Designing and Developing a Robust Automated Log File Analysis Framework for Debugging Complex System Failure

Presented by:
Tyrone Jade van Balla

A dissertation submitted in partial fulfilment of the requirements for the degree of

Master of Science in Engineering

in the Department of Electrical Engineering,
Faculty of Engineering and the Built Environment
at the

University of Cape Town

November 1, 2021

Supervisor:
Dr Simon Winberg

Plagiarism Declaration

I know the meaning of plagiarism and declare that all the work in the document, save for that which is properly acknowledged, is my own. This thesis/dissertation has been submitted to the Turnitin module (or equivalent similarity and originality checking software) and I confirm that my supervisor has seen my report and any concerns revealed by such have been resolved with my supervisor.

Tyrone van Balla
November 1, 2021

Abstract

As engineering and computer systems become larger and more complex, additional challenges around the development, management and maintenance of these systems materialize. While these systems afford greater flexibility and capability, debugging failures that occur during the operation of these systems has become more challenging. One such system is the MeerKAT Radio Telescope's Correlator Beamformer (CBF), the signal processing powerhouse of the radio telescope.

The majority of software and hardware systems generate log files detailing system operation during runtime. These log files have long been the go-to source of information for engineers when debugging system failures. As these systems become increasingly complex, the log files generated have exploded in both volume and complexity as log messages are recorded for all interacting parts of a system. Manually using log files for debugging system failures is no longer feasible.

Recent studies have explored data-driven, automated log file analysis techniques that aim to address this challenge and have focused on two major aspects: log parsing, in which unstructured, free-form text log files are transformed into a structured dataset by extracting a set of event templates that describe the various log messages; and log file analysis, in which data-driven techniques are applied to this structured dataset to model the system behaviour and identify failures. Previous work is yet to address the combination of these two aspects to realize an end-to-end framework for performing automated log file analysis.

The objective of this dissertation is to design and develop a robust, end-to-end Automated Log File Analysis Framework capable of analysing log files generated by the MeerKAT CBF to assist in system debugging. The Data Miner, Inference Engine and the complete framework are the major subsystems developed in this dissertation. State-of-the-art, data-driven approaches to log parsing were considered and the best performing approaches were incorporated into the Data Miner. The Inference Engine implements an LSTM-based multi-class classifier that models the system behaviour and uses this to perform anomaly detection to identify failures from log files. The complete framework links these two components together in a software pipeline capable of ingesting unstructured log files and outputting assistive system debugging information.

The performance and operation of the framework and its subcomponents is evaluated for correctness on a publicly available, labelled dataset consisting of log files from the Hadoop Distributed File System (HDFS). Given the absence of a labelled dataset, the applicability and usefulness of the framework in the context of the MeerKAT CBF is subjectively evaluated through a case study.

The framework is able to correctly model system behaviour from log files, but anomaly detection performance is greatly impacted by the nature and quality of the log files available for tuning and training the framework. When analysing log files, the framework is able to identify anomalous events quickly, even when large log files are considered. While the design of the framework primarily considered the MeerKAT CBF, a robust and generalisable end-to-end framework for automated log file analysis was ultimately developed.

Acknowledgements

First and foremost, I would like to thank Dr Simon Winberg, my supervisor, for his patience, insight and advice in guiding me to complete this research project. In addition, I would like to extend a special thanks for his time and willingness to thoroughly review certain aspects of my work even in the final days before submission.

I would like to thank the South African Radio Astronomy Observatory (SARAO) and the National Research Foundation for funding this research.

In addition, I would like to thank the following people, persons and groups for their contributions, in any shape or form, toward the completion of this research project:

Dr Jason Manley, my former manager at SARAO, for allowing me the opportunity to use the MeerKAT Correlator Beamformer as the case study system for my research.

My colleagues at SARAO for being the bunch of people that they are: always inspiring, supportive, encouraging and just generally-all-around-great people.

My parents for laying the strong foundation during my upbringing, and instilling and shaping the qualities and values that I hold. Without this foundation, I would not be where I am today.

And last, but certainly not least, my wife, Ayesha: who put up with me and all my moods as I completed this project. For being the kind, caring and supportive individual that you are. For believing in me even in the times I stopped believing in myself. And for the continuous supply of unhealthy amounts of caffeine in the last few days before submission!

Contents

List of Figures

List of Tables

Glossary

ALFAF: The Automated Log File Analysis Framework designed in this dissertation. A robust, end-to-end framework that performs automated log file analysis to detect failures that occur during system runtime.

AEL: Abstraction Execution Logs. A log parsing algorithm that employs a combination of rules-based and clustering approaches

CASPER: Collaboration for Astronomy Signal Processing and Electronics Research

CBF: Correlator Beamformer. A system of the MeerKAT Radio Telescope responsible for performing the digital signal processing required for an interferometer

CMC: Correlator Master Controller. The master controller node in the CBF

Component: A systems engineering terms that refers to any software, hardware or interface of a system

CPU: Central Processing Unit

CSV: Comma Separated Values

Data Miner: A subcomponent of the ALFAF that performs the automated parsing of log files into a structured dataset

Debugging: A routine process of identifying, locating and resolving bugs and issues in a software or hardware system

Downtime: Periods of time when system operation is unavailable

Drain: A clustering-based log parsing algorithm that makes use of parsing trees

F1 Measure: A classification accuracy metric that is the harmonic mean of the achieved precision and recall

False Negatives: Anomalous events that are misclassified as being normal

False Positives: Normal events that are misclassified as being anomalous

FPGA: Field-Programmable Gate Array

Gbps: Gigabits per second

GPU: Graphics Processing Unit

HDFS: Hadoop Distributed File System

HMC: Hybrid Memory Cube

HPC: High Performance Computing

IDE: Integrated Development Environment

Inference Engine: A subcomponent of the ALFAF that performs automated log file analysis to detect system failures from log files

IPLoM: Iterative Partitioning Log Mining. A log parsing technique that performs three stages of partitioning to extract event templates

LenMa: Length Matters. A log parsing technique that considers the length and position of the words or tokens occurring in log messages

LFA: Log FIle Abstraction. A frequent patterning mining-based algorithm similar to SLCT

LKE: Log Key Extraction. A clustering-based log parsing technique

Log File: A file that records events that occur during system operation

Log File Analysis: A process in which log files generated by systems are reviewed to extract information about the system's behaviour during runtime

LogCluster: A log parsing technique that employs frequent pattern mining

LogMine: A hierarchy-based clustering algorithm for performing log parsing

LogSig: A clustering-based log parsing technique

LSTM: Long Short-Term Memory Neural Network. An enhanced architecture of Recurrent Neural Networks capable of modelling long range dependencies across sequences

MeerKAT: "More" Karoo Array Telescope. A 64 Antenna Array Radio Telescope built by the South African Radio Astronomy Observatory

Model Accuracy: Metric describing the classification or prediction performance of a machine learning or deep learning algorithm. Defined as the ratio between the total number of correct predictions and the total number of predictions made

MoLFI: Multi-objective Log message Format Identification. A search-based log parsing technique

NumPy: A Python library that adds support for efficient computation on multi-dimensional arrays

pandas: A Python library that provides features for data analysis and manipulation

Parsing Time: The time taking for a log parsing algorithm to extract event templates from a given log file

PDU: Power Distribution Unit

Precision: Metric that describes how much of the predicted positives were correctly classified

Python A high-level and interpreted programming language

PyTorch: An open-source deep learning library for Python

Recall: Metric that describes how much of the actual positives were identified correctly

RNN: Recurrent Neural Network. A neural network architecture that has memory and is capable of modelling sequences

SARAO: South African Radio Astronomy Observatory

scikit-learn: An open-source machine learning library for Python

SciPy: A Python library that supports scientific and technical computing

SHISO: A log parsing technique that uses incremental pattern mining

SKA: Square Kilometre Array

SKARAB: Square Kilometre Array Reconfigurable Application Board

SLCT: Simple Log Clustering Tool. A log parsing technique based on frequent pattern mining

Spell: A log parsing algorithm that extracts event templates by identifying the longest common subsequence across sequences

System:A systems engineering term that refers to an arrangement of interacting components organized to perform a function

Systems Engineering: An approach to system design that aims to enable the realisation of a successful system through the transformation of user requirements into a suitable solution

System Failure: An undesirable state during system operation. May manifest as a loss of operational capability or degraded performance and usually causes system downtime

True Negatives: Normal events that are correctly classified as normal

True Positives: Anomalous events that are correctly classified as anomalous

UML: Unified Modelling Language. A modelling language used to visualize the design of software systems

YAML: YAML Ain't Markup Language. A data-serialization language typically used for configuration files

1

Introduction

Engineering systems have grown in complexity over the years in direct response to the increasing complexity of the problems that engineers have been required to solve. Software systems have evolved into distributed applications, capable of leveraging increasing processing capability and the advent of micro-services has seen software systems being made up of many interacting components of which the sum of the whole is greater than its individual parts. Hardware systems have evolved similarly, both in raw processing capability, and the complexity of the systems deployed. It is now commonplace for typical hardware back-ends to consist of a combination of off-the-shelf servers housing accelerator cards and high-speed network interface cards, alongside custom-designed, application specific components and processing nodes [1]. To keep up with the increased processing capability, data transmission systems and networks have also seen numerous enhancements over the years. Entire systems, themselves as a whole, now consist of multiple interacting and interconnected subsystems, each performing a function vital to the system itself.

While this increase in the capability of engineering systems has been welcomed, it has also introduced new challenges in terms of managing, maintaining and ensuring the continued operation of such systems. One area that is particularly challenging is the debugging of failures, during runtime, within complex engineering systems. Typically, log files produced by hardware and software have been the first point of call for any runtime failure debugging activity. This process is commonly known as Log File Analysis [2]. These log files record events that occur during system operation and can offer insight into system health and performance and provide a record of the events that occurred before, during and after a runtime failure [3]. However, as engineering systems increased in complexity, the volume of log files produced by these systems increased exponentially. This increase in the volume of log files has made Log File Analysis, and the process of debugging system failure, even more challenging [4].

An example of such a complex engineering system is the MeerKAT Radio Telescope's Correlator Beamformer (CBF). This system implements digital signal processing algorithms for radio astronomy.

The main objective of this research project is to design and develop a robust, end-to-end Automated Log File Analysis Framework that is capable of automating the process of Log File Analysis with the aim of identifying failures during system runtime, and to evaluate the performance and usefulness of this framework on the CBF system. The development of this framework will aim to leverage the increased volume of log files by considering and implementing data-driven approaches to Log File Analysis.

While the developed framework is designed to primarily be evaluated on the CBF system, the robustness and generalisability of the framework to other systems is considered as a secondary objective to influence the design of the framework.

This chapter introduces the background to the problem, the research focus, and finally provides an overview of the rest of the dissertation.

1.1 Background to the Study

The South African Radio Astronomy Observatory (SARAO) designed, built and is currently operating and maintaining the MeerKAT Radio Telescope - a 64 antenna array radio telescope located in the Karoo in the Northern Cape in South Africa. This radio telescope was built as a technology demonstrator and serves as a precursor to the mid-frequency component of the Square Kilometre Array (SKA) Phase 1 Radio Telescope [5].

The MeerKAT Radio Telescope is an example of a complex engineering system. It consists of multiple, unique subsystems, each contributing to the overall performance and functionality of the telescope as a whole. There are numerous interfaces and interactions between the various subsystems and it is this collection of interactions between many individual subsystems that results in the complete, complex system.

The Correlator Beamformer (CBF), one of the subsystems of the MeerKAT Radio Telescope, is the system of interest in this research project. The CBF subsystem forms part of the digital back-end of the telescope and is the digital signal processing powerhouse of the telescope. It is responsible for performing various signal processing functions for radio astronomy such as channelisation, cross-correlation, accumulation, phase-shifting and summation[6]. The CBF is itself a complex system and comprises commercial-off-the-shelf servers, network switches, Ethernet interconnect, custom-designed FPGA-based processing nodes, power distribution units, and various software packages, firmware and gateware developed to run on the physical hardware. It is the interaction of all these components that enables the CBF to deliver its primary capability of performing real-time digital signal processing for the MeerKAT Radio Telescope.

As the CBF consists of many interacting and complex hardware and software components, when system failures occur, localising the fault and debugging the failure often proves challenging and time consuming. The CBF, as well as other subsystems of MeerKAT, was designed to report and log its operation and health status during runtime. These generated log files can often contain insights to the cause of system failures and are briefly investigated by engineers and developers when attempting to debug a system failure. While these log files are useful for the

purposes of debugging, the scale of the system has rendered manual Log File Analysis infeasible. Currently, the CBF consists of two off-the-shelf servers, 288 FPGA-based processing nodes and 54 network switches, and during typical operation reports on, and logs, close to 11000 unique sensors. This increase in volume of log files also increases the time and cost associated with locating and debugging system faults and failures. Ensuring the continued operation of the CBF and meeting reliability and uptime requirements, then, by quickly and efficiently localising faults and debugging failures, is a challenging and non-trivial task.

1.2 Research Focus

This research project focuses on addressing the challenges associated with performing Log File Analysis on the logs produced by large-scale complex systems, such as MeerKAT's Correlator Beamformer, through the design and development of an end-to-end framework capable of performing Log File Analysis automatically to detect failures and provide insights to assist in the debugging of system failures. While the MeerKAT CBF is the primary system of interest, this research project aims to design a robust and generalisable framework for automated Log File Analysis in complex systems, such that it may be adapted for future generations of the CBF and also the other subsystems of the MeerKAT Radio Telescope.

Preliminary research led to the understanding that automated Log File Analysis consists of two major processes. Firstly, the parsing of unstructured log files into a structured dataset and then secondly, using this structured dataset to perform data-driven analysis techniques to extract the desired information from the log files. Recent studies have considered the two processes separately, but the combination of the two into a single pipeline or framework is yet to be addressed.

The objective of this dissertation is to address a number research questions concerning the design, development, implementation and operation of a data-driven, robust, end-to-end automated log file analysis framework for detecting system failures. This framework will implement the two processes required for Log File Analysis described above, through the design and implementation of two subcomponents, namely the Data Miner and the Inference Engine, and the overarching framework. While the MeerKAT CBF is used as a primary case study system to provide necessary context for the development of the framework, the generalisability of the framework and its applicability to other systems is used to influence its design. Addressing these questions involved stages of research, functional prototype development, testing and evaluation on a real-world system.

Some of the research questions inform high-level system requirements and design considerations that are incorporated into the development of a functional prototype. These considerations and requirements are further detailed in Chapters 4, 5, and 6 in which the design and development of the framework and its subcomponents is described.

The research questions put forward in this dissertation, and that chapters in which they are considered and addressed, are presented as follows:

- **Research Question 1:** *What data-driven log parsing techniques can be used to build a Data Miner capable of parsing raw, unstructured log files into a structured dataset? How well do these techniques perform?*

  In recent years, much progress has been made in the field of automated log parsing. Many of the most prominent techniques have been collectively evaluated and benchmarked, and have implementations available in an open-source tool known as Logparser[7]. This dissertation will use Logparser as the basis of development for the Data Miner subcomponent. Log parsing techniques are introduced in Chapter 3, but are analysed in more detail in Chapter 4 as the design of the Data Miner is considered. Once implemented, the various log parsing techniques are evaluated in terms of their performance, using the Data Miner, in Chapter 7.

- **Research Question 2:** *Which machine learning and/or deep learning technique is best suited for an Inference Engine to perform failure detection using the log files generated by a system? How well does the Inference Engine perform?*

  Initial research has shown that various machine learning techniques are quite adept at failure detection and the various techniques consider different approaches when applied to the analysis of log files. Chapter 3 discusses state-of-the-art machine learning techniques that have been applied to the problem of log file analysis with the aim of determining the best suited technique. Following this, Chapter 5 details the design and development of the Inference Engine which implements the chosen machine learning technique for log file analysis. Finally, Chapter 7 considers the performance of the Inference Engine and the chosen technique.

- **Research Question 3:** *How can the Inference Engine be tailored to generate assistive debugging information to aid in the debugging of detected failures? How can the Inference Engine be given a degree of interpretability?*

  Machine learning models are often criticised for being *black boxes* and the argument can be made that in the context of failure detection and debugging, some degree of interpretability of the model is desired. In Chapter 5, while considering the design of the Inference Engine, the output of the Inference Engine is carefully considered and designed to contain information that can be translated back into human-interpretable information. Chapter 6, in which the design of the overall framework is considered, describes how the Inference Engine output is translated to generate outputs that can assist in failure debugging.

- **Research Question 4:** *How can the Data Miner and Inference Engine be combined into an automated, end-to-end, log file analysis-based failure detection framework that is capable of ingesting raw system log files and outputting assistive debugging information?*

  The Data Miner and Inference Engine subcomponents are designed as modular, stand-alone components capable of performing their respective tasks and being evaluated independently. The design and development of an automated, end-to-end log file analysis framework, consisting of the Data Miner and Inference Engine combined in a pipeline, is

the main objective of this dissertation. Chapter 6 describes the design and development of this framework and also details the interfaces required between the Data Miner and Inference Engine, as well as the system under test. The operation and acceptance of the framework is considered in Chapter 8 in which the framework is put to work on a real-world system.

- **Research Question 5:** *What requirements or guidelines can be imposed on the generation of system log files to make them more amenable to automated log file analysis using the developed framework?*

  Log files generated by systems are inherently unstructured and the content of the log files is often decided by the system developers. Chapter 8 explores the performance of the developed framework on a real-world system and discusses what impact the structure and content of the log files themselves have on failure detection performance. In Chapter 9, these findings are then used to formulate requirements or guidelines for the generation of log files to ensure optimal performance of the developed framework.

- **Research Question 6:** *What is the predicted impact of this framework on the process of debugging system failures, system uptime, operation, reliability and usability in the context of the MeerKAT Correlator Beamformer? How can the performance of this framework be tested at scale?*

  Running the developed framework alongside the MeerKAT Correlator Beamformer and performing extensive testing is outside the scope of this project. Instead, Chapter 8 subjectively evaluates the performance of the framework in the context of the MeerKAT CBF and Chapter 9 provides insights for further refinement and testing.

1.3 Scope and Limitations

The scope and limitations were established and agreed upon after consultation with a number of stakeholders including the supervisor of this research project and lead engineers working on the development of the MeerKAT Correlator Beamformer. The scope is used to focus the research project such that the main objectives are accomplished. The limitations are established to define what this project will not consider. These are presented in the subsections that follow.

1.3.1 Scope

This research project shall only consider:

- the MeerKAT Correlator Beamformer as the primary system of interest to establish the project context,
- the design and development of a Data Miner for performing automated log file parsing,
- the design and development of an Inference Engine for performing deep-learning-based log file analysis,

- the integration of the Data Miner and Inference Engine to realise a complete, end-to-end Automated Log File Analysis Framework,
- testing and verifying the performance of the framework on the MeerKAT Correlator Beamformer,
- the generalisability of the framework and its subcomponents to the extent that influences the design.

1.3.2 Limitations

The following project limitations are established:

- due to time and operational constraints, the developed framework shall not be deployed to run alongside the MeerKAT Correlator Beamformer to evaluate its performance on the system running in real-time,
- while online processing of log files, for both log parsing and log file analysis, may be considered, this developed framework shall only be required to process log files offline and in batches,
- while framework generalisability is a secondary objective, the developed framework and its subcomponents shall only be tested and evaluated in terms of its performance and operation on log files generated by the MeerKAT Correlator Beamformer.

1.4 Overview of Dissertation

This section provides an outline of the remaining chapters of this dissertation.

Chapter 1 introduced the topic of this research project and provided the necessary background and context. It also detailed the research questions to be addressed and provides an overview of the entire project.

Chapter 2 outlines the methodology employed during this research project to investigate and answer the research questions that were posed, and to design and develop the end-to-end automated log file analysis framework.

Chapter 3 presents a literature review in which the context of this research is established. It also introduces the MeerKAT Correlator Beamformer as the case study system, reviews and critiques current solutions to the problem of log file analysis and identifies suitable candidate solutions and techniques for the implementation of the Data Miner, Inference Engine and overall framework. It then develops the theory behind the chosen techniques as required and explores tools and frameworks that may be employed for the design, development and evaluation of the framework and its subcomponents.

Chapters 4, 5 and 6 address the detailed design component of this dissertation. Chapter 4 considers the design of the Data Miner subcomponent. It explores the state of the art log parsing algorithms and describes the design of the stand-alone Data Miner component for

implementing log file parsing. Chapter 5 details the design of the Inference Engine, using the chosen deep learning techniques, and discusses how it models the problem of system failure detection from system log files. Finally, Chapter 6 considers the integration of the Data Miner and Inference Engine subcomponents to realise the complete end-to-end, automated log file analysis framework. All developed software is open-source and stored in a publicly accessible GitHub repository available at: `https://github.com/tyronevb/alfaf`

Chapter 7 describes the algorithmic tuning of the Data Miner and Inference Engine and evaluates and verifies the implementation of these components independently using a labelled dataset. Chapter 8 puts the developed framework to work on a real-world system through a case study in which the operation, performance and implementation of the framework is validated in the context of the MeerKAT CBF. Following on from the results presented in Chapter 7 and the case study, Chapter 8 then also analyses the performance, implementation and operation of the entire system.

Chapter 9 concludes the dissertation by providing answers to the research questions, assessing whether the main objective was met and then providing recommendations for future research that may be conducted.

2

Methodology

This chapter describes the methodology for this research project, covering the phases from the initial concept formulation through to the processes used to investigate and respond to the research questions that were posed in Chapter 1. Figure 2.1 overviews the entire methodology followed during this investigation. Three major phases were identified for this project: *Concept Exploration*, *System Design* and *System Verification & Analysis*. These phases are described in detail in the sections that follow.

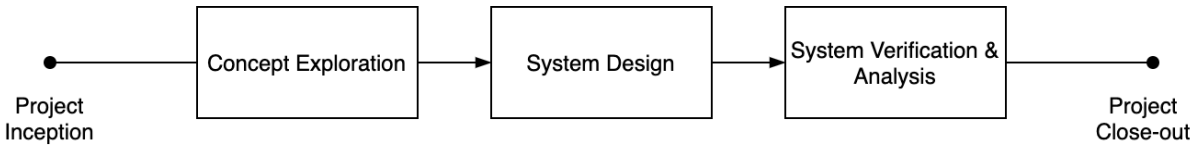


Figure 2.1: An overview of the methodology and the various phases followed to develop this project

A tailored systems engineering approach [8] is used to guide the development of this research project throughout the different phases. A systems engineering approach is adopted to ensure that the project is well-defined and that the problem space is well understood before any serious development effort is expended in building the system. This approach also mitigates risks associated with the design of the system as it encourages detailed design definitions and interface definitions to be completed before serious implementation takes place. This approach is influenced by the fact that changes made later in the development life-cycle of a system incur a greater cost than changes made earlier on.

It should, however, be noted that the systems engineering approach is only used to guide the development of this research project to answer the research questions put forward. The delivery of a formally realised and validated system is not within the scope of this project.

The system under development during this dissertation is a software-based system and parts of this system also encompass the implementation of data-driven, deep learning models. To facilitate the development of such a system, software design methodologies, tailored to building

deep learning applications, are used to guide the development processes where applicable. The detailed software design methodologies used are further described in Section 2.2.

2.1 Phase 1: Concept Exploration

The *Concept Exploration Phase* comprises a number of processes that aim to achieve a proper and thorough understanding of the problem space and context before conceptualizing and assessing viable solutions. Figure 2.2 details the various processes of this phase. The input to this phase is a problem description, captured in the project proposal.

