a S3 bucket. S3 bucket policy follow ACL (Access control list) type. Example of an S3 bucket policy is shown below.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": ["arn:aws:iam::111122223333:user/Alice",
                "arn:aws:iam::111122223333:root"]
      },
      "Action": "s3:*",
      "Resource": ["arn:aws:s3:::my_bucket",
                   "arn:aws:s3:::my_bucket/*"]
    }
  ]
}
```

Figure 6-2 S3 Bucket Policy

6.2 Schema Structuring

While logging has become one of the major components of many system in the modern world, most of the systems do not give structure to their logs. They either do not see any use case for having structured logs, or are just unaware of the values it can provide. There is always a use case for having a better schema without any doubt. Even for applications where logs just work as a debug information, structuring logs can provide many benefits. One is that having structured logs eliminates the time wastage a developer has to suffer browsing through unreadable logs.

Second, Logs can be shared across developers and to higher management as it becomes more and more understandable. In application which records user's data, it is a no brainer to say that structuring logs is the first and most important task to cover while building a logging system. In such an application, log's structure becomes the building block for all visualizations and dashboards that can be made. How real-time data can be viewed on the dashboards also depends on how schema bound the logs are.

6.2.1 LogFormatter Class

We use a Logformatter class to bind schema to our log messages. This class is also made a singleton class as the only method it exposes is a static method. This class has a static method called format() which does the required structuring. The method accepts log parameters as key, value pair and stores them in hash map. These key value pairs are then used to build a JSON object which is passed back to the logger object. This class also exposes a getInstance() method which returns a new instance of the class only if there are no existing instances, thereby following principles of singleton pattern. The implementation roughly looks as shown below.

```java
public class UTALogger {
    private static UTALogger instance = null;
    static List<Integer> range = IntStream.rangeClosed(-8640000, 8640000)
            .boxed().collect(Collectors.toList());
    static Random rand = new Random();
    public static UTALogger getInstance() {
        if(instance == null) {
            instance = new UTALogger();
        }
        return instance;
    }

    public static String LogFormatter(String message, String sessionId, Map<String, Object> resources) {
        Map<String, Object> map = new HashMap<~>();
        map.put("message", message);
        map.put("datetime", "2017-12-12");
        map.put("sessionId", sessionId);
        map.put("ThreadId", Thread.currentThread().getId());
        Gson gson = new GsonBuilder().create();
        String json = gson.toJson(map);
        return json;
    }
}
```

Figure 6-3 LogFormatter Wrapper

6.2.2 JSON/AVRO Data Format

When deciding about providing structure to our data, the next big decision to make is about the data format. There are many data formats available which are commonly used. XML is a mature data representation language with support of serialization/deserialization. This means the data can be easily transferred over wire and it works with almost all network protocols. The problem with XML is their verbosity. Consider an example below.

```
<Users>
<User>
   <name>John</name>
   <ssn>123-456-7890</ssn>
</User>
<User>
   <name>Maria</name>
   <ssn>123-000-4321</ssn>
</User>
</Users>
```

Figure 6-4 Sample XML

In the example above, it is clear that for handling large information sets, xml is not a good alternative from the storage perspective. It takes up more information for its metadata as compared to the actual data itself. This means that serializing and deserializing such data can be memory expensive.

Json on the other hand solves this problem to a great extent. It uses bracket notations to as an alternative to opening and closing tags in xml. It also provides construct to match with objects and arrays as in any programming language. The example below shows the above data in json format. Json looks quite concise and provides multiple benefits. Due to its lesser metadata size the payload can transmit more real information. Most of the language supports json out of the box.

```
{
 "Users": [
  {
   "name": "John",
   "ssn": "123-456-7890"
  },
  {
   "name": "Maria",
   "ssn": "123-000-4321"
  }
 ]
}
```

Figure 6-5 Sample JSON

6.2.3 Backward Compatibility for Plaintext Logs

Visual logging introduces a new wrapper class which provides methods to format log messages into a Json structure. This works very well as seen in the examples above. However, not all use case may need to employ the json formatting. For example, a code file among many others in a project may want to log just plaintext messages. In yet another scenario, a team may not want to move to the new logging approach because this may not work for them, and changing each logger statement would be a lot of work to do.

```
logger.debug(UTALogger.LogFormatter("Random message for api ",
, new HashMap<String, Object>
  () {{
  put("httpMethod", httpMethod);
  put("statusCode", statusCode);
 }}));

}
```

```
logger.debug("Random message for api");
```

Figure 6-6 Backward Compatibility

Visual logging provides backward compatibility. This means plain old logger statement with a string message without any log formatter implementation would work equally well with this new framework. The log statements shown below would work fine with this logging system.

6.2.4 Resource Field

Visual logging provides structure to a logging information using a schema. These schemas however, cannot be concrete one because logging use case will certainly differ for different project and organization. For example, a component may be interested in logging http related fields like status code, http method, execution time. Another component may be interested in logging fields like user login information, login session duration, user's demographic information etc. This makes it clear that no model can satisfy the schema usage in a generalized manner.

In order to solve this issue, visual logging introduces a "resource" field in logFormatter method which takes inputs as a key-value pair in a hash map. This ensures that framework receives all the information without imposing a strict schema on the user.

6.3 S3 Appender

Visual logging comprises of a data pipeline which not only logs data but also uses them to enable visualization with almost zero engineering. This requires data to be present in a centralized storage such as AWS S3 in this case. Out of the box, logging frameworks such as Log4J and other do not support sending messages to S3. They support logging mechanisms such as console logging, file logging, etc. However, they provide a flexibility to implement the custom implementation for the logging mechanism. User can customize the logging implementation to change message formats, change message bulk size to be logged, change destination to messages to SQL, S3 or any other storage medium possible.

6.3.1 Appender Skeleton Class

Customizing logging mechanism starts with the AppenderSkeleton Class. In java, a new class can be created, S3Appender in this case, which extends from AppenderSkeleton class provided by log4J package. This class has following methods which needs to be override.

- ActivateOptions – This method is invoked the first time a logger object is instantiated. This method can be used to setup initial configuration parameters or other options as required by the implementation. For example,

```
@Override
public void activateOptions() {
   System.out.println("Activating Appender");
   builder = new StringBuilder();
   startInstance(threadsize, fileextension, s3bucketpath, s3accesskey,
s3secretkey, sharedQueue);
   lStartTime =  System.nanoTime();
}
```

- Append – This method takes a parameter of type LoggingEvent which enables the custom class to intercept the logging message coming from the client. This message can be formatted, serialized or processed in any way before sending it to the destination. In the visual logging implementation, a check is added where this method checks for the total size of message in temporary buffer is more than a specified size. Only then the messages will be flushed to the storage. This reduces the number of clutters in the storage and keeps the file chunks consistent in size.

  As shown below, the message is collected using getRenderedMessage() method of LoggingEvent class. Capacity determines size of the shared queue. The if condition check ensures the total size of shared queue at any point of time does not become more than the configured maximum queue size. If the shared queue is not out of the max size, the implementation keeps on adding to the

temporary buffer. The implementation also uses string builder as a temporary storage data structure as compared to string buffers. String builders are faster and thread safe and therefore, they do not need to be synchronized explicitly.

```
@Override
protected void append(LoggingEvent loggingEvnt) {
   String message = loggingEvnt.getRenderedMessage() + "\r\n";
   int capacity = builder.capacity();
   if (capacity * 8 > this.maxsize) {
      try {
         enqueueStream(builder);
         builder = new StringBuilder();
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
   else
   {
      builder.append(message);
   }
}
```

Figure 6-7 Append Method Implementation

- Close – This method takes care of closing the logger instances when all messages has been logged. This message however, may not be called in all the condition. For example, a console application or a desktop application running on a machine has a defined exit scenario. For example, a music player application will be shut down by the user when he is done listening to music. However, in a web application which runs on a server, the application is expected to run indefinitely and serve the requests. In such a case the close method should not be called explicitly and logger instance should end only when the application is exiting.