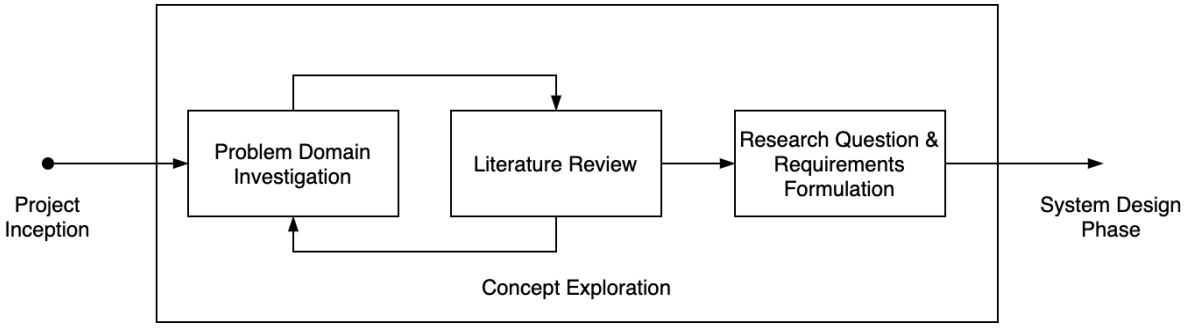


Figure 2.2: Diagram illustrating the Concept Exploration Phase and its processes

This phase begins with a problem domain investigation during which the problem, the reason for this research, is investigated alongside its context. This process considers the Correlator Beamformer of the MeerKAT Radio Telescope and the operation, debugging and fault-finding procedures thereof. It aims to understand what system information data is generated by the Correlator Beamformer during runtime and investigates why current methods of debugging and fault-finding leave more to be desired.

Following an initial investigation into the problem space, a literature review is conducted to bolster the understanding of the context and to investigate alternative solutions to addressing this problem. During the process of the literature review, based on findings, and questions that are generated, additional iterations of problem domain investigation take place to determine the applicability and potential usefulness of certain solutions. These two processes repeat until sufficient information is available to inform the next process in this phase.

As part of conducting the literature review, a critical analysis of similar work is undertaken to formulate ideas for the design of the solution system and to better understand some of the challenges that may be faced during implementation. Finally, the literature review then also considers the various implementation details associated with building the proposed system i.e. software languages, profiling tools, frameworks, environments, as well as metrics for evaluating the performance and behaviour of the system.

After multiple iterations of the literature review and problem domain investigation processes, the gathered information is then used to define the research focus of this project and formulate the key driving research questions. These research questions are put forward in such a way as to drive the development of the research project toward achieving its primary objective as outlined

in Section 1.2. During this process, the project scope and limitations are also identified.

Alongside the formulation of the research questions, high-level system requirements and design considerations for the solution system are derived. These requirements and considerations are used to further guide development of a system capable of addressing the research questions. It should be noted that not all derived system requirements will be verified or validated during this project.

The research questions and high-level system requirements and considerations that are formulated during the *Research Question & Requirements Formulation Process* are used to guide system design and development in later phases. Where applicable, this process is guided by Systems Engineering Practices to ensure that requirements are well-defined and unambiguous. Thought is also given to the verification of these requirements and this information informs a later phase in the overall methodology as described in Section 2.3.

2.2 Phase 2: System Design

The *System Design Phase*, as outlined in Figure 2.3 below, includes processes that culminate in the design and development of a system, as guided by the derived system requirements, that can be used to answer the research questions.

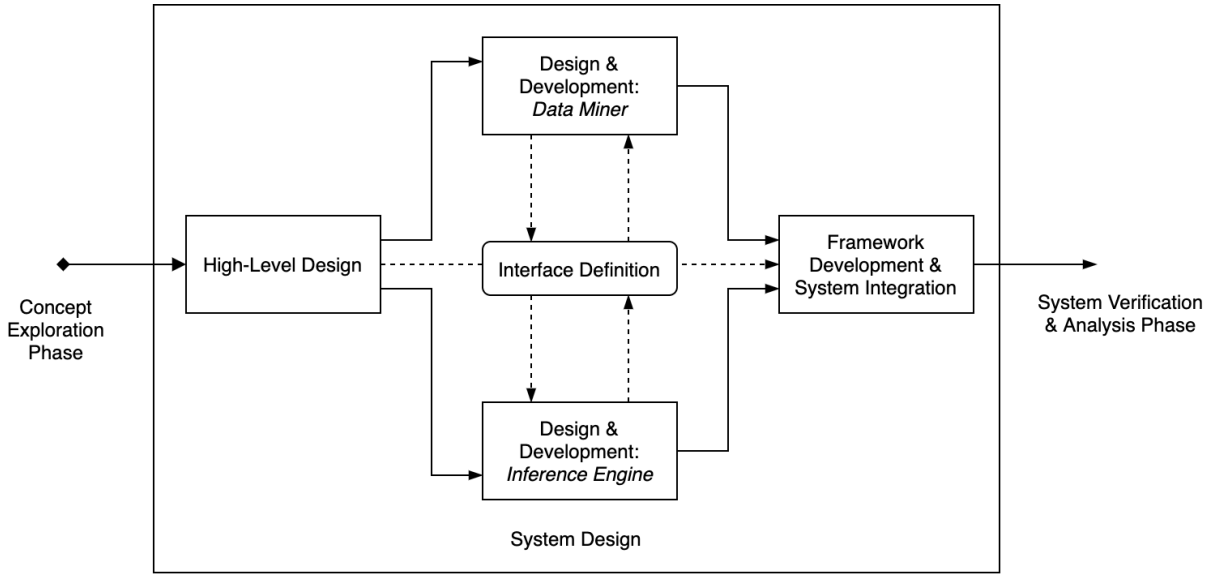


Figure 2.3: Diagram illustrating the System Design Phase and its processes

The research questions and high-level system requirements and design considerations, along with information gathered during the *Literature Review* and *Problem Domain Investigation Processes*, are used to conceptualize the high-level design of the system. This design identifies the major components, or sub-systems, of the system that require further detailed design and also begins to consider the interfaces between these components. The high-level design also considers the application system of interest in its intended operational context and environment i.e. the MeerKAT Correlator Beamformer. During this process, the identified system requirements and design considerations are also allocated to the identified sub-system elements to further guide

development where applicable.

2.2.1 Detailed Design & Development

Following the high-level design, allocation of system requirements and identification of interfaces, *Detailed Design & Development Processes* commence for each of the identified sub-system elements. In this project, the sub-systems to be further designed and developed are the *Data Miner* and *Inference Engine*. This process is expanded in Figure 2.4.

During this process, each sub-system, or component, is designed in full detail. In this project, the design process consists of creating a high-level design for each component, informed by the higher-level system requirements and design considerations. Thereafter, the high-level design of each component is expanded to enable the development of a prototype for each subsystem. The development of these prototypes takes place in the *Development Sub-process* as shown in Figure 2.4, and comprises *Software Development* and *Testing Processes*.

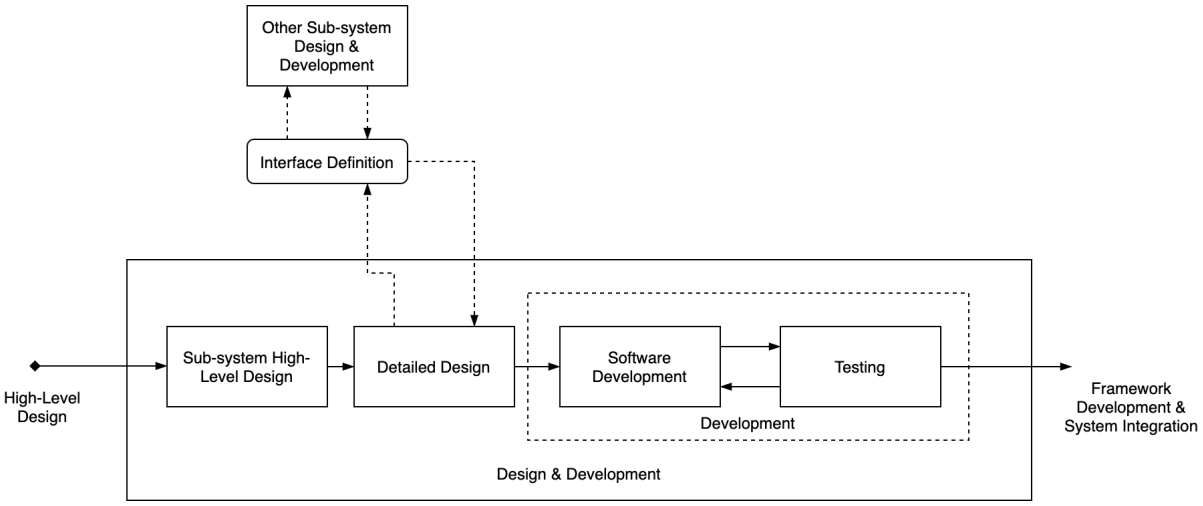


Figure 2.4: Expansion of the Design & Development Process. This process has various sub-processes that guide the design and development of each sub-system element

As the system, and its components, to be designed and developed is software in nature, development consists of software development and testing processes. Given that certain components of the software to be developed implement data-driven, deep learning models and techniques, an Agile-based software development methodology is used to guide development. This methodology allows for flexibility in terms of revising system requirements and, to an extent, the system design as and when necessary. This method also suits the iterative approach that the designing and building of deep learning models typically requires. As shown in Figure 2.4, the processes of software development and testing occur in a cycle. When building deep learning applications, testing procedures inform further development of the software model, to yield better results and this refined model then undergoes further testing. This is repeated until a system that yields optimal performance is developed. Additional detail regarding the specific software development methodology and processes employed, for each sub-system, is presented in the relevant design chapters.

While system design is under way, the detailed design of each sub-system informs the *Interface Definition Process* that occurs between interacting elements, and is also informed by this same process. Initial sub-system design, therefore, needs to occur concurrently for all sub-systems as all sub-systems elements need to be sufficiently designed such that their designs can adequately inform and guide interface decisions. Thereafter, once the *Interface Definition Process* is completed, further sub-system design and development of each component may happen in isolation.

At the end of the *Design & Development Processes*, the implemented components are staged for later system integration.

2.2.2 Framework Development & System Integration

Once all sub-system elements have been designed, developed and tested, the *Framework Development & System Integration Process* sees these elements being combined into a pipeline to realise the overall system of interest. During this process, additional development may be required to combine all software sub-systems and enable interfacing with the external system. Once the system is integrated and the framework developed, a rudimentary end-to-end test is performed on the entire system to verify that it accepts the appropriate inputs and produces the expected outputs along each stage in the data path.

Finally, this process also concerns the informal verification and testing of the interface(s) between sub-systems. After system integration, the system is ready to undergo verification, performance testing and analysis.

2.3 Phase 3: System Verification & Analysis

The *System Verification & Analysis* phase involves processes that analyse the performance of the system and its components after running through various tests and verifies a subset of the system requirements. This process has three primary activities:

- *Verify* the design of the system against system requirements,
- *Test* various aspects of the system to inform later analysis,
- *Analyse* the system's performance in the context of the research questions.

The processes that perform these activities are shown in Figure 2.5.

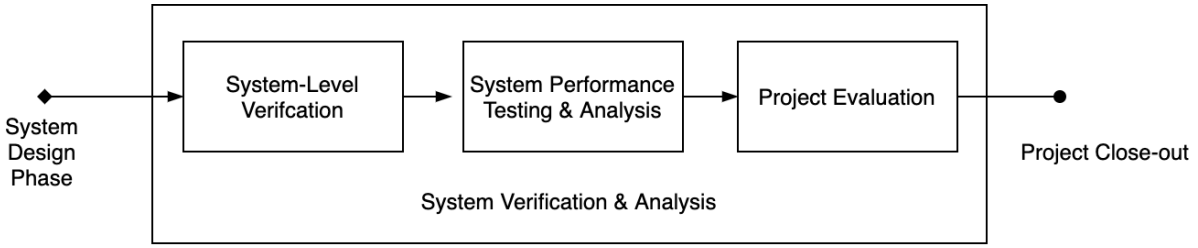


Figure 2.5: Diagram illustrating the System Verification & Analysis Phase and its processes

During the *System-Level Verification Process* the designed system is qualified to determine whether it satisfies a subset of the system requirements and to determine whether the system was designed and developed correctly. Again, it should be noted that not all system requirements will be verified as the primary objective of this project is to answer the research questions. This process may be informed by sub-system level tests and verification.

After performing system-level and component-level verification, the system is put through a series of tests designed to ascertain the performance and behaviour of the system in the context of this research project. These tests are executed and the results are analysed and evaluated as part of the *System Performance Testing & Analysis Process*. This process considers various metrics of the system that enables objective evaluation of the system. These metrics are detailed in the relevant design chapters of the sub-systems and in Chapter 7.

In addition to the execution of performance tests, a case study is also conducted during the *System Performance Testing & Analysis Process*. In this case study, the designed system is put to work on the MeerKAT Correlator Beamformer to evaluate the performance, operation and usefulness of the system in this context. The results of this case study are analysed as part of the primary objective of this project.

The findings from this process of testing and analysis are then used to formulate answers to the research questions posed in Section 1.2.

Finally, the overall project is evaluated and this research culminates with conclusions drawn from the results and from the system performance, and provides recommendations for improvement and further areas of research. At this stage, the project is completed and closed-out.

2.4 Methodology Conclusion

This section detailed the methodology followed in this research investigation. The processes followed in formulating the research questions and deriving the system requirements is outlined. The systems engineering processes followed in taking the system from concept to design, based on a set of guiding requirements, is also presented. Special mention is made about interface management, software development processes and the verification and testing of sub-systems. Finally, the processes followed to ultimately to answer the research questions put forth are presented.

3

Literature Review

This chapter reviews relevant literature surrounding the problem, context, potential solution architectures as well as tools to use for implementing the solution.

The objectives of this literature review are as follows:

1. Define the context of this research project in terms of the problem of debugging large, complex systems using log file analysis
2. Introduce the MeerKAT Correlator Beamformer as the case study system for this research project
3. Review and critique solutions to the problem of log file analysis as described in current literature
4. Identify potential solutions for further investigation and evaluation in this research project
5. Investigate tools and frameworks available for designing, implementing and evaluating potential solution candidates

Section 3.1 defines the context for this research project by introducing complex systems and discussing the implications and challenges associated with debugging complex system failure. The concept of Log File Analysis and its usefulness as a tool to debug systems is presented, alongside the increasing challenges currently being faced in the context of complex systems. The MeerKAT Radio Telescope's Correlator Beamformer Subsystem is introduced as the case study complex system for this research project and considerations for integrating the developed framework with this system are discussed. This section motivates the need for an automated solution to the problem of log file analysis.

Following this, current academic and research literature is surveyed to explore potential candidate solutions for automating Log File Analysis in Section 3.2. The process of Automated Log File Analysis is de-constructed into smaller components, namely Log Collection, Log Parsing, Feature Engineering and finally Log Analysis, and these components are detailed in the order they appear in the processing pipeline. Literature is reviewed and critiqued with the objective of identifying suitable solutions, to the problem of performing log file analysis for complex sys-

tems, for further prototyping and evaluation in this research project. In this research project, emphasis is placed on data-driven solutions employing Machine Learning and Deep Learning. Various objectives of Log File Analysis are also introduced. This research project considers using Log File Analysis for Anomaly Detection in the move toward automated system debugging and therefore mainly focuses on Anomaly Detection.

Section 3.3 considers the various tools and frameworks that are available to implement and evaluate the data-driven solutions for log file analysis and construct the Automated Log FIle Analysis Framework (ALFAF) prototype.

Finally, Section 3.4 consolidates the findings of the research to present the way forward for this research project in terms of the design and development of the ALFAF and its subcomponents, the Data Miner and Inference Engine, for performing automated Log File Analysis to aid system debugging.

3.1 Problem Context

This section presents the context for the problem being investigated in this research project. It introduces the concept of complex systems and the associated operational challenges associated with these systems as well as the challenges associated with conventional methods to debugging system failure. The MeerKAT Radio Telescope's Correlator Beamformer is also introduced as the case study system for this research project and is analysed from the perspective of the problem context.

3.1.1 Complex Systems

In Systems Engineering, a system is defined as an arrangement of interacting components that results in behaviours or meaning that the individual components do not [9]. The field of engineering typically results in the design and realisation of systems to solve problems.

One of the defining attributes of a system is that the output of the system is not equal to the sum of the outputs of the components making up the system. It is instead the interaction between the components of a system that yield the output desired from the system. These interactions give rise to unique properties or outputs that are critical to the system's main function or functions.

The term component, as used in the definition of a system above, applies to all the hardware, software, people, facilities, policies and documents that are required to produce system-level results [8]. That is, everything that is required to ensure the system functions as designed and intended, and that it produces the expected outputs and results, and that ultimately, it solves the problem it was designed to. From a strictly engineering design perspective, the components making up an engineering system are typically of the hardware and software variants, along with interfaces - either physical, logical or functional - that allow various system components, and subsystems, to interact with one another. In this dissertation the term *component* is used in this sense to refer to either the hardware, software or interface components of a system.

As the scope and complexity of the problems engineering has been tasked to solve increases, so too does the size and complexity of the systems designed to solve these problems. As these systems grow in size, the number of interacting components increases and the system is then known as a complex system [10]. The components making up this complex system may also be systems, i.e. subsystems, and these in turn may also be complex. Complex systems enable the realisation of engineering solutions able to solve increasingly more challenging problems more effectively and efficiently. Today, complex systems, and the practice of Systems Engineering that facilitates the realisation of such systems, are used by engineers across disciplines to develop innovative solutions to problems [8][9]. In this dissertation, the terms system and complex system will be used interchangeably to refer to an arrangement of components designed and employed by engineering practices to solve problems.

Engineering systems are often not only complex in their design, but also in their operation, maintainability and even their failure [10][11][12].

Complex systems, which often comprise both hardware and software, may be required to operate continuously, on-demand or when triggered by a certain event. In some cases, an end-user directly uses the system and in other cases, the system provides some implicit functionality that supports another service e.g. data centres for High Performance Computing or Multimedia Streaming. The operation of complex systems can often seem simplified from the perspective of the user, however, further investigation into the system and its interacting components can often reveal multi-level complexities. Furthermore, in addition to the stringent operating conditions, such systems may be exposed to harsh operating environments either physically, for example systems that are required to operate in extreme temperatures, or operationally in the case of systems that are required to operate for extended continuous periods of time [13]. In some cases, systems are also operated remotely which introduces another layer of complexity to the system. It is the responsibility of the engineers designing and building the system to ensure that it is capable of delivering the expected performance whilst operating in and under the specified set of conditions.

Given that these kinds of systems are employed to solve engineering problems, they are often expected to provide a certain degree of functionality over a specified period of time. This period is commonly referred to as the *operating lifetime* of the system. Thereafter, the system either reaches its *end-of-life* and is decommissioned, or it enters its *extended-life* period during which special maintenance procedures are invoked to extract further functionality from the system [10]. The inherent complexity of systems alludes to the notion that these systems are prone to failure [12][13].

When system failure occurs, the effects of the failure depend on the nature of the system and its environment, and may in some instances be fatal or catastrophic. In other instances, system failure may result in a complete loss of operational capability or it may result in the system transitioning to a degraded performance state in which it only offers limited functionality [9]

Most failures that occur during the operational lifetime of a system result in a loss of system operational capacity and the system enters a state known as downtime [10]. Again, depending

on the nature of the system, the severity of the impact of this downtime may differ, although unplanned downtime is generally regarded as undesirable. Downtime can, for example in the case of a manufacturing plant, result in limited production capability which carries with it a financial implication, or it may result in lost opportunity should the system not be capable of performing its intended function. When such failures do occur, the focus shifts to minimizing downtime by identifying and rectifying failures promptly. This task typically falls to engineers, operators and maintenance personnel and they often find themselves working under pressure to locate the failure and the cause of downtime, and to then restore the system to its expected operational capability.

Given that these complex engineering systems are built from a myriad of components, each with various failure modes and only a finite expected failure-free lifetime, coupled with the harsh operating environments and conditions these systems may be subjected to, it is understood that failures within such systems, although greatly undesired, are inevitable. Furthermore, when considering the nature of complex systems in general, it becomes apparent that there are many possible points of failure [12]. Although good engineering practice and intelligent system design methods can produce a robust system that is less prone to failure, not all risk associated with system failure can be mitigated entirely. Minimising system downtime then, when failures do occur, becomes the the focus when aiming to ensure that the system provides full operational capability during its operational lifetime. Achieving this requires effective and extensive system debugging in the event of such system failure events.

3.1.2 Complex System Failure & Debugging

Any system, whether simple or complex, is prone to failure [9]. Failure, in the context of engineering systems, is defined as an event or occurrence that halts or impairs system functionality and/or behaviour [1]. As previously discussed, consequences of system failure may vary depending on the system.

When a complex system fails, identifying the origin of the failure can be a challenging and time consuming task [14]. This is a direct implication of the nature of systems as they are made up of many interacting components. The larger and more complicated the system, the more individual components there are that are prone to failure. The process of debugging a failure then, in the case of complex systems, can sometimes lead to extended periods of downtime and may require additional efforts from engineers and/or operators.

Complex system failure is most analogous to failures that occur within large software systems [12]. Large software systems often comprise many interdependent and related libraries and modules that each contain a plethora of methods, classes, objects and variables. When developing large software systems, debugging uncommon failures often requires investigating all methods, classes and libraries that the particular suspicious section of code in question depends on. This process may often involve tracing a failure across and up through multiple function calls and execution paths, and investigating the code base that makes up other components of the larger software system.

Given that a complex system consists of components and smaller systems, that may in turn consist of smaller sub-systems that are further made up of assemblies of components, a similar debugging approach, as employed for the debugging of software systems is required as the failure may need to be investigated and traced across a number of components. Additionally, root cause analysis for failures caused by a component at a very low level in the system hierarchy may be even more challenging and time consuming to debug, as the failure is often flagged generically at a much higher level of the system.

In complex systems, it is typical for many of the subsystems making up the system to be dependent on one another. More specifically, they are dependent on the inputs and/or outputs to and from other subsystems and the interfaces between subsystems [8]. In this case, if one system fails, it will cause another system to also fail, and the error that gets propagated to the user level may not be an accurate representation of the actual, underlying cause of the failure. This scenario can often result in also having to trace a failure back and forth across multiple subsystems, and interfaces, in an effort to locate the root cause of the failure. Once the root cause of the failure is located, depending on the nature and complexity of the failure, additional analysis may be required to fully understand the failure.

From this, it can be seen that the process of debugging complex systems incurs additional challenges in performing fault localization and root cause analysis [12][15].

Debugging

Debugging is generally described as a task that combines the processes of both testing and correcting written code [16]. Typically the term is used to refer to software systems, but it is equally applicable to hardware, processes and system behaviours.

During the design of a new system, engineers and developers typically test various aspects of the designed system to ensure a smoother experience for the user and to ensure that the system delivers the intended functionality and performance under the specified operating conditions. A good developer or engineer is capable of identifying and testing the most common failure modes as well as a few obscure and uncommon ones. However, even system experts are seldom able to identify every single possible failure mode, and the nature of some systems is such that some failures only ever occur when the system is deployed and through user interaction with the system [15] [17] i.e. during system runtime. The occurrence of such failures often requires further debugging outside of the development phase, once the system is deployed, which, depending on the system and its application, may or may not always be possible.

The concept of debugging has evolved over the years to be more applicable to the ever-growing size and complexity of the systems being developed today [14]. Two types of debugging are considered, namely classic debugging and anomaly detection [17].

Classic debugging typically refers to software systems and involves the use of methods and techniques that determine which blocks of code, in a larger software program, are likely to contain errors and faults. This type of debugging is done by software engineers and developers. Classic debugging then, may be considered to be debugging that occurs during the design and

development phase of a system component. This applies equally to other system components including hardware and interfaces.

Anomaly detection describes the process of identifying unexpected events that occur during runtime, and/or while the user is interacting with the system [17]. Anomaly detection is not used during the design and development phase of components, but is instead performed while the system is in operation and this allows the process to identify system failures that typically only occur in the production or deployment environment. As a result, anomaly detection more accurately reflects the type of debugging that is required to identify failures in complex systems.