```java
public void close() {
    if (builder.capacity() > 0) {
        try {
            enqueueStream(builder);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    try {
        instance.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
    finally {
     long lEndTime = System.nanoTime();
     long output = lEndTime - lStartTime;
     System.out.println("Elapsed time in milliseconds: " + output / 1000000);
    }
}
```

Figure 6-8 Overriding Close Method

6.3.2 Singleton Wrapper

As discussed in section 4.1.3, visual logging implementation may lead to numerous threads instantiated and running at the same time. We use a singleton wrapper class to solve this problem. When a client requests the UTALogger class' getInstance method, the implementation checks for any existing instance. If there is an instance already in memory, that instance object is returned to the caller, else a new instance is created, cached and returned to the caller. For any subsequent requests this cached instance serves the purpose. This ensures that client has only one instance of the log wrapper object. Since, threads are spawned at every instantiation of the wrapper class, this solves the problem by having just one instance at any point in time.

6.3.3 Producer Consumer Pattern

Visual logging works by utilizing the producer consumer message sharing pattern at its core. The messages are produced by the parent application which runs on the main thread or the producing thread. The framework spawns several other threads called consuming threads. The number of consuming threads created can be specified as part of the configuration. The communication between producer and consumers take place through a shared queue. Main thread enqueues the messages on the queue, whereas, consumer threads constantly look for any new messages in the queue. If any new message arrives, consumer pick it up and processes the message, else it keeps on waiting.

6.3.4 Graceful Exit of Logging Threads

When the parent application exits and the related consumer threads do not terminate, they become a zombie thread simply eating up resources. This is not an ideal situation for any application and should be avoided. Visual logging uses the concept of poison pills to solve this problem. When an application exits, a shutdown of the logging system is initiated by LogManager.shutDown() method. This makes a call to close() method discussed above. Here, the shared queue is populated by ‘n’ poison pills, where ‘n’ denotes the number of thread. The poison pill can be any unique identifier or simply a null value. When the consumer tries to consume the poison pill the implementation identifies that thread needs to be terminated and calls abort() on the thread.

Chapter 7

Experiments and Results

7.1 Structured Data

The benefits by structuring the log data is the most important claims by visual logging framework. In order to test the effectiveness of the new format, we can use following judgement parameters:

- Readability
- Consumability
- Size of logs

Consider the logs below,

```
{
  "executionTime": "1",
  "datetime": "2017-12-05T23:44:56.468-08:00",
  "apiName": "assignUserProfile",
  "ThreadId": 12,
  "aWSRegion": "us-east-1",
  "message": "Random message for api assignUserProfile",
  "httpMethod": "PUT",
  "statusCode": "400"
}
```

and

```
PUT 400 12 us-east-1 - 1s execution time for api assignUserProfile, message=Random
message for api assignUserProfile
```

Figure 7-1 Structured Data Demonstration

It is clear from the logs that structured log is much more readable and comprehensible. A developer can easily make out meanings from the structured logs as compared to the unstructured one. On the Consumability front, the structured logs are much more efficient as compared to the plain format logs. A consuming service can access the structured

logs easily using key value pairs. This leads to lot less code and efficient performance of logging system. The unstructured logs on the other hand, forces use of a parsing engine which in turn uses complex regex which is not memory efficient and costly operation by time parameter. It may look that the size of structured logs is considerably more than the unstructured one, this may not always be the case. Roughly they may have twice the size increment. This problem too can be solved using more efficient data format like AVRO, where key-value pairs are stored intelligently to avoid repetition of keys which is the primary reason of large log size.

7.2 System Throughput

We use a multithreaded environment to handle pushing logs to S3. This is done to ensure that any processing in the logging system code does not hinder with operations of the parent application. First, It is important to test how many log messages the system is able to handle per second. This is measured by incrementing the threads for each run and finally identifying the optimum thread count. It is important to note that the optimum thread count cannot be a fixed number, but will depend on the system hardware configuration and CPU architecture. The graph below shows the result from this experiment.

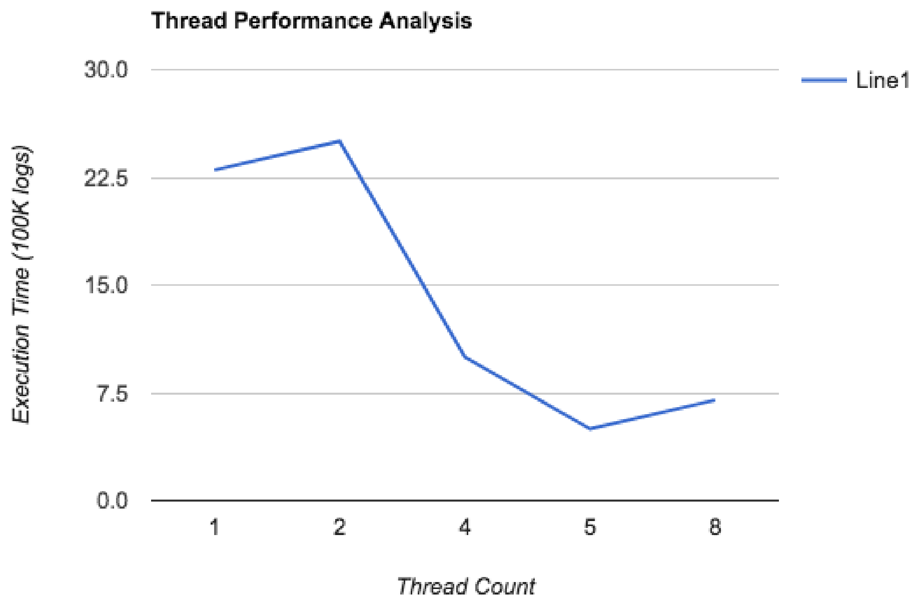

Figure 7-2 Thread Count vs Execution Time

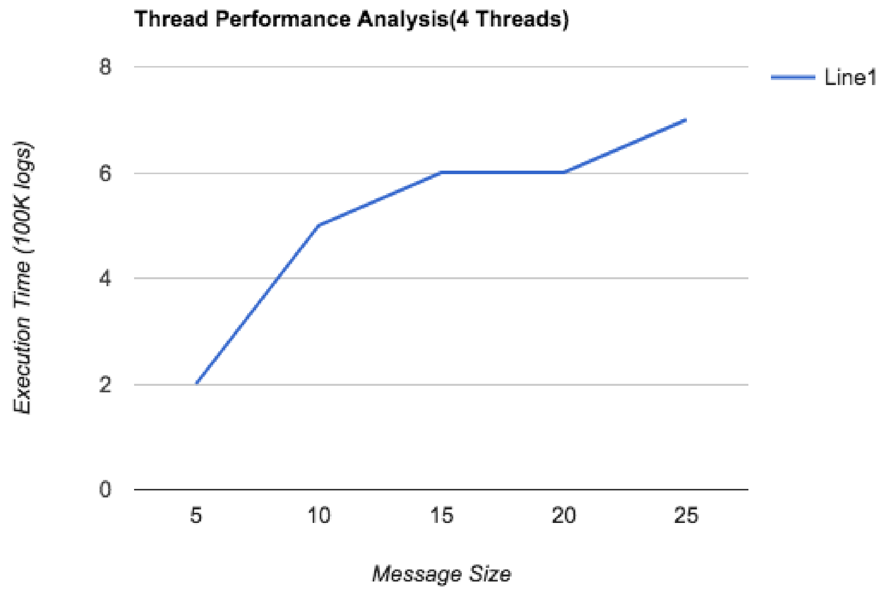

Figure 7-3 Message Size vs Execution Time

7.3 Demonstration of Debugging Efficiency

It is evident from above experiments that structured data makes it possible to comprehend a log message faster. This itself is a huge improvement in amount of time a developer may take to go through the logs and drill down to the system issue. However, this logging system also solves a problem of log stitching. In large application, an execution point starts at a certain line of code but flows through various other methods and api calls to perform the required task. It would be difficult to know which method or api in the entire flow is cause of the issue at hand. This can be solved by adding a common identifier such as “SessionId” in this case to log messages. Using this field, it is possible to gather complete logs of the execution flow and go through only those logs to find the problem. As shown in example below,

```
_source

sessionId: 0199c84a-4461-4b5e-89cb-08dab4552dc8 executionTi
apiName: deleteUser ThreadId: 1 message: Random message fo
_index: http-logs _score:

sessionId: 0199c84a-4461-4b5e-89cb-08dab4552dc8 executionTi
apiName: getProgramContract ThreadId: 1 message: Random me
_type: data _index: http-logs _score:

sessionId: 0199c84a-4461-4b5e-89cb-08dab4552dc8 executionTi
ETE apiName: deleteUser ThreadId: 1 message: Random messag
_index: http-logs _score:

sessionId: 0199c84a-4461-4b5e-89cb-08dab4552dc8 executionTi
apiName: deleteUser ThreadId: 1 message: Random message fo
_index: http-logs _score:

sessionId: 0199c84a-4461-4b5e-89cb-08dab4552dc8 executionTi
T apiName: assignUserProfile ThreadId: 1 message: Random m
_type: data _index: http-logs _score:
```

Figure 7-4 Log Stitching

7.4 Visualization and Dashboards

One of the most important benefits are the visualizations provided by the framework with almost zero engineering from the client side. These visualizations can easily be plugged into a Kibana dashboard or can be exported to use in a custom visualization tool. In the example below, the first graph shows http status code for all requests coming into status by its count. The second line chart visualization shows the successful number of requests per time period or the throughput of a system. Using these visual information, it is very easy to answer questions like:

- How many GET/POST requests were processed each day/hour?
- What is the average number of successful requests every hour? (200 OK)

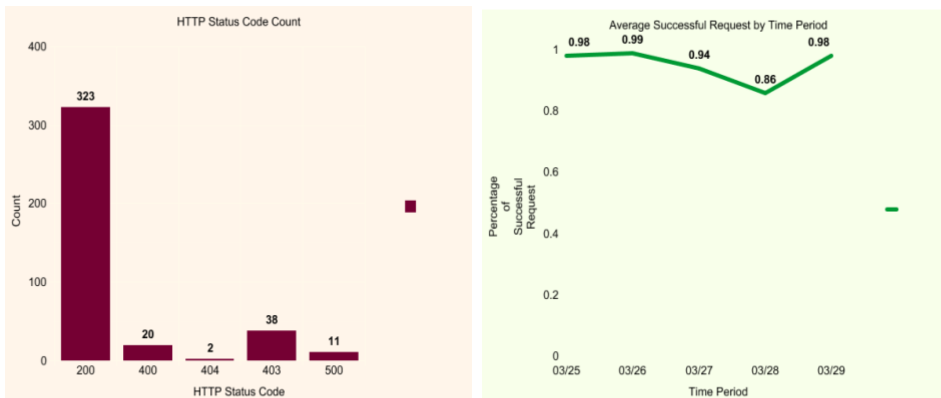

Figure 7-5 Visualizations

7.5 Backward Compatibility

In real world application, backward compatibility plays a very important role. Let's say an exciting upgrade to an application is scheduled to be pushed to existing users. This upgrade may remove existing features and add better ones as part of the upgrade. This however, is not an ideal business decision to take. There may be users having

specific use case related to older functionality which is going to be wiped out. This means company would be losing those customers.

The logging system design that is presented here supports backward compatibility. Consider the examples below which shows two different log statements.

```
logger.debug(UTALogger.LogFormatter("Random message for api ",
, new HashMap<String, Object>
   () {{
  put("httpMethod", httpMethod);
  put("statusCode", statusCode);
 }}));