When systems fail, the task of debugging falls either to the engineers and developers who built the system or to the system operators. In a study by Xu and Rajlich [16], it is put forward that the process of debugging is fairly cognitive in nature, and that it holds similarities to the process a medical doctor uses to perform a diagnosis in that developers are presented with the symptoms and have to use their knowledge on the subject matter to identify the cause. The researchers go on to suggest that the process of debugging employs all six levels of Bloom's Taxonomy of cognitive learning. These levels, in order, are *knowledge*, *comprehension*, *application*, *analysis*, *synthesis*, and finally, *evaluation*. This suggests that the process of debugging is a non-trivial task and effective debugging requires a good foundation in domain specific knowledge and insight to the system in question. It is also suggested by Xu and Rajlich [16] that the level of cognitive ability required for this process is predominantly what distinguishes novice debuggers from their expert counterparts.

When it comes to performing debugging, there is no general solution or methodology that may be applied. Debugging practices are typically tailored to a particular problem and system, and are either enabled by the availability of information about the failure or are impaired by the lack thereof. Typically though, the process of debugging, and specifically fault localisation and detection i.e. anomaly detection, fit the framework described below as adapted from [16]:

1. Create a hypothesis as to the cause of the error/failure
2. Test the hypothesis and try to replicate the error/failure
3. Based on findings, modify the hypothesis or create a new one
4. Repeat until the cause of the error/failure is determined

From the presented framework, it is fairly obvious that the effectiveness of the debugging approach depends largely on the debugger's domain knowledge and experience with the system. Knowing how a system works, and how it is expected to work, as well as understanding the intricacies behind its functioning, can greatly improve the effectiveness of the process of debugging. This idea is support by [16] in which it is suggested that in order to debug effectively, a debugger must:

1. Understand what the symptoms of the problem mean,
2. Be able to obtain more information about the problem through further tests,
3. Identify which component(s) of the system are likely to be responsible for the failure,

4. Identify which processes or functions, of the suspicious component, are causing the failure,
5. Be able to identify the the root cause of the error, and,
6. Evaluate the obtained evidence to increase confidence in the identified error and to effectively determine how to rectify it.

It is often found that novice debuggers spend more time understanding the symptoms of failure before moving on to running testing experiments. System experts, however, with their better understanding of the system, as well as experience in the process of debugging, are typically able to fast track the process and begin with isolating the cause of failure much sooner. This further supports the statement that domain knowledge, in addition to the ability to effectively debug, is critical to debugging failures that occur within systems.

With sufficient domain knowledge, debuggers have various options and tools for debugging available at their disposal. In the case of building software systems, developers and debuggers typically employ Integrated Development Environments (IDE) during development. These IDEs not only serve as an effective environment for software development by including useful features such as syntax highlighting, but they also make numerous options for testing and debugging code available. The most commonly used tools during software debugging are console outputs, break points and unit testing.

On the the other hand, when building hardware systems, engineers typically find themselves spoilt for choice when deciding which tools to employ to debug and test a particular hardware design. The tools employed range from basic multi-metres and vector network analysers, to complex simulation software packages that enable the simulation of various hardware designs. These tools provide the engineer with the freedom and flexibility to conduct isolated experiments in an effort to identify the root cause of the problem or failure.

Typically however, the debugging framework described previously, and the tools used by developers and engineers for the purposes of debugging, is only available and employed during the design and development phases of a system. Once deployed and operational, engineers and developers seldom have the opportunity to run tests and experiments to determine the cause of failure in large complex systems as minimising system downtime is often a priority. Without being able to apply the debugging framework, where applicable, engineers and developers turn toward the log files generated by the system to perform log file analysis [18][12][14][1].

3.1.3 Log File Analysis

As mentioned, for some systems after deployment, the opportunity to debug failures through experimentation and replicating failures and using IDEs or other common debugging tools rarely occurs. For these systems, engineers and operators are still expected to perform debugging and the necessary rectification steps when system failures occur, and they're expected to do so quickly and efficiently to ensure minimal system downtime. Without traditional debugging tools available, system debuggers rely on system generated log files to perform log file analysis in attempt to identify the root cause of the system failure [15][19].

Log file analysis is a form of debugging that focuses on error or failure identification and localisation and involves analysing log files that are produced by the system in question during operation [14][18]. Many systems, of both the software and hardware variety, are often designed and configured to generate log files during runtime. In the case of software systems, program flow and execution is recorded by writing, or printing, a text string that provides information about a certain event. This may be the result of a method, a change in program flow or state or an event triggered by some external action e.g. a user connecting to the system. These events are logged, that is printed to some system console providing information about the software's operation, using *print*, *printf* and other similar functions in programming languages, and are typically also written to a file known as the *log file*.

In hardware systems, the manner in which log file generation is handled depends on the nature of the system. Some hardware systems that have microcontrollers may log system states and events, and store this information in non-volatile flash memory. This log may then be retrieved by reading the contents of this flash memory. Other hardware systems generate log files through software applications and processes that control or monitor the hardware. For example, a custom processing board may be polled, through controlling software, for information about its operating temperature or system state. This controlling software application then uses the received information to generate a log file for the hardware component.

Log files are generated while the system is in operation and, in most cases, engineers, operators, debuggers and sometimes even end-users, can access these log files at any time after they have been generated. The content and format of these log files vary and can differ substantially between systems. Most commonly, log files contain information pertaining to the various events that occur within the system, or they contain information about the status of the system in terms of its health and performance. This information may include, but is not limited to, timestamps of when the event occurred, the inputs and outputs of the event, objects that triggered or are impacted by the event, health and status information, performance related metrics and information about the user who initiated the event [18][19][15]. System designers have control, and subsequently the responsibility, during development over which system information is generated and stored in these log files.

A snippet of an example log file is shown in Figure 3.1 below. This log file was generated by the Correlator Beamformer, a subsystem of the MeerKAT Radio Telescope detailed further in Section 3.1.4.

```
2020-04-22 16:14:38,879 - INFO - array_1.wide.bc8n856m4k instrument.py:65 - FxCorrelator array_1.wide.bc8n856m4k created.
2020-04-22 16:14:38,890 - INFO - array_1.wide.bc8n856m4k fxcorrelator.py:120 -
========================================
Successfully created Instrument: array_1.wide.bc8n856m4k
========================================

2020-04-22 16:14:38,895 - INFO - array_1.wide.bc8n856m4k.katcp.server server.py:567 - Reading loop for client 127.0.0.1:40577 completed
2020-04-22 16:14:38,898 - INFO - array_1.wide.bc8n856m4k instrument.py:283 - Set synch epoch to 1587450614.00000.
2020-04-22 16:14:38,898 - INFO - array_1.wide.bc8n856m4k.katcp.server server.py:567 - Reading loop for client 127.0.0.1:40578 completed
2020-04-22 16:14:38,900 - INFO - array_1.wide.bc8n856m4k fxcorrelator.py:145 - Sync epoch pre-set to 1587450614.0.
2020-04-22 16:15:39,254 - INFO - array_1.wide.bc8n856m4k.skarab02042e-01 transport_skarab.py:2753 - Skarab is rebooting from SDRAM.
2020-04-22 16:15:39,254 - INFO - array_1.wide.bc8n856m4k.skarab02060a-01 transport_skarab.py:2753 - Skarab is rebooting from SDRAM.
```

Figure 3.1: Example log file from the MeerKAT Correlator Beamformer

From the log file in Figure 3.1, the various events that occur during the initialisation process of the Correlator Beamformer can be seen.

With sufficient content, log files are able to describe and illustrate the system behaviour during runtime. A system expert is then able look at the log files generated under normal operation and understand what the system is doing, and why the system is exhibiting a particular behaviour. Similarly, an expert is also able to analyse log files generated during failure cases and begin deducing where the problem originated.

Using log file analysis for debugging system failures makes the following assumptions (partly adapted from and supported by [20]):

1. The system being tested or monitored generates log files that includes information applicable to the debugging of the system (execution times, state changes, variable changes, use requests etc.)
2. System designers can define a logical and agreed upon logging policy that specifies the information contained within and the format of the log files
3. Engineers and operators with sufficient domain knowledge can analyse the generated log files to detect, identify and localize errors and failures that occur within the system

If these assumptions hold true, log files become an indispensable resource to operators and engineers tasked with debugging systems that experience failures during runtime.

Although log files contain a substantial amount of information, much of which is useful in determining the cause of failures within systems, there are challenges associated with log file analysis.

Because log files are generated during system runtime, and because the systems are typically run for extended periods of time, the amount of information contained within log files can quickly become overwhelming. Furthermore, the increasing complexity in the systems being designed and deployed today also results in more verbose log files as there are now more components within the system that have messages to log [14]. The size of the log files becomes a problem when tasking an operator or engineer to manually analyse them to detect faults. This is partly alleviated by the use of text search tools, such as *grep* [21], that enable debuggers to search for common expected log messages that may be associated with a system failure. To further motivate the use of search tools, it is common for log messages to be associated with a log level indicating the severity or importance of each log message. Examples log levels from the Python logging framework include ERROR, WARNING and CRITICAL [22]. Debuggers can use search tools to search for log messages of a particular level to reduce the volume of logs that need to be analysed. However, filtering log messages based on log level may hide useful system information that could potentially provide insights into the lead up to the failure.

Given the often unstructured and random format of log files [15][18], they are not easily readable by humans. The content of these log files is also predominantly text-based data, and it is well known that humans typically experience difficulty in analysing and finding patterns within

textual data [23]. Also, given the large amount of data to be analysed, it is readily assumed and accepted that human-error will be a factor.

Log files are also commonly extremely verbose and noisy [12]. Log files log the system operation during runtime, and not only during failure, although some more complex logging schemes do employ log levels and intricate logging schemes to determine which events ultimately get logged. As a result, the information that is important to debugging an error or failure is often hidden away among unrelated log messages. This further increases the difficulty, even for experts, in locating the root causes of failure from these log files.

Finally, log files, by design, log all kinds of failure. A study done by Mirgorodskiy et al. [1], makes a distinction between *fail-stop* and *non-fail-stop* errors and failures. *Fail-stop errors* refer to errors that cause an event or process to cease prematurely, whereas *non-fail-stop errors* refer to errors that do not result in a break in execution, but instead result in anomalous behaviour within the process or execution. Depending on the system in question, different types of errors and failures may be of varying levels of importance and since log files record all types, it further complicates the process of analysis as the errors of concern are interleaved with failures and errors of less significance.

One of the most concerning aspects that limits the usefulness of log file analysis as an effective debugging tool is that, in most cases, only an engineer or operator with sufficient domain knowledge of the system in question is able to extract useful information from these log files [13]. Given the nature of what is being logged i.e. execution flows, state changes, variable changes, request traces etc., it becomes obvious that an adept understanding of the system and its innermost workings and behaviours is required in order to deduce the root causes of failure from log files. To the untrained, log files appear to be nothing more than endless lines of random text.

Log file analysis then, while potentially useful, is not without its shortcomings. However, in the presence of sufficient domain knowledge, it is especially useful for determining the root cause of an error or failure within a large, deployed system. With sufficient domain knowledge, an engineer can analyse log files and pinpoint exactly where the error occurred, what triggered it, and even how it affected other components of the system. For this reason, log file analysis is deemed a viable solution when considering the debugging of large and complex systems.

Debugging Complex System Failure

As alluded to in Section 3.1.2, the complexity found within complex systems extends into the failure and debugging of these systems. In the previous subsection, it was discussed how debugging is typically employed to find the root cause of a failure or error that occurs within a system, and how this is a non-trivial task. With complex systems, however, the cost and complexity involved with identifying root causes of failure is exacerbated as the the size of the system increases [14][13].

The reason for this is due to the very nature of complex systems as previously defined: they are a collection of interacting components and systems. This essentially results in many more possible

points of failure and a larger area of focus when performing debugging as the interdependency between systems makes root cause failure analysis even more challenging. Additionally, complex systems are typically systems that are deployed for use for extended periods of time or in critical applications [10], and as such, conventional debugging tools and methods are usually not applicable as minimising system downtime is a major priority. Debugging deployed complex systems then typically relies on log file analysis.

Log file analysis is appropriate for debugging complex systems because the logs generated, when logging is implemented correctly, encompass the entire system and all its interacting components. As a result, the root cause of failure can be identified albeit with significant manual effort.

The shortcomings of log file analysis are however exacerbated in the context of large, complex systems:

- due to the increase in size and complexity of the system, the volume of data generated for log files increases exponentially and this makes manual review impractical and time consuming,
- given that a complex system consists of many systems, it may require multiple system experts, each with a specific domain knowledge, to collaboratively debug together to identify the root cause of failure,
- it might prove challenging to enforce a standard for log files that is system-wide throughout all components of the complex system.

As previously mentioned, downtime is incurred when a complex system fails. Reducing downtime involves both identifying the error and rectifying the error, and in some instances applying fixes to reduce the likelihood of the error occurring again. It has been demonstrated in practice that the time to recover a system from failure is dominated by the time taken to identify and localise the error after it occurs [23]. As a result, it is not uncommon for system failures to be resolved by simply restarting the system to restore operation in attempts to minimise downtime. The process of debugging the failure is then handled as a separate activity that does not impact or inhibit continued system operation.

While log file analysis can assist in identifying the root causes of failure, the shortcomings associated with this approach make it impractical in the context of complex systems. A more effective, and preferably automated, approach to debugging failures using log file analysis within complex systems is therefore desired.

3.1.4 Case Study System: MeerKAT Radio Telescope's Correlator Beamformer

This subsection introduces the MeerKAT Radio Telescope's Correlator Beamformer as the case study system for this research project.

Radio telescopes are instruments used in the field of radio astronomy to observe and study celestial objects. These instruments allow scientists and astronomers to observe the radio spectrum.

Today, most radio telescopes are designed as arrays made up of a number of smaller antennas as opposed to a single large antenna. Using interferometry, a technique in which electromagnetic waves are superimposed to cause interference, these smaller antennas are used to observe the radio spectrum as a single, larger antenna [24]. Designing and building radio telescope arrays, or interferometers, in this manner has benefits over building larger and more capable radio telescope instruments comprising a single antenna. Firstly, from a practicality and cost perspective, it is easier and more cost effective to construct multiple smaller antennas as opposed to a single large one. From a performance point of view, a greater telescope resolution can be achieved when using an interferometer. For a single antenna radio telescope, the resolution is dependent on the antenna's diameter while for an interferometer, the resolution is instead dependent on the longest distance between antennas in the array [25].

The Square Kilometre Array, commonly referred to as SKA, is a large array radio telescope that is to be built in South Africa and Australia. Upon completion, the array is envisioned to have a total effective collecting area of one square kilometre[5]. This radio telescope is expected to be 50 times more sensitive and 10 000 times faster, in terms of survey speed, than the best radio telescopes in existence today. This telescope is being built to allow scientists to address many unanswered questions regarding the Universe.

South Africa, being one of the host countries for the SKA Radio Telescope, has taken it upon itself to construct the KAT-7[26] and MeerKAT Radio Telescopes [27]. Both of these telescopes are being built in the Karoo, in the Northern Cape, 90 kilometres outside of Carnarvon. KAT-7, or Karoo Array Telescope - 7 Antennas, depicted in Figure 3.2 below, is an array radio telescope consisting of seven antennas each with a diameter of 12 m and a maximum baseline of 185 m. KAT-7 was primarily built as an engineering prototype for the larger MeerKAT Radio Telescope, but upon its completion in 2010, scientists have motivated to make use of the instrument for scientific observations.

Figure 3.2: Figure showing the KAT-7 Radio Telescope on site in the Karoo, Image source:[26]

The MeerKAT, meaning 'More' Karoo Array Telescope, Radio Telescope, depicted in Figure 3.3 below, is currently being built as a technology demonstrator and will serve as a precursor to the mid-frequency component of SKA Phase 1, with it eventually being integrated into SKA

Phase 1. MeerKAT consists of 64 antennas with a diameter of 13.5 m and a maximum baseline of 8 km[27].

Figure 3.3: Figure showing a subset of the MeerKAT Radio Telescope on site in the Karoo, Image source:[27]

The SKA, KAT-7 and MeerKAT Radio Telescopes are examples of complex engineering systems. These telescope systems comprise many different subsystems that each perform a unique function integral to the performance and operation of the telescope as a whole. It is the interactions between the various subsystems that give rise to the functional performance of the telescope. Remove just a single subsystem, component or interface, and the functionality of the telescope is severely impaired or reduced. The diagram in Figure 3.4, taken from the MeerKAT Correlator Beamformer Requirement Specification Document [28] is a functional context diagram that illustrates the entire MeerKAT Telescope System. All of its subsystems, components and interfaces making up the MeerKAT Telescope are visible on the diagram.

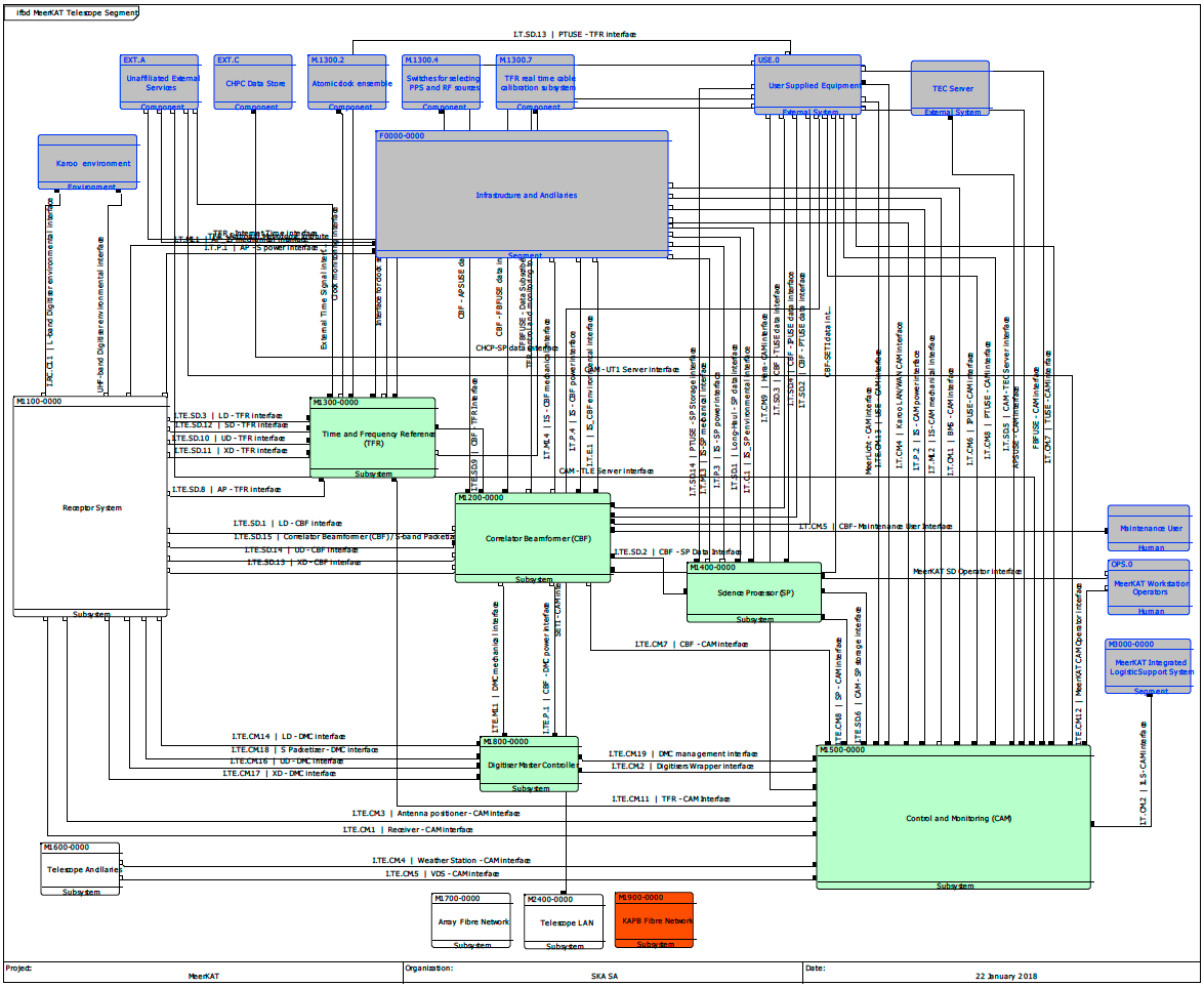


Figure 3.4: The MeerKAT Radio Telescope Functional Context Diagram. Image source: [28]

Each of the components or subsystems, depicted in Figure 3.4, may be considered a complex system on its own as each is made up of a number of interacting components, some of which also entire systems. The diagram in Figure 3.4 also serves to illustrate just how many various interacting components make up a complex system even at the top most levels of abstraction. In the case of the MeerKAT Radio Telescope, each component is itself a complex system designed to provide some specific non-trivial functionality to the larger telescope. One such component is the Correlator Beamformer which serves as the case study system for this research project.

The Correlator Beamformer, or CBF, forms part of the digital back-end of an array radio telescope. The Correlator Beamformer is responsible for performing the signal processing functions necessary to correlate the digitised data from all the antennas in the array to form a baseline data product that can be used to perform imaging. A simplified signal path for an array radio telescope is shown in Figure 3.5 below.

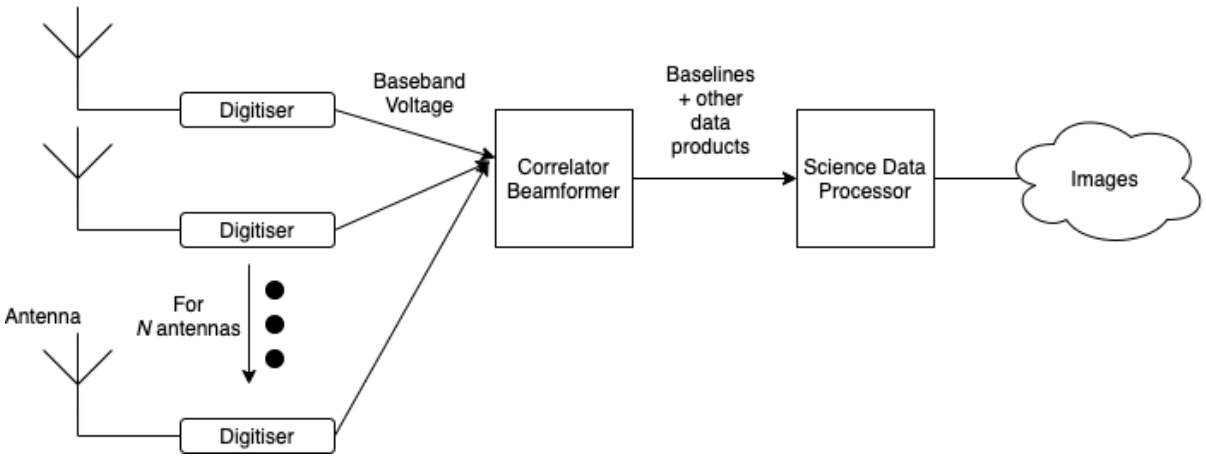


Figure 3.5: Figure showing a simplified signal path for an array radio telescope

In Figure 3.5 above, the digitisers are responsible for converting the analogue signal received by the antennas from space into digital format. For MeerKAT, each antenna has a digitiser situated on it. From the digitiser, all the digitised signals from each antenna, known as the baseband voltage data product in the MeerKAT system, then flow to the Correlator Beamformer via an Ethernet network interconnect. The CBF then performs digital signal processing to form a variety of data products that are then ingested by another subsystem, known as the Science Data Processor, which is responsible for performing the process of imaging to produce images of the observed sky.

The CBF performs an assortment of digital signal processing on the received signal to produce the required data products. The signal path through the Correlator Beamformer is shown in the Control and Monitoring Flow Diagram, taken from the MeerKAT Correlator Beamformer Control and Monitoring Interface Control Document [29], in Figure 3.6.

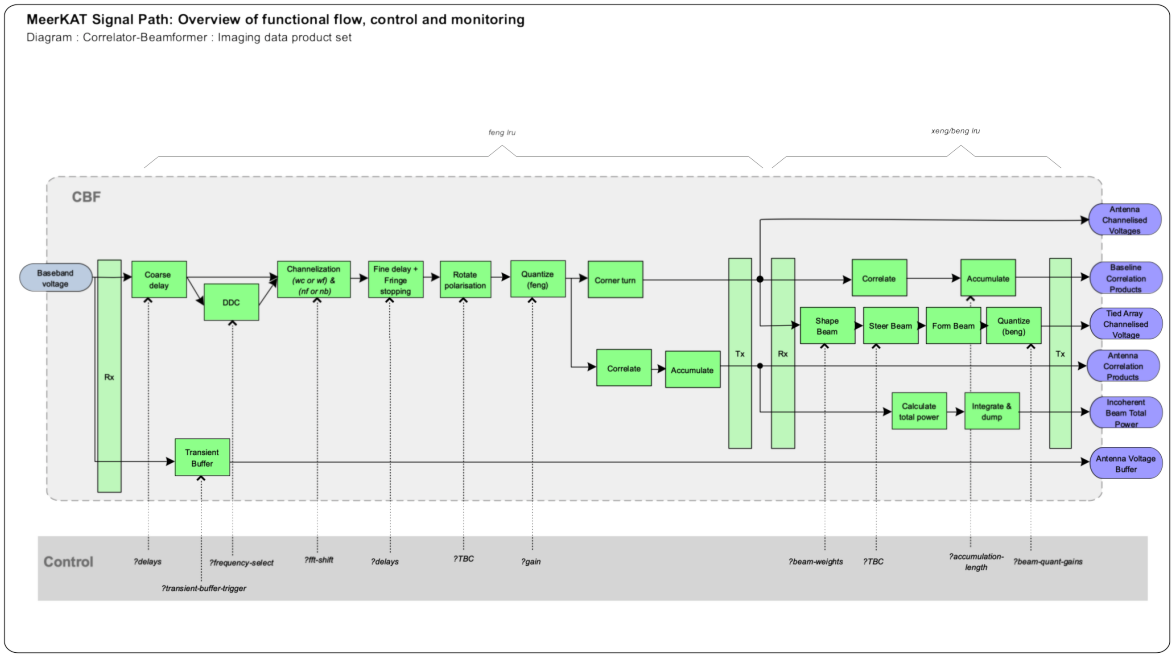


Figure 3.6: Diagram illustrating the functional flow of the MeerKAT Correlator Beamformer's signal path. Image source: [29]

The MeerKAT CBF is implemented as an FX Correlator. This implementation sees incoming antenna signals first being Fourier transformed, and then the correlation operation is simplified to a complex multiply in the frequency domain [30]. The digital signal processing performed by the CBF is split across two signal processing engines: The F-Engine and the X-Engine. The F-Engine, as indicated in the diagram above as *feng lru*, performs the Fourier transform and additional functions to compensate for delays, polarisation correction and data reordering. The output of the F-Engine is then sent to the X-Engines, shown as *xeng/beng lru* which then performs correlation and accumulation in the frequency domain. This forms the baseline correlations data product.

Also shown in the diagram, the X-Engine also performs additional functions such as *Shape Beam*, *Steer Beam*, *Form Beam*, *Calculate Total Power*, and *Integrate & Dump*, which are used to generate various beamformer data products and together make up the B-Engine. All these data products are ingested by other subsystems that use them to form images or for further processing. The interface across which the CBF transmits this data to the other systems is controlled by defining the packet formats of the data generated by the CBF.

For the MeerKAT CBF, the F-Engines and X-Engines are implemented in gateware on FPGA-based processing nodes. These processing nodes are described later in this section.

The product breakdown structure of the MeerKAT Correlator Beamformer consists of a number

of components ranging from physical hardware such as servers, switches, processing nodes, racks and cables, to software, including gatware and firmware.

Considering the provided overview of the physical makeup of the CBF, its functional architecture and how it fits in with the rest of the MeerKAT Telescope system, it can be deduced that the CBF itself is a complex system. The CBF implements the digital signal processing algorithms on FPGA-based hardware which involves the compilation of Hardware Description Language (HDL) code to implement these features. To govern everything, there is also a layer of control software and all the various hardware components of the CBF are interconnected by a Layer-3 40 Gigabit Ethernet network.

CBF Hardware

The physical hardware making up the CBF is shown in Figure 3.7 which illustrates the rack layout of part of the CBF System for the MeerKAT Radio Telescope.

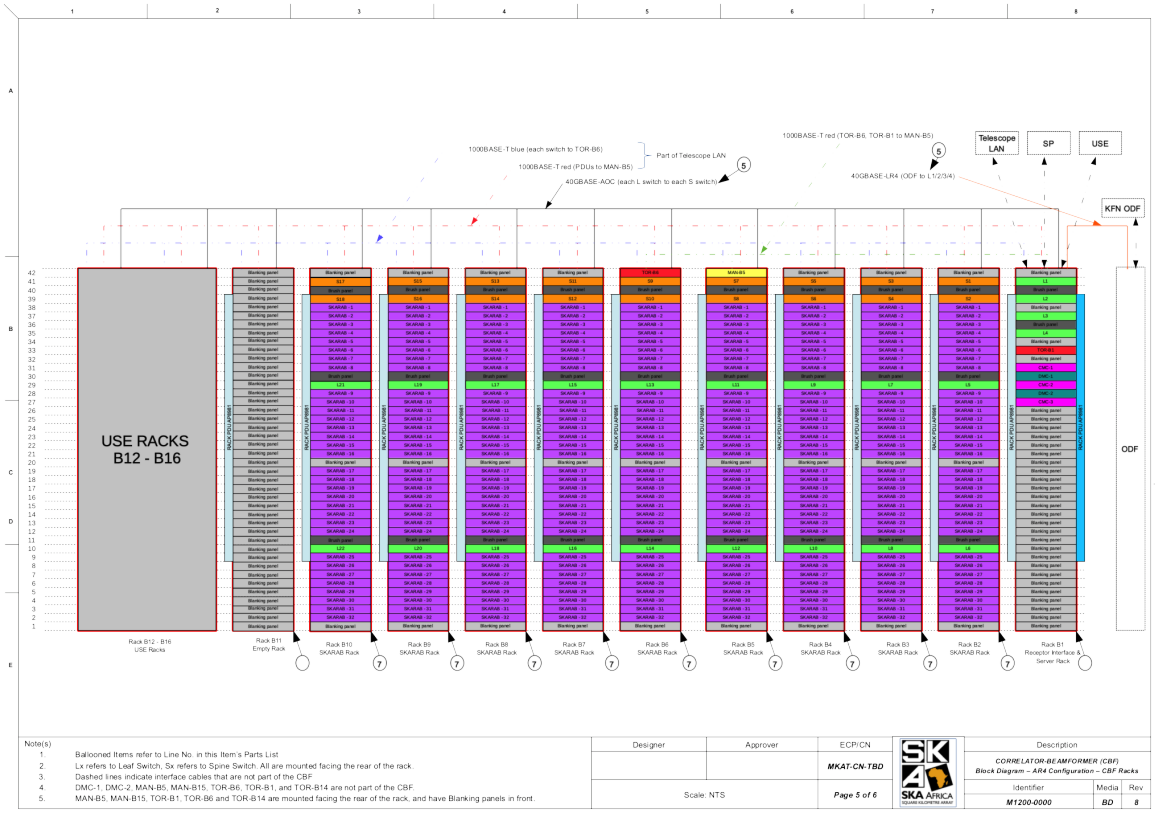


Figure 3.7: Diagram illustrating the rack layout of the Correlator Beamformer's hardware in the Karoo Array Processing Building. Taken from: [31]

As can be partly seen from the rack layout in Figure 3.7, the CBF consists of 288 SKARAB processing nodes, 54 switches (of which only 40 are shown in the rack layout and the remaining 14 are distributed across other rows in the data centre), three Correlator Master Controllers (CMC) servers, PDUs and power cables, and 42U Racks and necessary interconnect in the form of 1 GbE and 40 GbE network cables.

The processing nodes that implement the DSP are custom FPGA-based processing nodes known as SKARABs, or the Square Kilometre Array Reconfigurable Application Board, pictured in Figure 3.8. They are processing nodes developed by Peralex[32] according to the specifications put forth by the South African Radio Astronomy Observatory. These processing nodes are actively used by the CASPER community for the purposes of performing real-time digital signal processing, primarily for the purposes of radio astronomy[33]. These custom processing nodes comprise of Xilinx FPGAs, high-speed HMC memory, 40GbE network interconnect, and a soft-core microcontroller that runs a simple control server for controlling the individual nodes.

Figure 3.8: Figure showing the SKARAB processing node. Image source: [32]

The CMC servers are commercial-off-the-shelf servers running a version of Debian. Each CMC runs a collection of software processes that facilitate the control and monitoring of the CBF. Three CMCs are installed to provide redundancy in the case of failure. The PDUs and power cables provide redundant power to all the processing nodes, servers and network switches.

The interconnect between all the CBF processing nodes, CMCs as well as between the CBF and the other subsystems is provided by a 40 Gigabit Ethernet Network implemented with 54 switches that are arranged in a Multicast Clos Network Architecture, alongside a mixture of copper and optical 40GbE cables. The cables used depends on the distance from each node to an available switch. The CBF switches form what is known as the Telescope Data Switch that is responsible for moving all the signal data between all the subsystems. Another network, known as the Telescope LAN, that does not form part of the CBF system, is responsible for routing control and monitoring data between all the subsystems and the Control and Monitoring Subsystem as shown in Figure 3.4. This network is based on a 1 Gigabit Ethernet interconnect.

CBF Software

As mentioned in the previous section, the CMC servers run software processes that facilitate the control and monitoring of the CBF. The CBF system is not controlled directly by an individual user. Instead, another telescope subsystem, the Control and Monitoring (CAM) Subsystem, that interfaces directly with the user, interfaces with the CBF to issue controlling commands and to monitor the system health and status.

The interface between CBF and CAM is a network interface and the application layer is defined by list of commands that CAM can issue to the CBF as well as a list of sensors that CBF shall send to CAM [29]. These sensors report the health and status of the CBF system. Both in terms of the physical hardware and the functional DSP that is being implemented. These sensors are used to determine whether or not the CBF is functioning as expected and their values are populated by polling various hardware registers on the processing node or various state variables of the software processes running on the CMC.

The underlying protocol of the application layer of this interface is implemented using the Karoo Array Telescope Control Protocol (KATCP) [34]. It is a communication protocol developed by the South African Radio Astronomy Observatory (SARAO) for interfacing between various subsystems over a network. This protocol defines the syntax and formats of messages that are to be exchanged.

Control and monitoring of the CBF is facilitated through a number of software packages running on the CMC. These are illustrated in Figure 3.9 below.

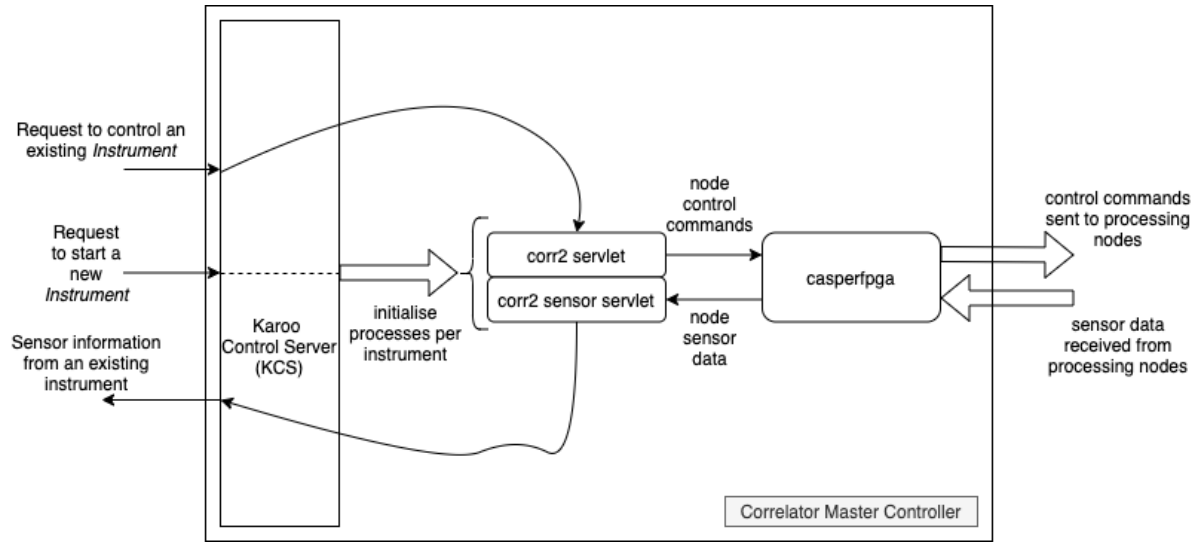


Figure 3.9: Figure showing the various software packages running on the Correlator Master Controller and the interactions between them

With reference to Figure 3.9, the way in which the CBF's software stack works is as follows: An instance of Karoo Control Server, KCS, runs on the Correlator Master Controller. Requests are received to start an instrument from the CAM subsystem. When a request to start an instrument is received, KCS assigns resources (processing nodes) for the instrument and starts two processes: *corr2 servlet* and *corr2 sensor servlet*. Any control commands received are sent to the *corr2 servlet* and sensors are sent from both *corr2 servlet* and *corr2 sensor servelet* to the CAM subsystem.

These processes are created by the the *corr2* Python package. This package provides an interface for controlling a group of processing nodes to perform DSP functions for a correlator. It relies

on the use of another Python package, *casperfpga*, which provides an interface for directly controlling and monitoring FPGA-based processing nodes based on pre-defined protocols. The *casperfpga* package supports many different underlying transport layers. The *SkarabTransport* transport layer interfaces with the soft-core microcontroller running on the SKARAB nodes.

After the resources have been assigned, the Correlator Beamformer is initialised with a specific configuration file depending on which instrument or mode is required. During initialisation, the SKARABs are programmed with the appropriate bitstream images and the *corr2 servlet* configures various parameters of the F-Engines and the X-Engines.

Once the Correlator Beamformer is initialised, the servlets remain running while the instrument is in use. Via the CAM subsystem, the telescope operator may issue commands to the CBF to tune or tweak certain parameter values (e.g. fft-shift, delay and gain values) as necessary. These commands are received by the *corr2 servlet* and this invokes *casperfpga* as necessary to configure the processing nodes. The CBF then also starts emitting sensors to the CAM subsystem. As soon as the CBF is initialised, it begins processing the incoming data and emitting this data to any subsystems that are subscribing to it.

Multiple instances of the Correlator Beamformer may be started and they may run simultaneously. Each instrument has a unique *corr2 servlet* and *corr2 sensor servlet*. These process keep running until the instrument is torn down.

CBF Logging

The various software packages running on the CMCs have been developed to implement logging and generate log files. The log messages that are generated are written to *stdout* and *stderr* during runtime and are also written to log files that are saved to disk.

The SKARAB processing nodes themselves keep log records about operating voltages, currents and fans speeds as well as unexpected shutdown events. These are stored in non-volatile memory on the SKARABs and can be retrieved remotely via the *casperfpga* utility. These log messages, however, are not periodically logged during operation nor are they written to the CBF system log files. These processing node specific logs are typically only consulted when a processing node is suspected of being faulty.

The log messages generated by the CBF contain information about instrument initialisation, system runtime operation, system health and processing node health and status. The log scheme has been implemented to facilitate the process of debugging when system failures do occur. An example log file generated by the CBF is shown in Figure 3.12 in Section 3.1.3.

These log files can become quite verbose and lengthy, and consulting them to identify an error or anomaly can be time consuming. Even system designers and experts find it challenging to find errors in the logs and progress is typically faster when the debugger knows what to look for and where to look for it. The CBF engineers rely on feedback from the telescope operators who report system errors to decide on a starting point when performing system debugging using the log files. Generally, debugging via log files involves the use of search tools, such as *grep*[21], to find the suspicious parts of the log file. Even with search tools, and with an idea of where

to look, this process can be time consuming and may not reveal the root cause of the failure. Additionally, the reliance on search tools and prior knowledge results in greater difficulty when tasked with finding new errors.

CBF Operation, Failure and Debugging

The Correlator Beamformer is regarded as one of the most critical components of the MeerKAT Radio Telescope. In the event of CBF failure, the functionality of the Telescope is severely impaired. Critical failure within the CBF system can result in an unusable telescope which causes a loss of valuable observation time. Furthermore, should the CBF fail during an observation, the data obtained during the observation may become lost or unusable to the rest of the signal processing pipeline.

Minimising downtime, then, of the CBF is a priority to ensure the continued operation of the MeerKAT Radio Telescope. As previously discussed, when a complex system fails, one of the most challenging activities is locating the root cause of the failure.

As can be seen from the MeerKAT Context Diagram in Figure 3.4, many of the subsystems making up the telescope interface with one another. As a result, each system is typically dependent on the inputs and/or outputs to and from other subsystems. Because of this interdependency, a failing subsystem often causes another subsystem to also fail. This further complicates debugging as the error that is propagated back up to the user level may not be the root cause of the failure. This scenario can often result in having to trace a failure back and forth across multiple subsystems in an effort to find the root cause.

Identifying the root cause of failures is challenging in the context of the MeerKAT CBF. Since minimising downtime often takes priority, engineers typically resort to strategies to restore system functionality as opposed to identifying and locating the root cause of failures. Log files, while available are rarely analysed in great detail due to the challenges associated with log file analysis.

As a result of the challenges associated with performing root cause analysis on the MeerKAT CBF system, it is a prime candidate as a case study system in the research to design and implement automated log file analysis for system failure detection.

This research project will consider the MeerKAT Correlator Beamformer as the case study system for the design and implementation of an Automated Log File Analysis Framework for Performing Anomaly Detection.

3.1.5 Summary

This section introduced the concept of complex systems and the challenges, as well as implications, associated with debugging failures that occur within these complex systems. Typical debugging methodologies and frameworks were introduced and the argument for log file analysis as a debugging tool, specifically for large, complex systems, was presented. The major challenges that make log file analysis impractical on large systems were discussed, and this motivated toward the need for automated log file analysis methods and techniques. The MeerKAT Radio

Telescope's Correlator Beamformer was introduced as the case study system for this research project and the associated challenges in debugging this system were presented.

3.2 Automated Log File Analysis

As can be seen from Section 3.1, there is a strong motivation for a move toward automated methods to perform log file analysis in the context of debugging complex systems.

Research shows that the topic of automated log file analysis has received much attention in last few years. The problem has been explored and some progress has been made toward achieving this. Over time, the methods of implementing the automated analysis of log files has shifted from statistical and rule-based methods to machine learning and deep learning based methods. This is mainly due to the increasing volume of log files, the maturity of deep learning algorithms and advancements in computing capability that has made these methods more effective and efficient [35]. However, there are still some challenges associated with these methods. Machine and deep learning methods typically lack some degree of interpretability and debuggers and engineers may not know how the algorithms arrived at their decisions [4]. Furthermore, many machine and deep learning algorithms require labelled datasets for training [36], and given the size of log files and the amount of log messages, generating these labelled datasets may be expensive. In spite of these shortcomings, machine learning and deep learning algorithms have been shown to outperform other methods of log file analysis and research shows that these shortcomings are being addressed.

This section introduces the concept of automated log file analysis and surveys relevant research literature to understand how this may be achieved. As per the current trends in research, this project focuses on methods based on machine learning and deep learning algorithms, but where relevant, other methods are presented for completeness. The concept of machine learning log file analysis is discussed and various techniques for implementing this are evaluated.

3.2.1 Machine Learning Based Log File Analysis

Machine learning based log file analysis employs state-of-the-art machine learning and deep learning algorithms to automatically perform log file analysis with the objective of performing some system diagnostic task or debugging [37]. Machine learning and deep learning algorithms are models that are trained on a given dataset and are later used to make predictions and inferences from new data [38]. Machine learning models are not programmed, but are instead trained to modelling non-linear complex relationships among multivariate data. It should be noted that in this research project, the terms Automated Log File Analysis and Machine Learning Based Log File Analysis will be used interchangeably.

Generally, Machine Learning Based Log File Analysis consists of four separate processes as shown in Figure 3.10 below [4], [35].

As seen in the diagram above, these processes are *Log Collection*, *Log Parsing*, *Feature Engi-*

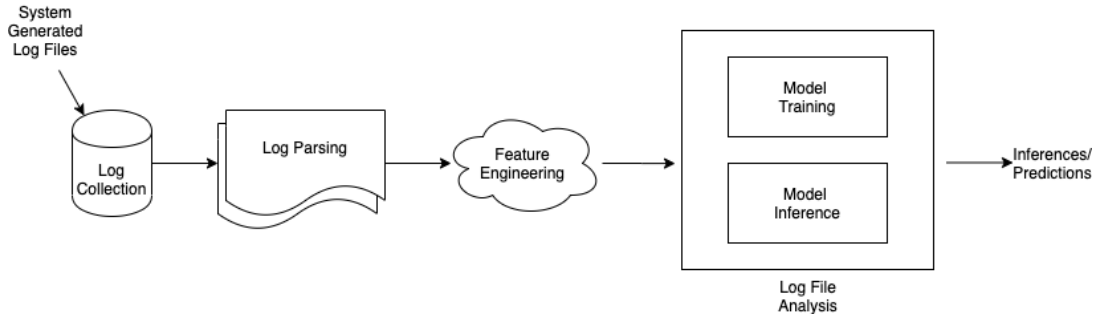


Figure 3.10: Figure detailing the four separate processes of Machine Learning Based Log File Analysis.

neering and finally, *Log Analysis*. In the context of machine learning, the first three processes are typically grouped together and referred to as the Data Preparation stage. The Log Analysis process further consists of a model training stage and a prediction/inference stage.

Log Collection is the process of acquiring the log files generated by the system under analysis. In some systems, log messages are written to a log file that is stored on disk. Other systems print log messages to a console output or stream log messages to another process. Some systems may even do both. The process of Log Collection is heavily dependent on the system under analysis and is therefore not considered further here. This process is detailed for the MeerKAT Radio Telescope's Correlator Beamformer in Chapter 8.

For the purposes of the remainder of this discussion, it is assumed that the system under analysis generates log files and that these can be acquired for further analysis. The remaining processes, Log Parsing, Feature Engineering and Log Analysis, are detailed further in the following sections.

3.2.2 Log Parsing

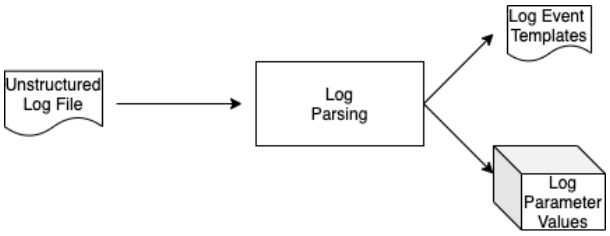


Figure 3.11: Figure illustrating the process of log parsing

Log parsing is an important pre-processing step in the effort to automate log file analysis. It is employed in various automated log file analysis methods and not just machine learning based methods which typically have data pre-processing as a prerequisite [39]. The process of log parsing, involves extracting information from log files and transforming them into a sequence of structured events [35].