}
```

```
logger.debug("Random message for api");
```

Figure 7-6 Backward Compatibility Demonstration

User may choose to log using the new LogFormatter static class or may want to stick to traditional logging statements as shown above. This make sense as a project may not have use case to log user data or their log messages are just plain string. Since, there is no use case for new framework, the customer would not upgrade to new ways of logging. In such as case, the logging system should not crash. The visual logging framework takes care of this caveat and handles traditional and new scenarios effectively.

Chapter 8

Conclusion and Future Work

Logging data is one of the most important activities in application development process. Logging not only helps in auditing applications, but their utility ranges from efficient debugging to important analytical decisions which can impact a business operation greatly. Though logging is deemed important part of an application, not much thought process goes into architecting the aspects of a logging system. Only after a deep inspection, one can find out the benefits of structuring data in a right way or using multithreaded environment for logging if data size is huge. Once understood, it takes quite an effort to come up with a clear and efficient solution.

The proposed solution tries to use existing technologies such as Kafka, ELK Stack and AWS S3 and provides a complete data pipeline starting from data production to visualizing data. The process of collecting data, storing it, preprocessing it and restoring it is all covered in some stages of this pipeline. The proposed solution seems to work very well and aligns properly to the specified requirements. However, there are few areas of improvement which can be taken care at a later stage.

One of the problem is data repetition. For example, SessionId is repeated in all the same session logs just to identify all the information for a given session. Though this is quite helpful, it clutters log message with repeated value which also increases log size. Since we use AWS S3 as storage, cutting down on repeated data can save costs. Another problem is posed by complex query writing style of Elasticsearch. ES follows JSON query which is not very intuitive to read and understand when compared to SQL query. Not everyone in an organization apart from developers would like to deal with non-SQL like syntax. It would be great future task to provide users with SQL-ES connecter

where a SQL like query can be parsed to ES query and allows user to fetch data from Elasticsearch database.

References

[1] Alspaugh, Sara, Archana Ganapathi, Marti A. Hearst, and Randy Katz. "Better logging to improve interactive data analysis tools." In *KDD Workshop on Interactive Data Exploration and Analytics (IDEA)*. 2014.

[2] Basak, Jayanta, and P. C. Nagesh. "A User-Friendly Log Viewer for Storage Systems." *ACM Transactions on Storage (TOS)* 12, no. 3 (2016): 17.

[3] Nathan Keegan, Soo-Yeon Ji, Aastha Chaudhary, Claude Concolato, Byunggu Yu, Dong Hyun Jeong, "A survey of cloud-based network intrusion detection analysis", *Human-centric Computing and Information Sciences*, vol. 6, pp. , 2016, ISSN 2192-1962.

[4] Anastopoulos, Vasileios, and Sokratis Katsikas. "A structured methodology for deploying log management in WANs." *Journal of Information Security and Applications* (2017).

[5] Pritom, Mir Mehedi A., Chuqin Li, Bill Chu, and Xi Niu. "A Study on Log Analysis Approaches Using Sandia Dataset." In *Computer Communication and Networks (ICCCN), 2017 26th International Conference on*, pp. 1-6. IEEE, 2017.

[6] Iversen, Morten Aursand. "When Logs Become Big Data." Master's thesis, 2015.

[7] Kaldor, J., Mace, J., Bejda, M., Gao, E., Kuropatwa, W., O'Neill, J., Ong, K.W., Schaller, B., Shan, P., Viscomi, B. and Venkataraman, V., 2017. Canopy: An End-to-End Performance Tracing And Analysis System.

[8] Rodrigues, A. J. "AUTOMATED LOG ANALYSIS USING AI: INTELLIGENT INTRUSION DETECTION SYSTEM." *Computer* 132 (2013): 0886.

[9] Sato, Tatsuya, Yosuke Himura, and Yoshiko Yasuda. "Evidence-based context-aware log data management for integrated monitoring system." In *Network Operations and Management Symposium (APNOMS), 2016 18th Asia-Pacific*, pp. 1-4. IEEE, 2016.

[10] Stoica, Ion, Michael Franklin, Michael Jordan, Armando Fox, Anthony Joseph, Michael Mahoney, Randy Katz, David Patterson, and Scott Shenker. *THE BERKELEY DATA ANALYSIS SYSTEM (BDAS): AN OPEN SOURCE*

PLATFORM FOR BIG DATA ANALYTICS. University of California, Berkeley Berkeley United States, 2017.

[11] Sun, Maoyuan, Gregorio Convertino, and Mark Detweiler. "Designing a Unified Cloud Log Analytics Platform." In *Collaboration Technologies and Systems (CTS), 2016 International Conference on*, pp. 257-266. IEEE, 2016.

[12] FENG C, WANG H, LU N, et al. Log-transformation and its implications for data analysis. *Shanghai Archives of Psychiatry*. 2014;26(2):105-109. doi:10.3969/j.issn.1002-0829.2014.02.009.

[13] Barish, Greg. Scalable and High-Performance Web Applications. Pearson 2005. Web. 25 Oct 2017. http://www.informit.com/articles/article.aspx?p=26942&seqNum=18

[14] Apache ActiveMQ. Apache Software Foundation 2004. Web. 25 Oct 2017. http://activemq.apache.org/clustering.html

[15] O. Baysal, R. Holmes, and M. W. Godfrey. Developer dashboards: The need for qualitative analytics. IEEE Software, 30(4):46–52, 2013

[16] Biyani, Vishal. Log Management as a Service. Site Point. Web. 26 Oct 2017. https://www.sitepoint.com/log-management-as-a-service/

[17] Liebetrau, Etienne. Simple Network Monitoring with Windows Firewall Logging And Reporting. Web Spy. Web. 28 Oct 2017. http://webspy.com/blogs/simple-network-monitoring-with-windows-firewall-logging-and-reporting/

[18] Devang, Suresh. JDBC Architecture. Web. 28 Oct 2017. https://sites.google.com/site/sureshdevang/jdbc-architecture

[19] Kerry Elijah. Data Communication in LabView. LabView. Web. 29 Oct 2017. http://slideplayer.com/slide/3448678/

Biography

Ravi Nishant received his bachelor's degree in Computer Science from R.V College of Engineering, affiliated to Visveswaraya Technological University, Belgaum, India in 2010. He worked in CSC, Newgen Technologies, NVIDIA and Adobe as a software engineer for a period of six years.

In 2015, he started his masters in computer science at the University of Texas at Arlington. He worked as web developer assistant for UTA Information Security Lab and worked as voluntary research assistant in the field of information security and anonymity. His research interests include Big Data, Machine Learning and Data Analytics.