Log parsing is required because the log messages contained in the log files generated by many systems are commonly unstructured as developers use free-form text to record log messages as it is more flexible [35]. This allows developers to log various events, debug information, warnings

and errors without having to conform to a strict, predefined structure. However, without any form of structure, automated analysis methods, and manual inspection methods, struggle to extract any meaningful information from the log files as there is seemingly no inherent pattern to the log messages. Log parsing addresses this by converting the logs into a data format with structure that is more efficient to work with and easier to extract patterns from. While the content of log messages varies greatly, they typically contain two major components: a static or constant event template, and the variable runtime parameters associated with each instance of an event. Consider the example log messages in Figure 3.12.

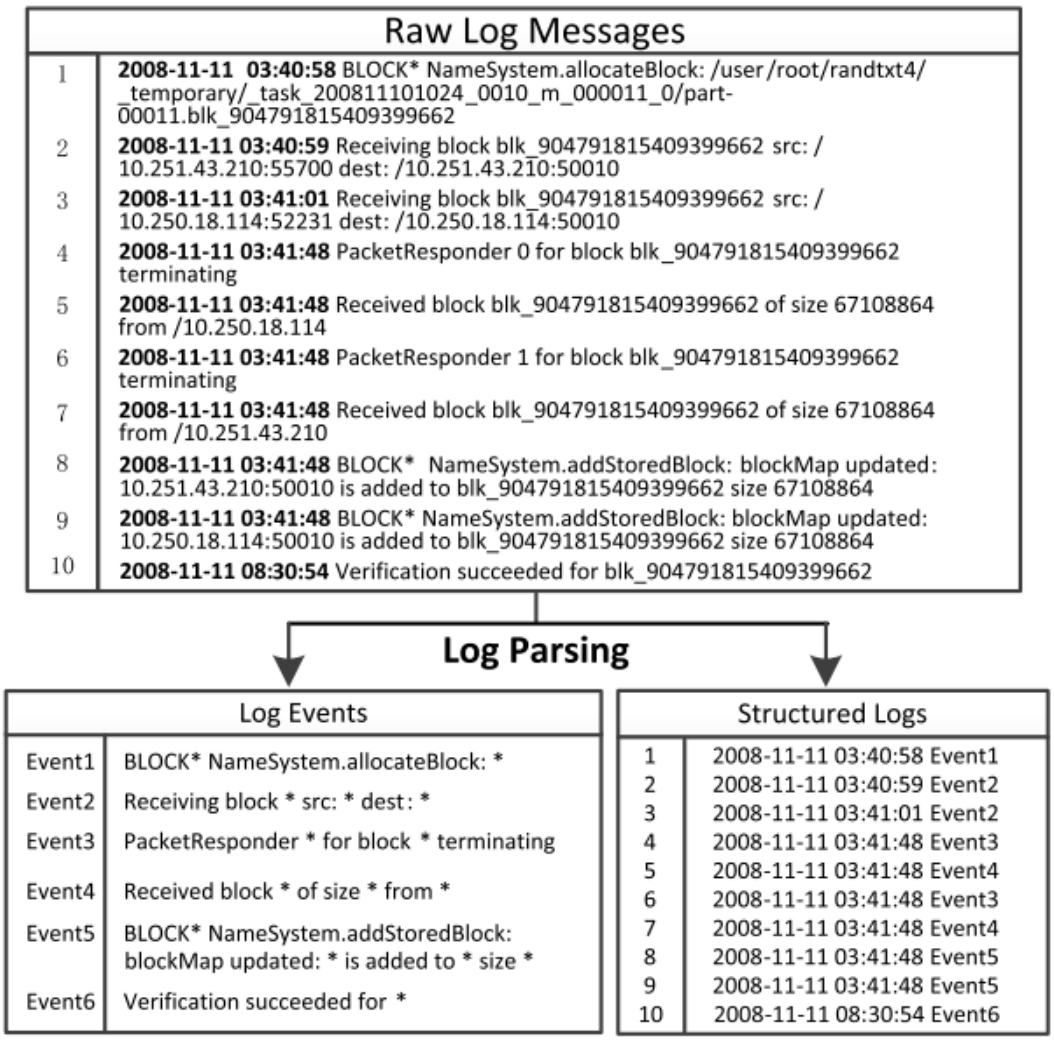


Figure 3.12: Figure showing raw logs being parsed into a event templates and arranges as a sequence of event templates. Image source: [35]

From the log message example in Figure 3.12, an example event template is *Receiving block * src: * dest: ** and the associated runtime parameters for the event are *blk_9047918154...*, */10.251.43.219:55700* and */10.251.43.210:50010*. Since events may occur frequently during program execution with different runtime parameters, log messages develop an unstructured nature. The goal of log parsing, as illustrated in Figure 3.11, is to extract the event template

and the associated runtime parameters for each log messages and to present the messages as a sequence of structured events. An example of this is shown in Figure 3.13.

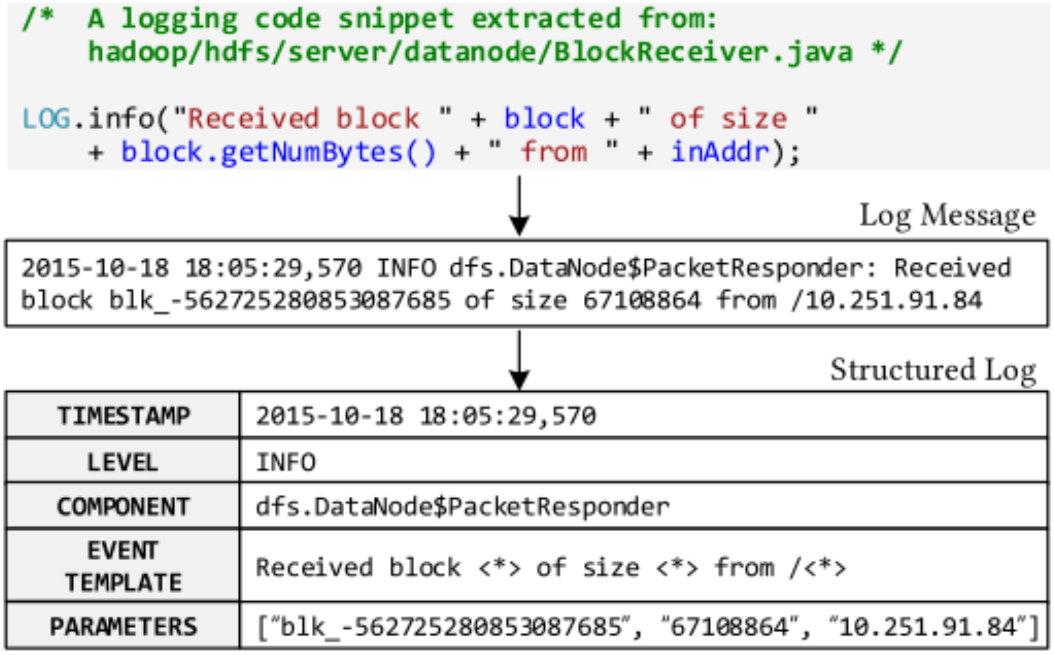


Figure 3.13: Figure showing log parsing. The source code generating the log message is shown, followed by the actual log message. Finally, the parsed log split into log event template and runtime parameters is shown. Image source: [40]

As can be seen from Figure 3.13 above, the log messages are all parsed to extract their event templates and runtime parameters. The log event templates are then arranged as a sequence of structured events as shown in Figure 3.12. Having the log files represented in this way makes patterns in the log file more readily detectable by data-driven analysis methods such as machine learning.

Early implementations of automated log parsing employed the use of regular expressions to define a search pattern to match to specific log events [41]. This yielded some success, but the the regular expressions had to be created manually for each event type expected in the logs. This proved to be quite challenging as the number of event templates for a given system increased. Additionally, if the system in question was ever upgraded or modified, and the event templates changed, the regular expression rules would need to be manually updated as well. Other methods considered extracting the event templates directly from the source code [4]. Most log messages are implemented using *printf*, *print* ,or other similar statements, to print a string consisting of the event template and variables corresponding to the runtime parameters associated with that event. Therefore, extracting these from the source code is plausible. However, source code is not always available and methods often need to be tailored to a specific programming language. As a result, these methods are neither scalable nor robust.

To fully automate the process of log parsing, research as moved toward data driven approaches that seek to automatically learn patterns from the data [40]. Such approaches commonly include Clustering [39], Frequent Pattern Mining [18] and Heuristic-based approaches [42] [43] [44].

Clustering

This approach refers to techniques that rely on clustering logs messages of similar patterns into groups, and then extracting log templates from the common components of all the log messages in each group or cluster. Research has resulted in a number of log parsing techniques based on clustering methods including LogMine[45], LogSig [46], and LKE[39]. LKE, uses a hierarchical clustering algorithm based on the weighted edit distance between pairs of log messages[39]. LogSig uses message signature based algorithm to group log messages into a pre-defined number of clusters [46]. LogMine also employs hierarchical clustering and clusters log messages into groups more efficiently by exploiting various optimisation techniques for clustering.

Frequent Pattern Mining

Frequent Pattern Mining techniques identify patterns that occur frequently in given datasets. The log message event templates are assumed to be patterns that occur frequently in log files. As a result, frequent pattern mining can extract these event templates from logs. Early examples of frequent pattern mining techniques can be seen in the SLCT [18] and in its extension, LogCluster [47].

Heuristic-based

Heuristic-based methods refers to a group a of methods that exploit the unique characteristics or features of log files [40]. One such method, AEL, compares the occurrences between constant parts of log messages, i.e. event templates, and variable parts, and then groups the messages into clusters based on this comparison [48][49]. Another method, IPLoM, uses iterative partitioning to group group messages by the length of the message and the position of various tokens occurring in the message [43][42]. Drain is a log parsing technique that uses a parsing tree to represent the log messages and extract common event templates [44].

Other methods

There are also other log parsing techniques that don't fall into the aforementioned categories. One such method is Spell which uses the longest common subsequence algorithm to parse log messages and extract event templates [50].

Logparser: A log parsing toolkit for benchmarking and evaluation

As can be seen, there are many log parsing techniques based on various methods. All these techniques share the common goal of extracting log message event templates such that the log files can be represented as a structured sequence of events. The authors of [40] and [37] have studied and evaluated various log parsing techniques that have been implemented in recent years. Their study has resulted in an open-source toolkit, Logparser, that implements and

packages many of the most prominent log parsing techniques in recent years into a single tool that may be used for evaluation and benchmarking [7]. This tool, alongside the associated research studies, will be used to initially identify and evaluate log parsing techniques that may be considered and implemented in this research project. This is detailed in Chapter 4.

3.2.3 Feature Engineering

In machine learning, feature engineering is the process of ensuring that the dataset that will be used to train the machine learning algorithm consists of enough relevant, useful and representative features [38]. Poor feature engineering can directly result in poor algorithm performance.

Once the log messages in log files are parsed into structured events, feature engineering needs to be applied to extract useful features from this new representation of the log files.

For example, in [35], each log message in the parsed log files is used to generate an event count vector that represents the occurrence of each event in the log file. Event count vectors are generated for various windows, or groupings, of sequential log messages, and the event count vectors are combined to form an event count matrix that is fed as a feature matrix to various machine learning algorithms.

Another study, [4], considers the ratio of each state occurring in the system (e.g. *Preparing*, *Aborted*, *Committed*, *Committing*) to form a state ratio vector and the occurrence of each event type to form a message count vector. These features are then passed to a Principle Component Analysis (PCA) model to detect anomalies.

The feature engineering required greatly depends on the machine learning algorithm to be used and the main objective of the model. As such, the process of feature engineering will be explored further as applicable when discussing the various methods of automated log analysis in the next section.

3.2.4 Log File Analysis

As discussed in Section 3.1.3, Log File Analysis is a debugging process that utilises the log files produced by systems during operation to identify and locate system failures and errors. However, these are not the only uses of Log File Analysis. Studies of Automated Log File Analysis have demonstrated the use of log file analysis for the purposes of performing usage analysis, anomaly detection, performance modelling, failure identification and diagnosis and even system modelling [40][37][51]. It can therefore be argued that Automated Log File Analysis has extracted more information from log files and has significantly increased the value and usefulness of these log files.

Automated Log File Analysis methods aim to address many of the shortcomings and challenges of conventional Log File Analysis such as enormous volumes of logs, the heterogeneous, unstructured nature of logs and the requirement of system knowledge to understand and make sense of the information contained in log files.

This section presents an overview of the various techniques that have been implemented in

recent years to achieve Automated Log File Analysis using machine learning and deep learning methods. As these methods are machine learning based, they often rely on data preprocessing steps, i.e. log parsing and feature extraction, as previously discussed. Additionally, these methods often include two stages of development and operation: a training stage during which a machine learning model is learned from the provided data, and a prediction stage during which the model is used to operate on new data and make predictions.

In this research project, the primary aim is to use Automated Log File Analysis to aid in debugging failures. This problem is modelled as an anomaly detection problem as anomalous log messages can be assumed to be indicative of a system failure. Anomaly detection, which is concerned with detecting abnormal system events and behaviours [40], in the context of Log File Analysis, is often used synonymously with failure identification and failure prediction. Therefore, only Automated Log File Analysis techniques that aim to perform anomaly detection, failure prediction or failure identification are considered.

Recurrent Neural Networks (RNN) & Long Short Term Memory (LSTM)

Recurrent Neural Networks (or RNNs) are a class of neural networks that excel at processing sequences of data [38]. They are most commonly used to build models for generating or predicting sequences of text for Natural Language Processing applications, but can be used to model any sequential datasets. An extension of RNN, namely Long Short-Term Memory (or LSTM), increases the memory capacity associated with the network and enables it to better model long range dependencies across the sequential dataset. Recent studies have made use of RNNs and LSTM to treat log messages as a sequential dataset and build models for anomaly detection and failure prediction.

Work by Du et al. [52] proposes an anomaly detection method called DeepLog that utilises RNNs and LSTM. DeepLog is presented as a general purpose anomaly detection deep learning model that is capable of detecting anomalies in an online manner. DeepLog also considers the generation of workflow models to assist with failure debugging and diagnosis. DeepLog uses two main approaches to detect anomalies. The first approach considers anomalies associated with the execution path and the second considers anomalies associated with system performance and parameter values. Both approaches make use of LSTM models. First, log messages are parsed into log events and log parameter vectors. This study makes use of the Spell log parsing technique [50] to parse logs into the required format. Once parsed, the logs are then used to train an LSTM network to model execution paths as a sequence model. A window of sequenced log events is considered as features, and the next event, after the window, is considered the output. In this way, an LSTM is trained to model system behaviour. When a new log message is received, the model can predict what the most likely log event should be, based on the prior log events in a window period, and decides if the received log message is anomalous or not. For system performance and parameter value anomaly detection, the log parameters are modelled as a time series and an LSTM is again trained to predict what the most likely values for the various parameters should be and flag anomalies if there are discrepancies. DeepLog only requires log files generated during normal system execution and claims to only require a

minimal dataset for training the models. Furthermore, only training the model using normal log events makes the model more robust in detecting unknown and new anomalous events. This method is also capable of handling concurrent tasks and task divergence as represented in log files. By modelling the log files as a sequence of log messages and building models around this, DeepLog circumvents the need for a labelled dataset to train a model. DeepLog also makes provision for user provided feedback to allow users to flag incorrectly detected anomalies and provide additional data for updating the model.

Meng et al. [53] also consider using LSTMs and propose various improvements over DeepLog via a new method called LogAnomlay. LogAnomaly differs from DeepLog mainly in the manner in which features are created. Similar to DeepLog, LogAnomaly considers two kinds of anomalies. The first is sequential anomalies that are similar to DeepLog's execution path anomalies. The second is quantitative anomaly detection that refers to the relative occurrence of certain events in a log file. The features for training the LSTM are generated using a variation of the Natural Language Processing preprocessing tool, word2vec, called template2vec. This creates embedding vectors for the log templates and is similar to techniques present in Natural Language Processing. To handle changes to the system, LogAnomaly assumes that new log messages introduced to the system, as a result of upgrades or updates, are largely just variations of existing logs. As such, new log messages are matched to existing log message templates based on similarity.

A study by Zhang et al. [54] proposed the use of RNNs and LSTM for the problem of anomaly detection using log files as traditional supervised learning methods cannot capture time dependencies between the features fed to these models. This study also proposed modelling the log messages as a time series dataset. Further research that uses RNNs is carried out by Rhode et al. [55]. In this study, RNNs are applied to the field of cyber security to detect malware. Special mention is made of the importance of hyper-parameter tuning of the proposed models. In [56], LSTMs are again considered for anomaly detection using log files. This study however considers the nature of real-world log data and discusses challenges around adapting models to evolving systems i.e. log files that change over time as the system changes. It also considers the challenges associated with noise that is introduced to the data and suggests that many log parsing techniques may introduce unwanted noise into the log files.

In [57], Qu et al. propose the use of an ensemble of deep learning models to perform anomaly detection using log files. First, a convolutional neural network is used to extract feature relationships in spatio-temporal dimensions. Then, an auto-encoder is used to learn robust representations from this feature space. Finally, an RNN is used to detect anomalies on new log data.

Work by Brown et al. [58] also considers using LSTMs for log file analysis but focuses on making the models more interpretable by humans. Machine Learning and Deep Learning models are often criticised as being *black boxes*, and its often challenging to gain insights into how the models arrived at particular predictions or inferences. This supports the argument that in the context of anomaly detection, some introspection into the process of anomaly detection can increase confidence in the model and further aid system debugging.

Principle Component Analysis (PCA)

Earlier work by Xu et al. [4] considers the use of Principle Component Analysis (PCA) and models log file analysis as an anomaly detection problem. First, log files are parsed to extract log events. Here the proposed log parsing method uses software source code for supplementary information. This reliance on source codes means that this parsing technique, while efficient, is not very robust as it assumes the use of certain constructs to write log messages and is tailored to a specific programming language. Once the log messages are parsed, two unique features are extracted from the structured sequence of logs. The first feature is termed the *state ratio vector* and summarises the aggregated system behaviour over a specific period of time by considering the ratio between the time the system spends in each possible state. The second feature, termed the *message count vector*, counts the number of times each message occurs and is used to detect issues pertaining to individual operations.

These feature matrices are then fed to a PCA model to detect anomalies. PCA is a statistical model that is often used as a dimensionality reduction method [38]. PCA works by projecting the dataset onto a new set of coordinates that represent components that capture the majority of the variance in the dataset. Using PCA for anomaly detection exploits the assumption that features created from log messages during normal operation are highly correlated and that anomalous events will manifest as outliers on the principle components. PCA showed promise in the research conducted by Xu et al. [4] and was further evaluated by He et al. [37] and a distributed implementation was investigated by Astekin et al. [59]. He et al. found that the accuracy of anomaly detection using PCA was sensitive to the data, but also noted that these models were simple to construct and tune. PCA is considered an unsupervised machine learning technique as it does not require a labelled training set to train a model. This is advantageous as a labelled set of log files may be expensive and challenging to obtain.

Clustering

Clustering refers to a collection of unsupervised machine learning techniques that aim to group various instances together into clusters [38]. Given that log files are generally unlabelled, i.e. the individual log messages are not labelled in terms of whether they are anomalous or not, it makes sense that many recent studies have considered clustering techniques for the problem of automating log file analysis.

BeeHive, an automated log file analysis technique presented by Yen et al. [60] uses a variation of the K-means clustering algorithm to group pre-processed log sequences into various groups. The presented method is specific to detecting anomalies in enterprise networks and, as such, the feature extraction process is tailored to this context. Log messages are first pre-processed with emphasis placed on certain key pieces of information in logs including IP address to Host Mappings and timestamps. The features that are created consider themes such as *destination*, *host*, *policy* and *traffic*, all of which are important considerations when considering the flow of data across a large enterprise network. With a feature set of 15 features, PCA is first applied to reduce the dimensionality of the dataset before the K-means clustering algorithm is applied. After performing the clustering, outliers instances, that are far away from all clusters are flagged

as incidents and are reported.

Another domain specific study by Landauer et al. [61] implements clustering techniques to perform log file analysis specifically for performing intrusion detection in Cyber Security. This particular study highlights a limitation of conventional clustering techniques for log file analysis, namely that static cluster groups cannot be used as a representative model of a computer system for an indefinite period of time. This is a result of systems that are upgraded or updated and, consequently, the log messages contained in the log file may change. To address this, the study implements cluster evolution analysis by generating clusters during separate windows of time. This technique can be used to perform anomaly detection online, in real-time.

LogCluster is another clustering based approach by [62] that is capable of performing online anomaly detection. It considers building a knowledge base of normal log clusters and uses this to determine if log messages received during production are anomalous. Log parsing is again implemented by separating log messages into log events and variables. Log messages are then clustered into groups based on similarity to other log messages. From these clusters, a single representative log sequence is extracted by identifying the centroid of the cluster. During the training stage, these representative log sequences are used to form the knowledge base. During production, the representative log sequences are checked against the knowledge base to see if they occur. If not, the sequence is flagged as anomalous. LogCluster aims to assist the debugging process performed by engineers by reducing the volume of log messages that need to be considered to identify the root cause of the failure.

Work by Liu et al. [63] considers the use of both clustering and classification techniques to detect anomalies in log files generated by IT systems. This implementation is specific in its consideration of system user behaviour, as many of the features extracted from the log files are generated from the user session information. These features are then passed to a K-Prototype clustering algorithm. This algorithm is favoured as the log messages contain a combination of numerical and categorical data. The clustering step groups all the log messages into normal and abnormal event categories. Following the creation of the clusters, filtering is then used to further determine which clusters contain anomalous events. This filtering is based on the assumption that normal events appear in greater numbers, so therefore, sparse clusters are likely to contain anomalies. This filtering stage is used to eliminate normal clusters to reduce the amount of data that needs to be processed in the next stage. The final stage is the anomaly detection stage which employs a k-Nearest Neighbour (k-NN) classifier to classify events as either normal or anomalous based on their distance to the previously found clusters. The combination of both clustering and classifications based methods improves the model's performance overall.

Although clustering techniques have received much attention in recent years, one comparative study by Zhu et al. [40] suggests that clustering techniques do not perform as well when compared to other methods that consider the inherent structure, content and semantics of log messages.

Natural Language Processing (NLP)

In contrast to the techniques previously discussed, work by Bertero et al. [64] considers the use of Natural Language Processing (NLP) techniques to treat the log messages contained in log files as raw text for the purposes of performing anomaly detection. The preprocessing step to parse log messages into a sequence of structured events is forgone, and instead, Natural Language Processing algorithms are used to create a vector representation of the words making up the log messages. These feature vectors are then used to train three classifiers, Naive Bayes, Neural Networks and Random Forest[38]. The main motivating factor for the use of NLP techniques to create features from log messages is to overcome the challenge often faced in parsing log files. Recent research has however shown that log parsing has received more attention and there are now various techniques that can achieve this [40]. Further more, while the work in [64] may more efficiently process larger volumes of log files, it does so at the cost of accuracy which could translate to an increase in the number of false alarms or missed detections. This is non-optimal in the context of debugging system failure.

Other methods

For completeness, techniques for automating log file analysis that do not rely on machine learning or deep learning are also briefly presented.

Work by Fu et al. in [39] investigates parsing log messages into log events, called log keys, and then using the sequence of these log keys to build a Finite State Automaton that represents normal workflows for expected system behaviour. Using this Finite State Automaton, workflow anomalies are detected by comparing new, incoming log messages to these models of expected workflows. That is, if the received log key sequence cannot be generated by any of the learned Finite State Automata, a workflow execution error is said to be detected. This model also considers anomalies in the transition time between various states of the Finite State Automata. While this method shows promise, it does rely on being able to build Finite State Automata for all valid workflows of the system. Depending on the system, such a dataset may be challenging to obtain. The idea of building workflows for normal system behaviour is also considered by Lou et al. in [65].

In efforts to assist software developers with error diagnosing, SherLog [66] makes use of the log files generated during the failed run and the source code of the failing program. With these log files and the source code, SherLog is able to map out various execution paths and categorises them as *must-paths*, *may-paths*, and *must-not-paths* to provide insights into which execution paths were followed, which may have been followed, and which were not followed during the failed run. This information is then presented to the system debugger and alleviates the need to attempt to reproduce the failure; an activity that is not always possible on live, deployed systems. SherLog requires a a substantial amount time to process log files, compared to to other methods, and generate the proposed set of execution paths. Additionally, the need for access to source code is a disadvantage as this may not always be readily available.

The use of invariant mining has also been explored in the context of log file analysis by Sharma

et al. [67] and Lou et al. [68]. Invariants, in these studies, are defined as relationships between metrics that are not expected to change during normal system operation. For example, the number of events related to closing a file is expected to be equal to the number of events related to opening that same file. If this relation is broken, it may indicate a system failure or anomaly. These methods also rely on log parsing to parse the log messages into log events and log parameters. One benefit of these methods is that the inferences are based on reasoning that human operators and debuggers can easily interpret. These methods do however fall short when the log sequences contain multiple tasks or represent multiple threads in a single task.

3.2.5 Summary

This section introduced Automated Log File Analysis and the various considerations required to implement this. Log parsing was introduced as an imperative preprocessing step and various log parsing techniques were introduced. The importance and relevance of good feature engineering was highlighted and was further explored for specific machine learning and deep learning methods. This section also modelled the research project's primary objective as being that of using Automated Log File Analysis for Anomaly Detection. Various machine learning and deep learning approaches to this problem were discussed.

From the research surveyed, the most promising machine and deep learning methods for automating log file analysis were found to be Recurrent Neural Networks and LSTM, Principle Component Analysis and Clustering techniques. Using RNNs, the log messages can be structured as a sequential dataset and time series analysis can also be performed. Both PCA and Clustering are unsupervised machine learning methods. The prominence of these methods alleviate the need for any labelled dataset which makes the application of machine learning to the problem of log file analysis more viable. Using machine learning and deep learning methods has shown to offer improved capability and usefulness of log file analysis and addressed many of the challenges faces with manual log file analysis methods.

One concern associated with these methods is their lack of interpretability. When such models are used to detect and flag system failures, system engineers and operators have little to no means of inspecting the decision-making process these models underwent to reach their conclusions. In some circumstances, more confidence in the model's performance is desired. As was discussed in this section, there has been some research undertaken with the aim of addressing the lack of interpretability of machine and deep learning models for this reason.

3.3 Selection of Tools and Frameworks

This section discusses various tools that will be used for the development and evaluation of the components required for the Automated Log File Analysis Framework.

The Data Miner and Inference Engine components together constitute an end-to-end machine learning pipeline. There are numerous frameworks across various programming languages that enable efficient design and implementation of machine learning pipelines. These frameworks enable rapid prototyping through leveraging pre-built algorithms, yet still allow for custom

model design and tuning as required. They also contain methods for quickly evaluating the performance and efficiency of trained models.

The Python programming language [69] has become increasingly popular for the development and deployment of machine learning and deep learning based models due to the availability of high performing frameworks. Owing to Python's increasing popularity in this field, the maturity of the language, abundance of supporting libraries for data-preprocessing and working with scientific data and the well-supported machine learning frameworks, it is chosen as the programming language for this research project.

3.3.1 Numpy

Numpy [70] is a powerful Python package that adds support for scientific computing in Python. It facilitates working with multidimensional arrays in Python and accelerates performance when working on these data types. Most machine and deep learning models require inputs structured as multidimensional arrays and the training of machine learning algorithms involves performing computations on multiple arrays that store parameters of the model. As a result, Numpy is a critical Python package when developing machine learning and deep learning models.

3.3.2 Pandas

Data pre-processing and mangling is an important step in machine learning pipelines. The pandas [71] library provides a flexible data structure designed specifically for performing real-world data analysis in Python. The pandas library supports different kinds of data and provides methods for easily cleaning, arranging, splitting, and reshaping datasets. It also provides functionality to support time series data. The pandas library is often recommend as the go-to Python library for data pre-preprocessing and initial data analysis and inspection [72].

3.3.3 Scikit-learn

The scikit-learn library is an open-source machine learning library for Python [73]. It provides a number of tools that are useful for creating models, performing data preprocessing, evaluating models and more. It supports both supervised and unsupervised machine learning methods and offers support for the most common models. Using this library, machine learning models can be rapidly built, tuned and evaluated as an initial step toward building a model for a particular problem. This library enables rapid prototyping and evaluation of various machine learning models.

3.3.4 PyTorch

The open-source PyTorch library for Python is a machine and deep learning package [74]. It is an end-to-end deep learning framework that facilitates rapidly building, evaluating and deploying deep learning models. Using PyTorch, the time required for, and complexity involved in, building large LSTM models is reduced. Through the the use of tensor computation, it also serves as an alternative to the popular Numpy package as it can leverage the compute capability of Graphics Processing Units (GPU).

3.3.5 Jupyter Notebooks

Jupyter Notebooks offer interactive computing through a web-based application [75]. These notebooks provide tools to jointly develop programs, document software, execute code and display results. Jupyter Notebooks are especially useful for the rapid prototyping stage of machine learning and deep learning projects and the interactive nature enables a quick feedback loop during experimentation. The Jupyter Notebook support back-ends, or kernels, from various programming languages including Python.

3.4 The Way Forward

This literature review provided the necessary background and context for the problem of debugging complex system failure. It also presented and explored the MeerKAT Correlator Beamformer as the case study system. Motivation for the use of log file analysis for debugging such complex systems was presented and the challenges and shortcomings associated with these methods were critiqued.

A need for more efficient, effective and robust methods to debug complex system failure steered the investigation toward automated log file analysis techniques. More specifically, this review considered data-driven machine learning and deep leaning based techniques as these methods have gained increased popularity in recent years. Research has shown that various machine and deep learning techniques may be used to perform log file analysis with the goal of detecting anomalies in system log files, with LSTM networks showing particular promise in terms of modelling expected behaviour from data contained in log files. Similarly, research has also highlighted the importance of preprocessing and parsing log files before implementing techniques for performing automated log file analysis.

With the necessary foundation laid, this research project now moves toward the development of the Automated Log File Analysis Framework (ALFAF) described in Chapter 1. This framework shall consist of two major components: a Data Miner and an Inference Engine.

The Data Miner will be responsible for pre-processing and parsing the log files obtained from the system of interest. It will implement log parsing as described in Section 3.2.2. The design and implementation of the Data Miner is presented in Chapter 4. The Inference Engine will be responsible for performing anomaly detection, using the features generated by the Data Miner, to flag, identify and detect failures that occur within a given system, from the information contained in log files. It will implement anomaly detection using an LSTM Neural Network as detailed in Section 3.2.4. The design and implementation of the Inference Engine is detailed in Chapter 5. With the context of the case study system as presented in Section 3.1.4, these components are then finally integrated to realise the complete, end-to-end ALFAF in Chapter 6.

4

Design and Development: Data Miner

With the Concept Exploration Phase of the project completed, sufficient information and detail has been acquired to inform the System Design Phase. This chapter describes the design and development of the Data Miner component of the complete end-to-end automated log file analysis framework described in Chapter 1. The process followed during this phase is detailed in Section 2.2.

Through a high-level overview of the Data Miner and detailed designs of all the sub-components making up the complete Data Miner, this chapter completely describes the Data Miner component of the framework. Testing and verification elements are considered during the design but are formally detailed in Chapter 7.

Informed by the research questions and project objectives, a set of guiding design considerations and constraints for the Data Miner is identified and collated. These considerations are used to guide the design and development process of the Data Miner and are described in Section 4.1.

To initiate the design process, with reference to the design considerations and system requirements, a high-level design of the Data Miner is presented in Section 4.2 to identify the major sub-components of the Data Miner, identify interfaces between the Data Miner, other framework components and the external system, and describe the high-level functionality of the Data Miner.

Following this, the design of the identified sub-components is detailed in Sections 4.3, 4.4 and 4.5. These sections detail the design of the Pre-Parser, Pre-Processor and Log Parser sub-components respectively.

Where and as applicable, the architecture of the software designed and developed for the various components and sub-components is described using UML Class and Activity diagrams.

With the design of all sub-components sufficiently detailed, the integration of these sub-components to realise the complete Data Miner component and the final design of the Data Miner is detailed in Section 4.6.

Lastly, the development of a tuning tool for tuning various log parsing algorithms is discussed in Section 4.7.

4.1 Design Considerations and System Requirements

The Data Miner is one of the components of the Automated Log File Analysis Framework (ALFAF) described in Chapter 1. The function of the Data Miner is to parse and process the raw, unstructured log files, containing log messages, from the system in question, into a structured dataset consisting of a sequence of events and extracted runtime variable parameters. As discussed in Section 3.2.2, log parsing is an important pre-processing function in performing automated log file analysis.

Guided by the research questions outlined in Section 1.2 and the literature reviewed in Chapter 3, the following design considerations are put forward to influence the design of the Data Miner:

Input Log Files The Data Miner shall be capable of parsing and processing log files, stored and available as raw text, containing log messages produced by any system. The Data Miner shall accept raw text files, containing log messages, as input. The Data Miner shall not be limited in terms of the size of log files it can accept as input. The log messages, contained in the log files, shall have a discernable *content* component that contains the actual log message and this shall be distinguishable from the rest of the log message.

Output Data The Data Miner shall, after processing input log files, generate a structured dataset consisting of a sequence of event templates corresponding to the sequence of log messages in the input file, and the extracted variable runtime parameters of each log message. This dataset may be used to build behavioural models of the system to determine if the correct sequence of events was followed during system operation/runtime. The availability of runtime parameters also facilitates anomaly detection. The Data Miner shall also output an event occurrence matrix that contains the occurrences of each event identified in the log file.

Interfaces The Data Miner shall have an interface to the Inference Engine. This interface shall be described by the format of the data passed from the Data Miner to the Inference Engine. The format of the data shall be machine learning model-ready e.g. a comma separated values (CSV) file that can be loaded into a matrix structure.

The Data Miner shall also have an interface to the system under test to access the log files. Depending on how the Data Miner is used, this may be a direct interface, or the interface may be realised through the encapsulating ALFAF.

Parsing Algorithm As evidenced by Section 3.2.2, log parsing algorithms typically vary in performance across datasets. The Data Miner shall implement various state-of-the-art log parsing algorithms to process and parse log files. The availability of multiple algorithms enables the evaluation of all algorithms in the context of the Data Miner to see how they perform. Many of the log parsing algorithms have parameters that may be tuned to affect the performance of the algorithm. The Data Miner shall expose an interface for configuring the parameters of the various log parsing algorithms that are implemented.

Usage The Data Miner shall be a modular, stand-alone tool that may be run independently of the ALFAF. This facilitates independent development and testing of the Data Miner before integrating into the framework.

Performance Metrics The Data Miner shall output various performance metrics after processing such as parsing accuracy and time taken to parse. This will aid in evaluation of the performance of the Data Miner across various datasets. These performance metrics are further detailed in Chapter 7.

These design considerations and guiding system requirements inform the design and development of the Data Miner in subsequent sections of this chapter.

4.2 High-Level Design: Data Miner

With the design considerations and system requirements clearly defined, this section presents a high-level design of the Data Miner. This high-level design is used to both identify the major subcomponents required and to fully describe the complete functionality of the Data Miner.

The high-level design is developed by considering the various processing stages that the log files need to undergo to transform them into a structured dataset, particularly one that can be fed as input to a subsequent machine learning algorithm. The high-level design, and the log file processing pipeline, is shown in Figure 4.1 below.

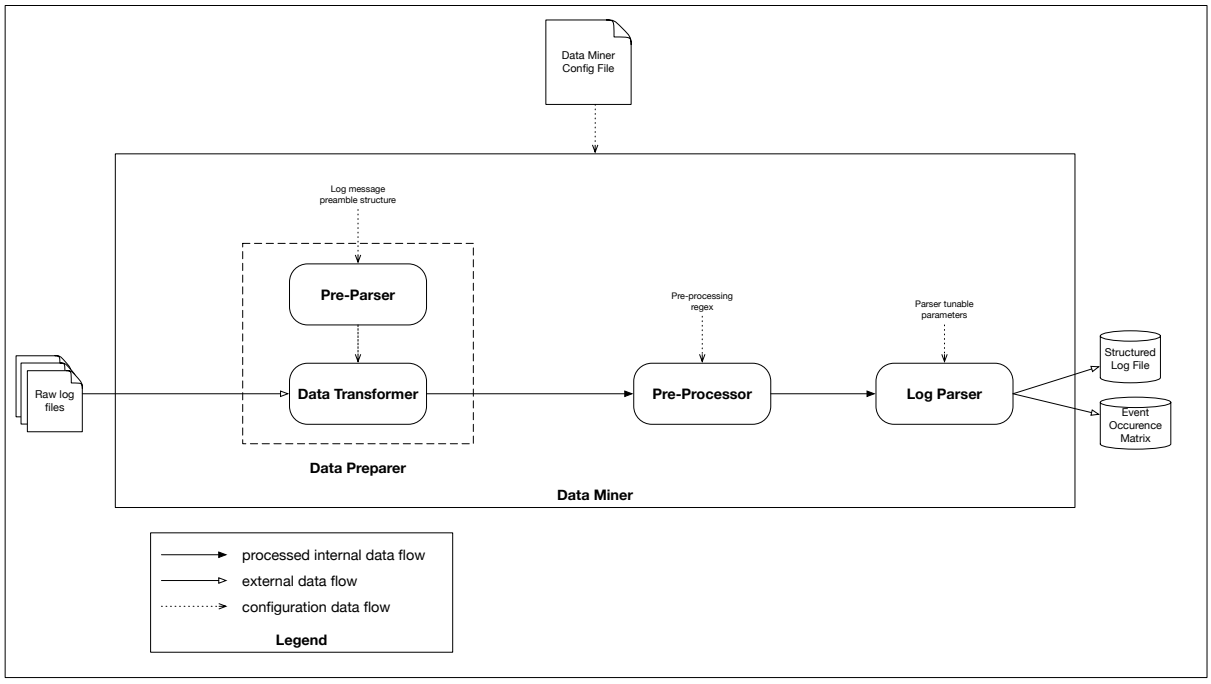


Figure 4.1: High-Level Design of the Data Miner component of the Automated Log File Analysis Framework. The major subcomponents of the Data Miner, the *Pre-Parser*, *Pre-Processor* and *Log Parser*, are illustrated alongside the inputs to and outputs from the Data Miner.

As shown in Figure 4.1, the Data Miner consists of three processing stages that are implemented by three subcomponents namely the *Data Preparer*, *Pre-Processor* and the *Log Parser*.

Together, these three stages transform raw, unstructured log files, that are supplied to the Data Miner as inputs, into a structured dataset consisting of a structured log file and an event occurrence matrix. The flow of data into and out of the Data Miner is indicated by hollow arrows in Figure 4.1. The three processing stages execute sequentially and the flow of data between these stages is indicated by solid arrows. The processing stages each also require configuration information and this is retrieved from a provided configuration file and passed to the relevant subcomponents.

The first processing stage, data preparation, is implemented by the Data Preparer subcomponent. Log files generally consist of multiple lines of unstructured, raw text. To enable any meaningful data-driven analysis and processing, these log files must first be transformed into a structured dataset. In order to transform the log files in this way, the Data Preparer performs two functions: pre-parsing and data transformation.

Although there is no defined standard for the format of log files and log messages, most modern systems have at least adopted some sort of informal structure in which the actual log message content is preceded by metadata such as a timestamp for the log message and the verbosity level at which the message was logged. As systems vary, the metadata also varies and may include information about the component, module or function that generated the log message. The function of the Pre-Parser is to parse each log message and extract the various metadata and the actual log message content. The Pre-Parser requires information about the general structure and format of the log messages. The design of the Pre-Parser subcomponent is detailed in Section 4.3.

Using the structure of the log messages extracted by the Pre-Parser, the Data Transformer splits the log message up into various components, including the log message content, and stores the extracted data from the entire log file in a matrix-like data structure for further processing. This is described in Section 4.3.

After data preparation, the processed log messages are then sent to the Pre-Processor for the second stage of processing. As discussed in Section 3.2.2, log messages usually consist of a static part that remains constant during different occurrences of the event and variable parts that contain runtime parameters that vary across different occurrences of the same event. One of the goals of log parsing is to identify the static and variable components of the log messages and to extract the variable runtime parameters. The variable runtime parameters are most commonly values, states, or other unique identifiers and descriptors.

The function of the Pre-Processor is to use provided system knowledge to match patterns corresponding to certain variable parts of the log messages expected to occur. The system knowledge describing the variable components is provided as a list of regular expressions. This optional processing step is implemented to improve the performance of later log parsing algorithms. By identifying and abstracting uniquely formatted variable components, it aids in log parsing when the the log messages are not varied enough, or do not occur frequently enough, to accurately detect variable and static components of the messages. The design of the Pre-Processor subcomponent is detailed in Section 4.4.

The final processing stage of the Data Miner is Log Parsing. This is implemented by the Log Parser subcomponent. The goal of Log Parsing is to take the log files and generate the structured dataset in which log messages are identified by event templates. During Log Parsing, the content of each log message is assessed to determine if an event template for it exists, or if a new event template should be created. This is continued for all log messages within a log file.

The output dataset is a structured sequence of event templates, along with the extracted runtime variable parameters, and an event occurrence matrix. The Log Parser is a highly configurable subcomponent of the Data Miner and may be tuned in terms of the log parsing algorithm to be employed and the various hyper-parameters associated with each algorithm. The design of Log Parser subcomponent is detailed in Section 4.5.

After the Log Parsing processing stage, the Data Miner function is complete and the output data is stored and available for further analysis. In the context of the Automated Log File Analysis Framework, the output of the Data Miner is fed to the Inference Engine, the component that performs the anomaly detection function of the framework. Being machine-learning based, the Inference Engine prefers data that can be easily read into a matrix-like data structure for further feature extraction, transformation and analysis. Although the design of the Inference Engine is detailed in Chapter 5, it is considered here as the interface between the Data Miner and the Inference Engine is detailed in Section 4.6.

The various subcomponents of the Data Miner require configuration information. To facilitate seamlessly providing the necessary configuration for all subcomponents, the Data Miner is designed to require and make use of a configuration file that specifies the various parameters and settings in a structured manner.

Finally, with all the subcomponents sufficiently designed and implemented, the integration of all the subcomponents and the design of the auxiliary supporting infrastructure, including the implementation of the configuration files, and interfaces of the Data Miner is detailed in Section 4.6.

4.3 Subcomponent Design: Data Preparer

As discussed in Section 4.2, the Data Preparer prepares the incoming raw log files for further processing and analysis and performs two functions: pre-parsing and data transforming. This section describes this processing stage in more detail and also describes the design and implementation of this subcomponent.

The Pre-Parser function parses each log message according to the log message format to extract the metadata and content of each log message. To more clearly illustrate the pre-parsing function, an example excerpt from a log file is shown in Figure 4.2.

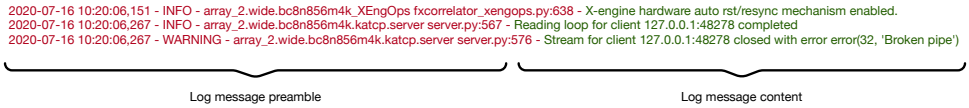


Figure 4.2: Example excerpt from a log file highlighting the log message preamble and the log message content

Figure 4.2 shows three log messages from the MeerKAT CBF system and clearly distinguishes the metadata from the actual log message. In this dissertation, the collection of log message metadata will be referred to as the *log message preamble*.

With sufficient knowledge about the design and operation of the MeerKAT CBF, the log message preamble can be further dissected to identify all the individual metadata components it comprises. This decomposition is shown in Figure 4.3 below.

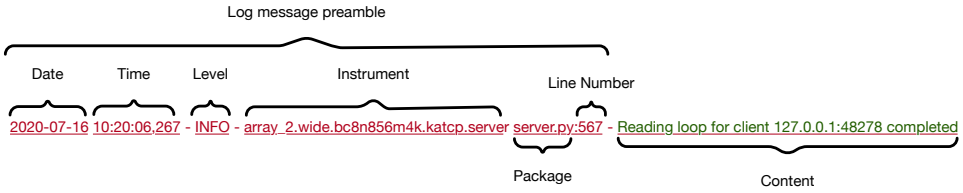


Figure 4.3: Decomposition of the MeerKAT CBF log message preamble

It can be seen that the log message preamble consists of the following metadata components: *Date*, *Time*, *Level*, *Instrument*, *Package* and *Line Number*. The actual log message component is simply referred to as *Content*. The goal then, of the Pre-Parser, is to extract and identify the log message preamble and its components, and to identify and extract the log message content for further analysis.

The Pre-Parser is adapted from work by Zhu et al. and their work on the LogParser toolkit [40][35][7]. The Pre-Parser implementation requires the format of the entire log message, with the preamble decomposed into its constituents, to be provided as a text string. The Pre-Parser then uses this format to generate a regular expression sequence for matching and extracting the various components of the entire log message and also extracts the names of the various components of the log message to be used as headings in the final structured dataset that is created by the Data Transformer. An example of the input and outputs of the Pre-Parser is shown in Figure 4.4.

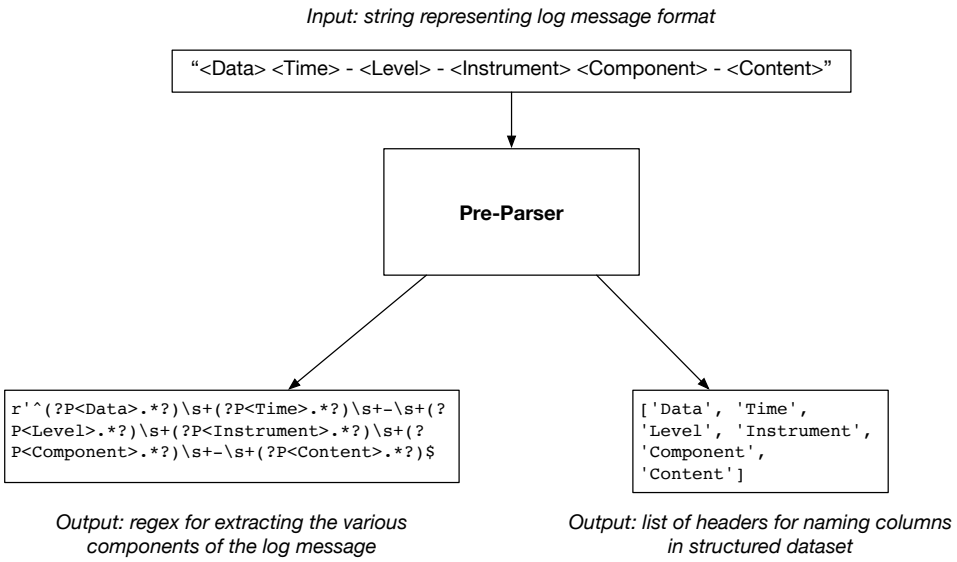


Figure 4.4: Input to and outputs of the Pre-Parser subcomponent

As shown in Figure 4.4, the components of the log message are specified using enclosing $<,>$ characters and the delimiters separating the components are explicitly indicated. The Pre-Parser uses these delimiters, including white space characters, as the basis for generating the regular expression that can extract all the various components from the log message. The grouping feature of regular expressions is used to individually group the various components of the log message for ease of accessibility later in the processing pipeline.

The Pre-Parser is implemented as a Python function and only requires a string representing the log message format and generates a regular expression for pattern matching. The function makes use of the built-in Python *re* module [76] for compiling the regular expression used for pattern matching and for splitting the input string based on the occurrence of enclosing $<,>$ characters. The functionality of the Pre-Parser is illustrated through a UML Activity Diagram shown in Figure 4.5.

The Pre-Parser initially creates an empty string for storing the regular expression pattern to be used to extract the various components of the log message. It also initialises an empty list for storing the names of the components of the log message. The provided input string is split on the occurrence of enclosing $<,>$. This is achieved using the *re.split* method and results in a list of substrings containing the named components and delimiters. Each of the resulting substrings are then further processed until none remain.

If a given substring contains a component name, the name is extracted from the substring by removing the enclosing $<,>$ characters. The name is then appended to the list of component names. To pattern match the particular component in a given log message, which may be of varying length and contain an arbitrary collection of characters, *(.*)* is appended to the log format regular expression. The sequence *(.*)* will match 0 or more of any character.

If a given substring instead does not contain a component name, it may be regarded as a delimiter. The regular expression generated to extract the components of a log messages relies

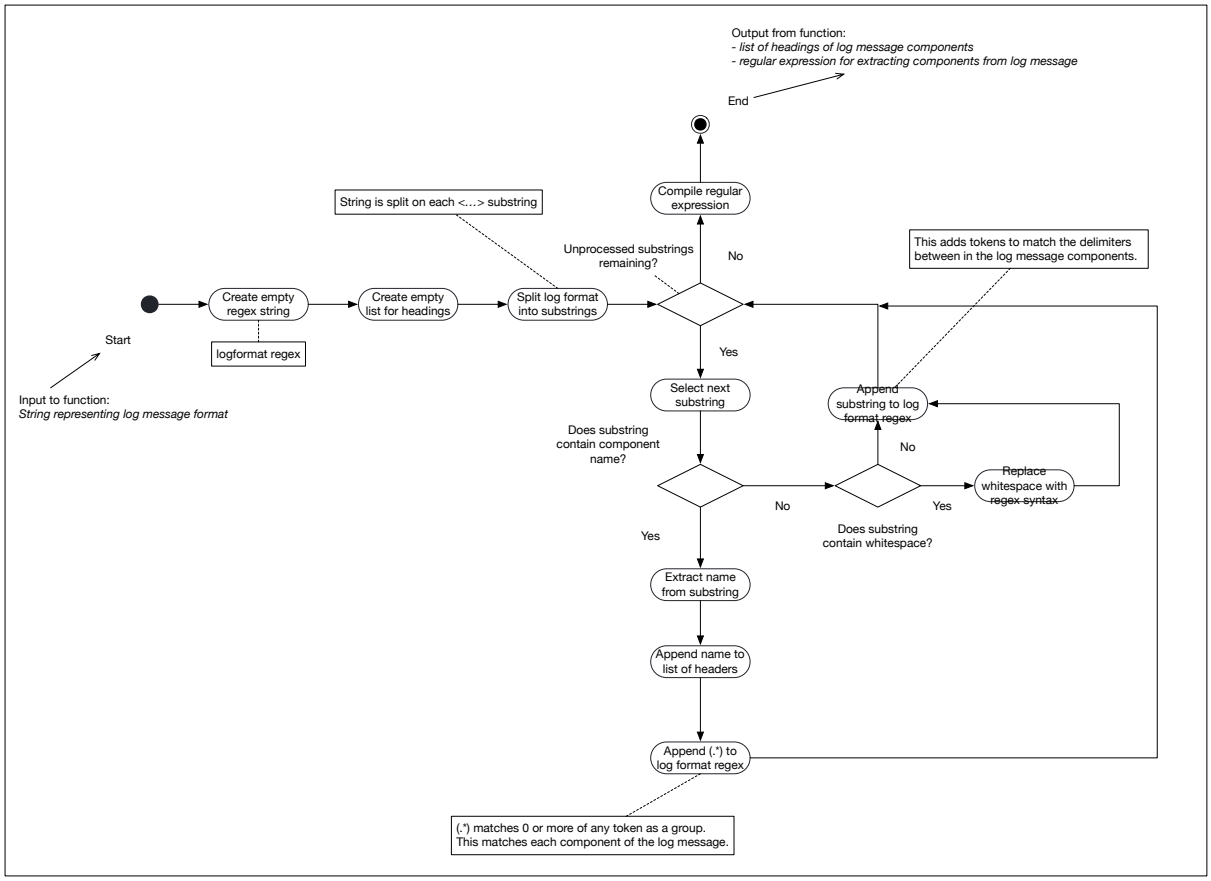


Figure 4.5: UML Activity Diagram illustrating the functionality of the Pre-Parser process

on the delimiters between components to ensure correct decomposition. If a delimiter substring contains white space, they are replaced by the regular expression syntax for white space before being appended to the log format regular expression. If the delimiter substring does not contain white space, the substring is simply appended to the log format regular expression.

Once all substrings have been processed, the log format regular expression is compiled and is returned alongside the list of component names. The resultant regular expression is able to match and extract the various components, including the message content, of a given log message corresponding to the provided log message format.

The Pre-Parser stage does not use, nor does it require, the actual log files. Instead, it generates information used by the Data Transformer.

The Data Transformer uses the regular expression generated by the Pre-Parser to extract the various components of the log message preamble and the log message content. Using the extracted preamble headers, the Data Transformer processes the the raw log messages into structured, matrix-like dataset in which each row is a unique log message and each column is a component of the log message. This structured version of the raw log files allows for data manipulation and processing by later stages in the pipeline.

4.4 Subcomponent Design: Pre-Processor

As discussed in Section 4.2, the goal of the optional Pre-Processor is to identify and abstract certain variable components contained within the log messages. This function is informed by regular expressions, derived and generated by system expert knowledge, that describe the format of certain variable components expected to be found in the log messages of a particular system. The function performed by the Pre-Processor is illustrated in Figure 4.6.

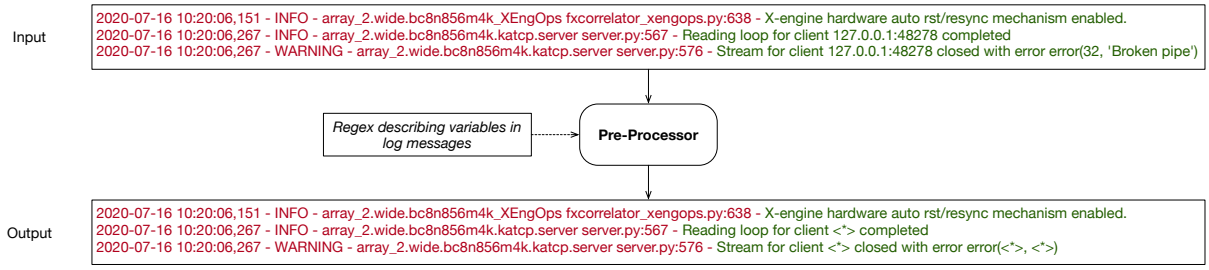


Figure 4.6: Function of the Pre-Processor. Variable components of log messages described by regular expression patterns are replaced with wildcard tokens: $< * >$

As seen in Figure 4.6, after undergoing pre-processing, log messages, specifically the message content component, that contain variable components matching the given regular expressions are transformed to have these variables replaced with $< * >$ tokens. These tokens are used to represent the variable components of log messages and the presence and location of these wildcard tokens is used to extract the variable components of the log messages in the Log Parsing function described in Section 4.5.

The Pre-Processor is implemented as a Python function that makes use of the *re* module's *sub* method to replace any matching occurrences of a given pattern with wildcard tokens, $< * >$. The function takes in a list of regular expression patterns and the content portion of a single log message. The UML Activity Diagram in Figure 4.7 illustrates the functionality of the Pre-Processor.

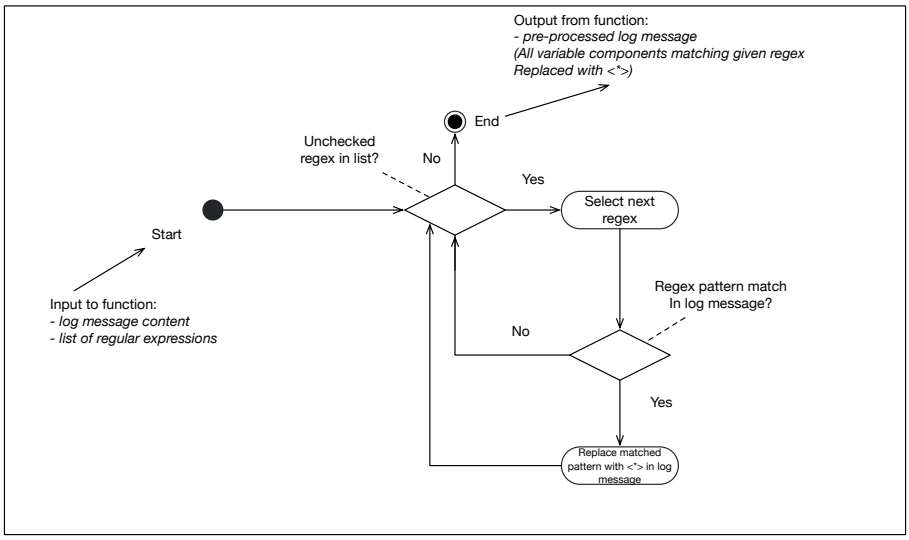


Figure 4.7: UML Activity Diagram illustrating the functionality of the Pre-Processor

The Pre-Processor first determines if there are unchecked regular expressions in the given list. If there are regular expressions still to be checked, the Pre-Processor selects the next one and checks if the pattern is present in the log message. If the pattern is present, all occurrences are replaced with the wildcard token, $< * >$. If the pattern is not present, the log message remains unchanged. This process repeats, for a given log message, until it has been checked against all provided regular expressions. Once all regular expression patterns have been considered, the pre-processed log message is returned by the Pre-Processor.

This process is then repeated for every log message contained in a given log file provided to the Data Miner for log parsing. Once all log messages have undergone pre-processing, they are passed to the next stage in the processing pipeline, the Log Parsing stage.

If no regular expression patterns are provided, the Pre-Processor function is effectively bypassed and the log messages, in their structured dataset format as generated by the Data Preparer, are simply passed to the next processing stage.

4.5 Subcomponent Design: Log Parser

As described in Section 3.2.2, the goal of log parsing is to transform an unstructured set of log messages into a structured, sequenced set of events. During log parsing, a set of event templates describing all possible log messages is extracted from the given log files. These event templates identify the static and variable parts of the log messages. After log parsing, each log message is represented by an appropriate event template and a list of variable parameters that occurred in the log message. This transformation enables more efficient pattern extraction and feature engineering from the log messages contained in a given log file. For the Data Miner, the Log Parser sub-component performs this function as illustrated in Figure 4.8 below.

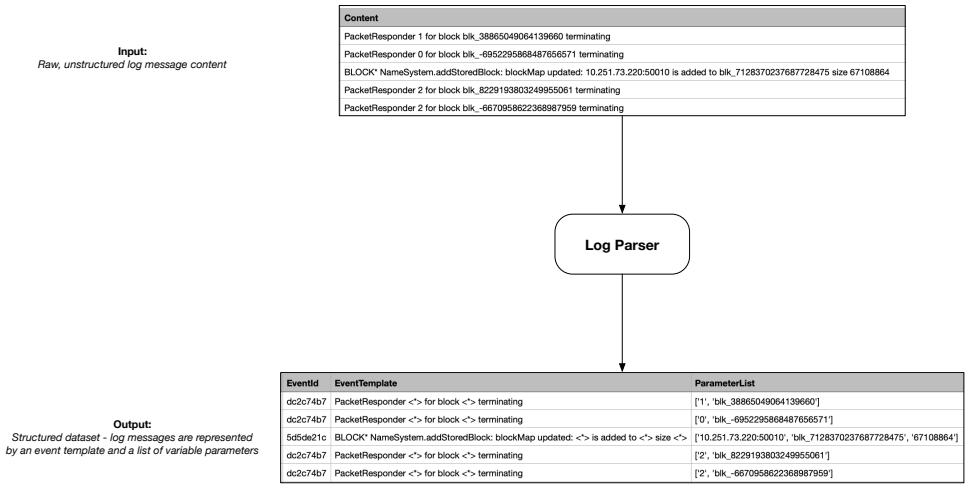


Figure 4.8: Function of the Log Parser. A log file containing multiple log messages is transformed into a structured dataset consisting of a sequence of events corresponding to each log message

As shown in Figure 4.8, the input to the Log Parser subcomponent is the content component of each log message that was extracted and pre-processed in prior stages. At this point in

the pipeline, some variable parameters may have already been replaced by wildcard tokens as a result of the pre-processing stage. The Log Parser attempts to identify the static and variable components of the messages and extracts event templates to describe the set of log messages. Once all log messages in the given dataset are described by an event template, the variable components of each log message are extracted to populate a parameter list for each log message. A unique identifier is also generated for each event template and this is also included in the structured output produced by the Log Parser. The output of the Log Parser, including the event identifier, event template and list of parameters describing each log message, is shown in Figure 4.8.

The functionality of the Log Parser relies on the implementation of data-driven approaches to log parsing. Research has shown that various approaches achieve varying degrees of success on log files from various systems. As a result, to improve the robustness and generalisability of the Data Miner, the Log Parser is designed to offer a choice in the parsing algorithm that is employed.

The following subsection describes the log parsing algorithms that are considered and that are supported in the Log Parser subcomponent.

4.5.1 Log Parsing Algorithms

While traditional methods of log parsing relied on the existence of manually created regular expressions or rules to extract event templates and variable parameters from log messages, recent works in achieving automated log parsing have considered data-driven and statistical approaches. Thirteen of the most recent, state-of-the-art automated log parsing algorithms have been reviewed, implemented and benchmarked by Zhu et al. in [40] and [35], and their work has resulted in an open-source toolkit, *logparser*[7], that implements these algorithms in Python. The algorithms were benchmarked across sixteen different log file datasets generated by various systems including supercomputers, mobile systems, operating systems, distributed systems and server applications. Metrics for the benchmarks include parsing accuracy, robustness and efficiency. These metrics are described further in Chapter 7.

Based on the findings presented in [40] and [35], seven of the log parsing algorithms are ruled out as suitable candidates for the implementation of the Log Parser for the Data Miner. The algorithms that were ruled out are briefly presented below for completeness, along with motivation for their non-consideration. As discussed in [35], even small differences in parsing accuracy can have a considerable impact on the performance and results of later data analysis, and therefore, this metric is used as the primary deciding criteria. In addition, algorithms that were ruled out were evaluated in terms efficiency and robustness. The reader is referred to the cited papers of each algorithm for more detail.

- *LogSig* [46] - a clustering-based log parsing algorithm. *LogSig* performed poorly across the datasets with an average parsing accuracy of 48.2%. The time required to run the algorithm also significantly increased as the size of the dataset increased.
- *LKE* [39] - also a clustering-based algorithm. *LKE* was the second worst performing

algorithm across the sixteen datasets. It also exhibited low efficiency and had difficulty with small datasets.

- *MoLFI* [77] - this algorithm only performed well on datasets with comparatively simple log messages and event templates. It also exhibited low efficiency when tasked with parsing larger datasets. Its implementation in Python also requires specific versions of additional dependencies.
- *SLCT* [18] - an algorithm based on frequent pattern mining. *SLCT* failed to achieve a parsing accuracy of greater than 88% across the sixteen datasets.
- *LFA* [78] - another frequent pattern mining based algorithm. *LFA* builds on *SLCT*. While it offers some improvement, it still falls short in the same manner.
- *LogCluster* [47] - also based on frequent pattern mining. *LogCluster* performs better than both *SLCT* and *LFA*, but only achieves an accuracy of above 90% on one of the sixteen datasets.
- *SHISO* [79] - a clustering-based algorithm. While it achieves an average parsing accuracy of just under 70%, it performs very poorly on some datasets. The configuration of the algorithm is also not well defined in the literature.

The remaining six log parsing algorithms that have not been discarded are selected for inclusion as available algorithms in the Log Parser subcomponent of the Data Miner. These algorithms have been selected because they exhibited high efficiency and achieved generally good parsing accuracy across the sixteen datasets they were benchmarked on in [40] and [35]. These algorithms are briefly summarised below. The reader is again referred to specific literature of the various algorithms for more details.

It should be noted that many of the log parsing algorithms have tunable parameters that influence their performance across different log message datasets. These parameters are highlighted, where applicable, but the reader is informed that the tuning of these parameters is further addressed in Section 4.7.

AEL

The Abstracting Execution Logs (*AEL*) algorithm employs a combination of rules-based and data-driven clustering approaches to abstract log messages to execution events[48], [49]. The AEL algorithm consists of four main steps: *Anonymize*, *Tokenize*, *Categorize* and *Reconcile* as shown in the overview of the algorithm in Figure 4.9.

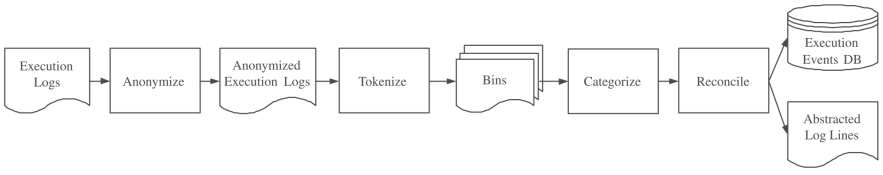


Figure 4.9: Overview of the AEL log parsing algorithm. Image source: [49]

The *Anonymize* step uses a defined set of rules to identify variables that occur within the log messages and replace them with a wildcard token. This is similar to the function performed by the Pre-Processor as described in Section 4.4. For the implementation of the *AEL* algorithm for the Data Miner, the *Anonymize* step is performed by the Pre-Processor. The *Tokenize* step then groups the now anonymized log messages into different groups, or bins, based on two criteria: the number of tokens or words in each message and the number of wildcard tokens in each message. For example, *Bin 3_0* contains log messages that consist of three tokens in total, and no wildcard tokens, while *Bin 5_1* contains log messages that consist of five tokens, one of which is a wildcard. The *Tokenize* steps aims to reduce the search space in the later processing steps to improve performance and efficiency of the algorithm. The third step, *Categorize*, performs the abstraction and generates execution events, or event templates, for each log message. It does this by comparing the log messages in each bin to determine whether or not they may be described by the same event template. The *Categorize* step starts with an empty list of event templates for each event. It then considers the first log message in the bin, and since the list of event templates is empty, a new event template, corresponding to the log message, is created and appended to the list. The next log message in the bin is then considered. First, it is now compared to the event templates in the list, if a match is found, the log message is abstracted to the corresponding execution event. If no match is found, a new event template is created. This process is repeated for all log messages within a given bin, until all bins have been processed. The final step, *Reconcile*, address the case in which not all variables are replaced with wildcards in the *Anonymize* step. During this step, all execution events within a given bin are re-examined to determine which could be merged based on two rules. Firstly, the execution events must differ, in terms of tokens in specific position, by some threshold, and secondly, execution events are only considered for merging if the number of events to be merges exceeds a specific threshold. The *Reconcile* step leads to tunable parameters of the *AEL* algorithm: the *threshold* specifying the minimum number of events that should be considered for merging and the *event difference* that specifies the maximum allowable difference, calculated as a percentage of different tokens between events, that is considered for merging.

Drain

Drain is a clustering-based log parsing algorithm that parses log messages in a streaming manner as opposed to batch processing all the log messages in a given log file [44]. The clustering is performed through the use of a parsing tree to accelerate clustering performance. This offers improved performance compared to other clustering methods as the number of clusters that are ultimately considered is bounded. An overview of *Drain's* parsing tree structure is shown in Figure 4.10.

At the very top of the parsing tree lies a *root node* and the tree ends in *leaf nodes*. Nodes between the *root* and *leaf nodes* are called *internal nodes*. The *root* and *internal nodes* are used to encode a sequence of rules used for clustering and do not contain any log messages. Log messages traverse down the tree through a number of *internal nodes* and each path terminates with a *leaf node*. The *leaf nodes* contain log groups - data structures that contain the event template that best describe log messages belonging to the group and the log IDs of the the log

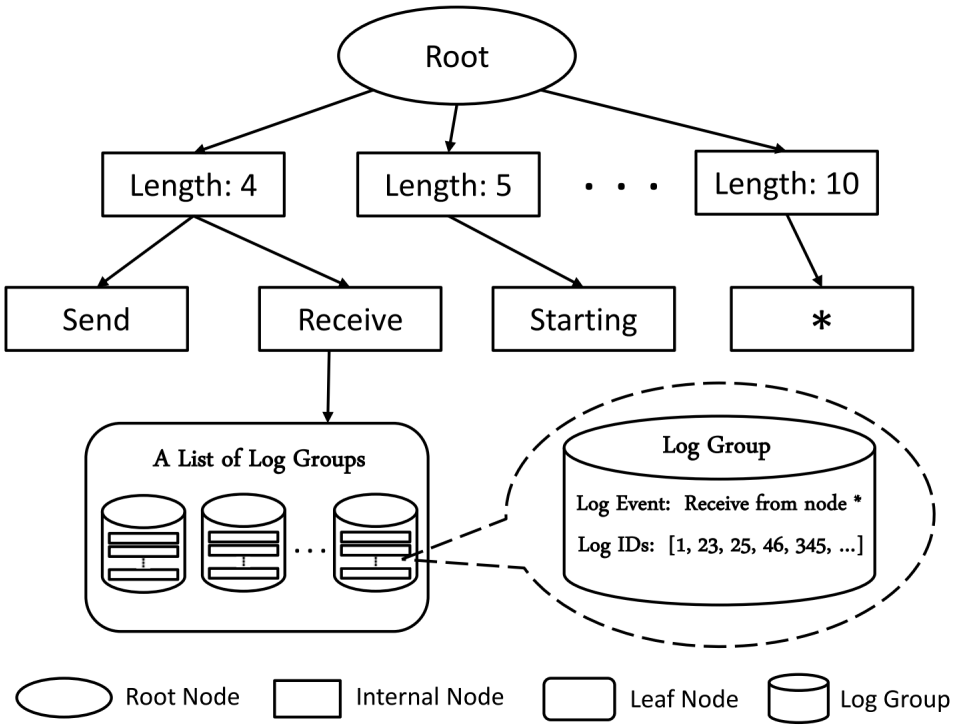


Figure 4.10: Overview of the Drain log parsing algorithm's parsing tree structure. A tree depth of 3 is shown. Image source: [44]

messages that belong to the group.

Before traversing the parsing tree, log messages undergo preprocessing to match variable components of log messages and replace these with wildcard tokens. In the implementation of the Data Miner, this function is performed by the Pre-Processor. Once preprocessed, the log messages then traverse the parsing tree until a suitable *leaf node* is found.

The first internal layer of the tree encodes the log message length, and the path to the first internal node is selected based on the length of the log message. This is based on the assumption that log messages generated by the same event are likely to have the same message length.

After the first internal node in the tree is selected, paths to subsequent internal nodes are selected based on the sequence of tokens in the log message. The number of internal nodes traversed, and the total length of the parse tree, is controlled by the parameter of the algorithm called the *parse tree depth*. The number of internal nodes traversed is equal to *parse tree depth* $-$ 2, which results in the first *parse tree depth* $-$ 2 tokens of the log messages being used as search rules in the algorithm. To account for an infinite number of possible tokens, and for the fact that some tokens may be variables, a parameter called *maxChild* is introduced to control the maximum number of children nodes that can spawn from any given node. Additionally, all tokens that contain digits are matched to a special internal node denoted by * in Figure 4.10. Once *maxChild* is exceeded any new tokens that are found are matched to the special internal node.

After traversing the entire tree depth as specified by *parse tree depth*, a *leaf node* is found. The

log messages in the log groups of the *leaf nodes* satisfy all the rules encoded by the internal nodes along the path. At this point, the most suitable log group is selected from the list of log groups in the leaf node based on similarity between the log messages and the representative event template of the log groups. The algorithm first finds the log group with which the log message exhibits the highest similarity score. Following this, this score is compared to a *similarity threshold*, another parameter of the algorithm. If the score is greater than this threshold, then the log group is selected as the most suitable group. If not, a new log group is created.

Once a suitable log group is found, the log ID of the log message is added to the list of IDs of the log group and, if necessary, the event template describing the group is updated by comparing the tokens of the event template to the new message and replacing any differences with wildcards. In the event that a new log group is created, *Drain* updates the parsing tree to include a new path that leads to the new log group.

IPLoM

Iterative Partitioning Log Mining, or *IPLoM*, is an automated log parsing technique that performs a three-step partitioning process to group log messages that can be described by the same event template into clusters [43], [42]. After partitioning all log messages into clusters, the event message template that describes the log messages contained in a cluster is extracted. An overview of the *IPLoM* technique is shown in Figure 4.11 below.

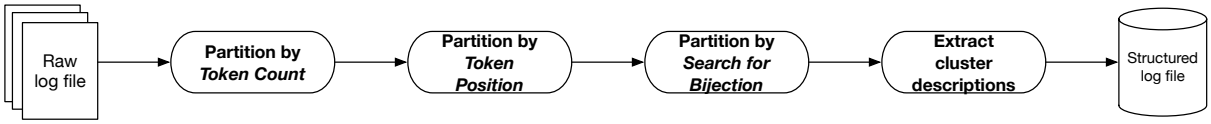


Figure 4.11: Overview of the *IPLoM* log parsing technique

The first stage of partitioning uses the token count of the log messages as the partition criteria. This stage assumes that log messages that are generated by the same event are likely to have the same token length. This stage partitions log messages into clusters of log messages that have the same total token length.

The second stage of partitioning is carried out on each partition found during the first stage. These partitions now contain log messages that have the same length. This second stage then uses the position of tokens as the criteria for further partitioning. In this stage, the token position with the fewest unique possible tokens is identified and the the log messages are split into further partitions based on the unique possible tokens in that position. This partitioning is based on the assumption that the token position with the least number of unique possible words is likely to contain words that represent constant parts of the log messages. In the event that a chosen token position actually represents a variable part of the log message, the ratio between messages in the original partition and the new partition is compared to a *partition support threshold* parameter, and this is used to decide whether a partition is valid or whether it should be merged with other partitions.

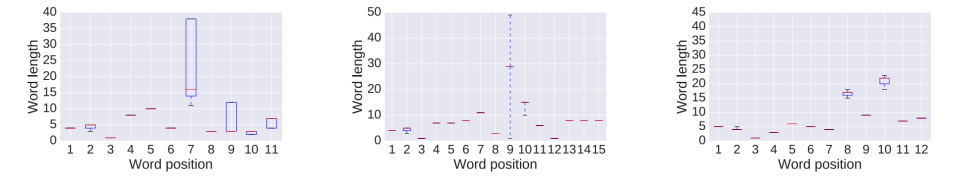
The final partitioning stage partitions the messages in each cluster based on the occurrence of

bijective relationships between the sets of tokens that occur in two token positions. The token positions that are considered are chosen based on the unique token count values for each token position. Before this partitioning takes place, each incoming cluster is assessed to determine whether it is good enough and does not require further partitioning. This assessment considers the ratio between the number of token positions that only have one unique value to the total token length of the messages in the cluster. This ratio is compared to a *cluster goodness threshold*.

After all partitioning stages are complete, it is assumed that each cluster contains log messages that were generated by the same event and that may be represented by the same event message template. The final stage of the algorithm attempts to extract these event message templates for each cluster. This is performed by counting the number of unique tokens for each token position. If a token position only contains one unique value, that token position is considered to be constant and that value is used in the event template. If a token position contains multiple possible values, that token position is considered to be a variable part of the message and is represented by a wildcard token, $< * >$, in the event template.

LenMa

The Length Matters, or *LenMa*, algorithm is an online processing log parsing algorithm that extracts log message event templates based on the length of the words contained in the log messages[80]. This approach assumes that the fixed or static parts of the log messages will have the same word lengths across messages generated by the same event, and that while the variable components of log messages will have varying lengths, they will be similar because they are expected to share the same context e.g. IP address, process number. A sequence of word lengths representation of various log messages is shown in Figure 4.12.



(a) `sshd * : Received disconnect from * 11: * * *` (79182 samples)

(b) `sshd * : reverse mapping checking getaddrinfo for * * failed - POSSIBLE BREAK-IN ATTEMPT!` (11926 samples)

(c) `named * : DNS format error from * resolving * invalid response` (878 samples)

Figure 4.12: Log messages represented as a sequence of varying word lengths. Image source: [80]

From Figure 4.12, it can be seen that each log message has a unique sequence of word lengths and *LenMa* exploits this feature to perform clustering to extract log message event templates. In addition to considering the length of words in a log message, the algorithm also considers the position of each word, and therefore the word length, in the message.

LenMa perform online processing and as such, determines the most suitable cluster for each new incoming log message. If no suitable cluster exists, a new cluster is created. Initially, each log message is converted into a *word vector* containing all the words of the log message and a *word*

length vector that contains the lengths of each word, in their respective positions, in the log message. Each cluster is described by a single representative event template with corresponding *word* and *word length vectors*. The *word length vectors* of new log messages are compared to that of each existing cluster that is represented by a *word vector* containing the same number of words. The comparison is done using the cosine similarity equation, also taking into account the position of the words, and the cluster with the highest similarity score is considered as a candidate. If this similarity score is greater than a tunable threshold, the *cluster similarity threshold*, then the cluster is chosen for that log message. As a final check before deciding that the new log message belongs to the chosen cluster, the number of shared words in the same positions, between the log message and cluster *word vector*, is considered and if it falls below a given threshold, that cluster is not chosen for the given log message and a new cluster is instead created. Alternatively, if the similarity score is less than the threshold, a new cluster is created for the log message.

In the event that a new cluster is created, the representative *word vectors* and *word length vectors* are updated with information from the new log message. If any words, in any given position, differ, these are replaced with wildcards in the cluster *word vector*. Similarly, any discrepancies in the *word length vector* are replaced with the values from the new log message.

LogMine

LogMine is a clustering-based log parsing algorithm that generates a hierarchy of log message event templates[45]. Event templates at the top of the hierarchy are the most general templates, containing the most wildcard tokens, and templates on the lower levels of the hierarchy are more specific, containing specific substring patterns. This approach allows for flexibility in that more general or more specific event templates may be extracted. An overview of *LogMine's* algorithm is shown in Figure 4.13.

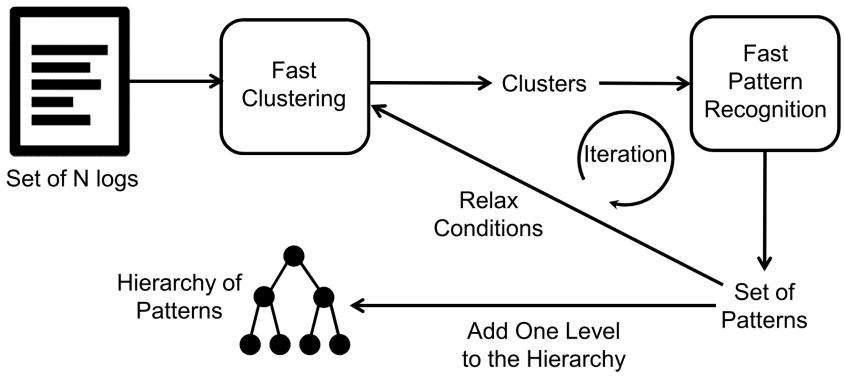


Figure 4.13: Overview of the *LogMine* algorithm's workflow for generating a hierarchy of log message event templates. Image source: [45]

LogMine processes a batch of log messages contained in a log file. Initially, log messages are first clustered with very strict clustering criteria. A distance function is used to determine how similar log messages are to one another and the strictness of the clustering is specified by tuning the maximum distance allowable between messages through a parameter called *maxDist*. Messages that are very similar will have a smaller distance between them, while messages that

are different will have larger distances between them. After clustering, a pattern is generated for each cluster by merging all log messages in the cluster and replacing differing tokens with wildcards. The sequence in which the log messages in a cluster are merged is of little significance since the algorithm assumes that the messages are very similar. This initial set of patterns, derived for each cluster, is added as the lowest level of the pattern hierarchy.

The processes of clustering and pattern extraction is then repeated, this time, with clustering being performed on the set of patterns, and the clustering criteria relaxed by increasing *maxDist*. This allows for more general patterns to be clustered together. Once clusters have been created, representative patterns are then extracted for each cluster from the set of patterns in each cluster. These representative patterns are added as the next level, one level up, of the hierarchy and are used in the next iteration of clustering and pattern extraction.

This process continues iteratively until the most general pattern, one that describes all log messages and patterns on the lower levels of the hierarchy, is found. In each iteration, the adjustment of *maxDist* is handled by another parameter, *alpha*, which scales *maxDist* for each iteration of the clustering process.

Spell

The *Spell* algorithm implements log parsing in an online streaming manner and extracts log message event templates using an approach based on identifying the longest common subsequence across sequences [50]. Identifying the longest common subsequence differs from identifying the longest common substring in that for subsequences, it is the order of the individual elements that matters more and not their absolute position within the sequences. As an example the longest common subsequences for the sequences $ABCD$ and $ACBAD$ are ABD and ACD.

In the context of log parsing, the longest common subsequence approach is employed under the assumption that when viewing log messages as a sequence of tokens or words, the static or fixed part of the event messages will make up the majority of the sequence compared to the variable components. Additionally, the longest common subsequence between two log messages that are generated by the same event, and that only differ in variable parameter values, is likely to be the event message template describing those logs. *Spell* employs the longest common subsequence approach, with clustering, to extract event templates for given log messages. An overview of the algorithm is shown in Figure 4.14.

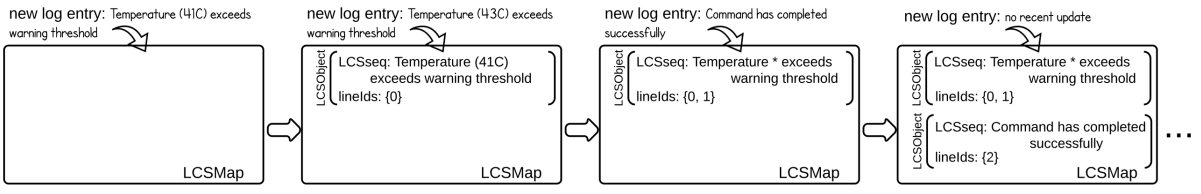


Figure 4.14: Overview of the Spell algorithm's workflow. Image source: [50]

Spell uses a data structure called $LCSObject$ that contains the longest common subsequence of multiple log messages, i.e. the event template, called $LCSseq$ and a list of indices that refer to the

log messages that are described by the event template stored in *LCSseq*. Multiple *LCSObjects* exist, one for each event template found, and the collection of these objects is stored in a list called *LCSMap*. This structure is illustrated in Figure 4.14.

When a new log message arrives for parsing it is first parsed into a sequence of tokens based on delimiters and is assigned a unique line index. The tokenized message is then compared to the *LCSseq* of all the *LCSObjects* in the *LCSMap* list. This comparison involves computing the longest common subsequence between the tokenized message and each *LCSseq*. After computing this across all *LCSObjects*, the length of the longest common subsequence found is then compared to a configurable threshold. If the length of the longest common subsequence is greater than the threshold, then the incoming log message is considered to correspond to the event template described by *LCSseq*. In the event that multiple longest common subsequences between the log message and the *LCSObjects* are found, then the *LCSseq* with the lowest token count is considered. Finally, the *LCSseq* is then updated to be the longest common subsequence found between the previous *LCSseq* and the log message and the list of indices is updated to include the new log message.

If the length of the longest common subsequence is smaller than the threshold, then a new *LCSObject* is created and the tokenized log message is used as the *LCSseq*. This new *LCSObject* is then added to the list of objects.

Initially, *LCSMap* is empty when the first log message for parsing arrives. This message is parsed into a sequence of tokens based on the delimiters contained in the message and the first *LCSObject* is created.

The *Spell* algorithm improves processing efficiency by applying a pre-filtering step to limit the search space of which *LCSObject* each log message is compared to. This filtering is based on comparing the length of the incoming log message to the length of each *LCSseq*.

4.5.2 Log Parser Design

For the Data Miner, the core of the Log Parser is adapted from the *logparser*[7] toolkit implemented by Zhu et al. [40], [35]. This open-source toolkit implements the various log parsing algorithms as described in Section 4.5.1.

logparser implements the log parsing algorithms in Python. The algorithms of interest are implemented in Python 2.7. To keep in line with the rest of the Data Miner development, the implementations of these algorithms are ported to Python 3. The scope of the changes required to ensure Python 3 compatibility include the correct use of relative and absolute import statements and using updated escape characters for defining the patterns for use with the *regex* module. The *logparser* module also has a number of dependencies including *scipy* [81], *numpy* [70], *pandas* [71] and *scikit-learn* [73].

Each algorithm is implemented as a separate *LogParser* class. The different classes share the same interfaces in terms of the log files provided as input and the output produced, as well as the main methods that invoke the automated log parsing process. The classes differ in terms

of the attributes and methods for implementing the algorithm. A UML Class Diagram for the generalised *Logparser* class is shown in Figure 4.15.

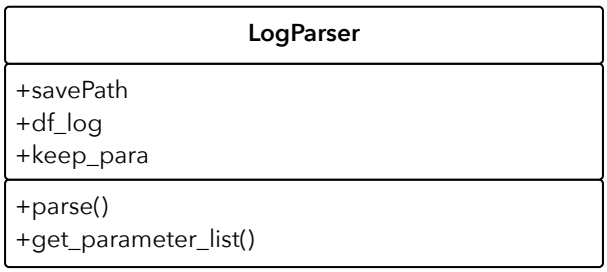


Figure 4.15: UML Class Diagram of the general *Logparser* class showing the common attributes and methods across parsing algorithms

Figure 4.15 only shows the attributes and methods that are common across all versions of the *LogParser* class for the various algorithms.

In terms of class attributes, the *savePath* attribute stores the path to the directory where the output data structured are to be stored. *df_log* stores the pandas *DataFrame* of the pre-parsed and pre-preocessed log file after the initial data preparation and pre-processing stages. *keep_para* is a flag that enables or disables the extraction and storage of runtime variable parameters from the log messages after the parsing process.

The main method of the *LogParser* class is the *parse* method. This method performs the actual parsing function by calling algorithm specific methods that result in the execution of the log parsing algorithm. When specified to run, the *get_parameter_list* method extracts the variable runtime parameters from log messages based on the derived event templates.

For the implementation of each algorithm, additional attributes and methods are defined. The methods are specific to the implementation of each log parsing algorithm. The attributes also include the configurable parameters of the algorithms that are passed to the *LogParser* class during object instantiation.

The instantiation of various *LogParser* objects, along with the appropriate initialisation parameters is handled by the higher-level Data Miner component as described in Section 4.6.

4.5.3 Developing a Regex-based Log Parser

In addition to the automated log parsing algorithms described in Section 4.5.1, a manual, regular expression-based log parser is also developed as a method that may be employed by the Data Miner. This log parser extracts event templates and variable runtime parameters from log messages based on a set of pre-defined regular expression patterns.

This regex-based log parser is designed to enable the generation of a ground truth dataset to be used in evaluating the performance of other log parsing algorithms. Automated log parsing algorithms will also be compared to this regex-based log parser in terms of implementation complexity, effort, and efficiency.

This log parser is implemented using the same *LogParser* class structure as defined in Section

4.5.2. This is done to ensure that interfaces are maintained and the output data structure is consistent across all log parsing methods to more readily facilitate analysis and comparison between generated outputs.

The regex-based log parsing algorithm is implemented in Python using the *re* module and uses a manually created set of patterns describing the log message formats to parse log files into a structured sequence of events and event templates.

4.6 Component Design: Data Miner

With reference to Section 4.2 and Figure 4.1, this section details the integration of all the subcomponents to realise the implementation of the Data Miner.

The Data Miner is implemented as a Python class named *DataMiner*. This class structure is illustrated through the UML Class Diagram in Figure 4.16.

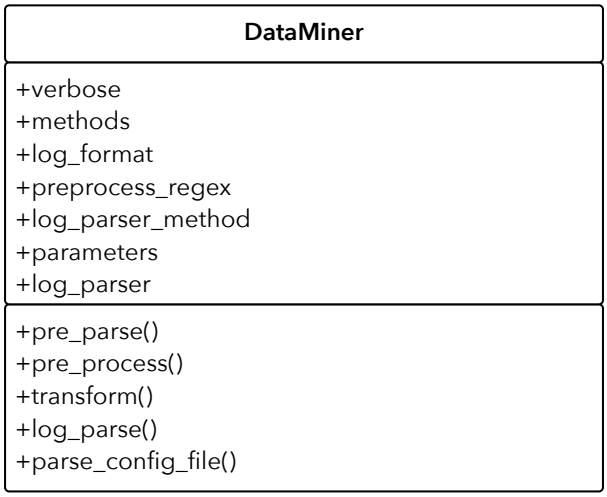


Figure 4.16: UML Class Diagram of the Data Miner class

When instantiating a Data Miner object, three arguments are required:

- *config_file* - path to the *yaml* configuration file for the Data Miner
- *input_dir* - path to the directory where the input log files are located
- *output_dir* - path to the directory where the output data structure is to be stored

As previously discussed, the Data Preparer and Pre-Processor components are implemented as Python functions. These are implemented in the *DataMiner* class as methods, namely *pre_parse*, *transform*, and *pre_process*.

The Log Parser is implemented by instantiating a *LogParser* object pertaining to a particular algorithm. This object is referenced as *log_parser* in the Data Miner. The *log_parse* method invokes the Log Parser to run the log parsing algorithm.

verbose is a flag that enables or disables parsing statistics and *methods* is a dictionary that stores string name to module mappings of the various log parsing algorithms.

Additional attributes, namely, *log_format*, *preprocess_regex*, *log_parser_method* and *parameters* store configuration information for the Data Miner parsed from the provided configuration file by the *parse_config_file* method. The format of the configuration file is shown in Figure 4.17.

Data Miner Configuration File (.yaml)

```
log_format: # format of the log messages

preprocess: # list of regex for preprocessing
   -
   -
   -

logparser: # log parsing technique & configuration
   Method:
   Parameters:
      Param1:
      Param2:
```

Figure 4.17: Format of the Data Miner Configuration File

The Data Miner configuration file is a *yaml* [82] file that defines various configuration parameters for the Data Miner. These are:

- *log_format* - a string describing the format of the log messages as required by the Pre-Parser function
- *preprocess* - a list of regular expressions for matching variable runtime components in log messages
- *logparser* - configuration of the log parsing technique to be used by the Data Miner including the method name and configurable parameters

By calling the various methods of the *DataMiner* class, the processing pipeline as illustrated in Figure 4.1, can be implemented. By modularising the various processing stages, opportunities for introspection are afforded to assess the behaviour of the Data Miner at each stage.

After parsing the log files, the resultant data is to be further analysed by the Inference Engine. Given the inherent interface that exists between the Data Miner and Inference Engine, the output of the Data Miner was designed to best accommodate the processing that would be done by the Inference Engine.

The output of the Data Miner is two data structures generated by the processing pipeline. First, a structured version of the log file in which each log message is represented by an appropriate event template and a list of parameters. And second, an event occurrence matrix that lists all the event templates describing the log file and number of times each event occurred. Both of these data structures are stored as *comma separated value* files as this allows for simple reading into a pandas *DataFrame* object, a preferred data structure for machine learning frameworks.

The reader is reminded that the Data Miner is a single component of the larger Automated Log File Analysis Framework. This framework employs the Data Miner and interfaces with the system under test. This framework is fully detailed in Chapter 6.

4.7 Parsing Algorithm Tuning

As discussed in Section 4.5.1, the various automated log parsing algorithms each have a set of configurable parameters that control various aspects of the algorithm. Research has shown that in some cases, the performance of the algorithms is sensitive to the values of the configurable parameters. As a high parsing accuracy is required for later machine learning-based analysis, finding the optimal set of configurable parameters for a given algorithm is imperative.

To aid in finding the optimal set of configurable parameters for a given algorithm, for a given set of log messages, a tuning tool was developed. This tool implements a grid search method for searching across the entire parameter space for any given log parsing algorithm. The grid search is implemented using *scikit-learn's* ParameterGrid [83] object that returns a Python iterable over the search space. For each set of parameters, a log file is parsed using the algorithm and the parsing accuracy is calculated with reference to a ground truth dataset.

The set of parameters that resulted in the highest parsing accuracy is returned. The tuning tool uses a configuration file similar to that used by the Data Miner as detailed in Section 4.6 with the only difference being that instead of single values for the parameters, a range is instead specified.

The tuning tool can also automatically generate a Data Miner-ready configuration file with the optimal set of parameters.

It should be noted that tuning of the parsing algorithms' configurable parameters is only possible when a ground truth dataset, i.e. one with 100% parsing accuracy, is available.

4.8 Data Miner Design Conclusion

This chapter has described the detailed design and development of the Data Miner component of the Automated Log File Analysis Framework (ALFAF).

A high-level design and overview of the Data Miner was provided and the design of the lower-level subcomponents, namely the *Data Preparer*, *Pre-Processor* and *Log Parser* were detailed.

The interface between the Data Miner and the Inference Engine, the next stage in the ALFAF, was considered and used to inform the structure of the output of the Data Miner.

The integration of all subcomponents to realise the Data Miner was detailed and the implementation of a configuration file-based control interface was discussed.

Finally, an auxiliary tool to assist in tuning log parsing algorithms was also developed.

The Data Miner is integrated into the ALFAF, alongside the Inference Engine, in Chapter 6. The Data Miner is tested and verified in Chapter 7, and algorithmic tuning of the log parser is also considered. The next chapter in this dissertation details the design of the Inference Engine.

5

Design and Development: Inference Engine

This chapter continues the System Design phase that was initiated in the previous chapter, and describes the design and development of the Inference Engine component of the end-to-end Automated Log File Analysis Framework (ALFAF). The process followed for the design and development of the Inference Engine is outlined in Section 2.2.1.

To guide the design and development of the Inference Engine, a set of constraints and design considerations is put forward. These are based on, and informed by, the objectives of this research project, the research questions and the literature reviewed. These design considerations are detailed in Section 5.1.

Initially, informed by the design considerations, a high-level design of the Inference Engine is detailed and described. This design is used to identify the major subcomponents of the Inference Engine, the interfaces between subcomponents and other framework components and to describe the functionality of the Inference Engine. This high-level design is presented in Chapter 5.2.

After exploring the high-level design, the detailed design of Inference Engine subcomponents is considered. There are two major subcomponents, namely the *Feature Extractor* and the *Anomaly Detection Model* and these are detailed in Sections 5.3 and 5.4 respectively.

The high-level design also identifies the major processing stages and processes of the Inference Engine. These are identified to be *Feature Extraction*, *Data Loading*, *Model Training* and *Anomaly Detection*. These processes, alongside the integration of the subcomponents to realise the complete system, are detailed in Section 5.5.

It should be noted that this chapter only details the design and development of the Inference Engine and its subcomponents. The tuning, testing, evaluation and verification of the Inference Engine is considered, but is detailed in Chapter 7.

5.1 Design Considerations and System Requirements

The Inference Engine is the other major component of the overall Automated Log File Analysis Framework. The Inference Engine has two main functions. Firstly, it is to ingest a dataset containing pre-processed and parsed log file data and to use this data to train a deep learning model capable of identifying and isolating failures within a system based on the content of the provided log files. Secondly, it is to load a previously learned model and again ingest a dataset containing pre-processed and parsed log file data and to identify possible errors and failures within the system from the provided log files.

Using the research questions put forward in Section 1.2 and the findings from literature as described in Chapter 3, key design considerations, and system requirements as applicable, are proposed to guide the design of the Inference Engine:

Inference Engine Objective The main objective of the Inference Engine is to identify and isolate failures that occur within a system from information contained in runtime-generated system logs. Given that log files describe the sequence of events executed during system runtime, the problem of identifying failures is modelled as an anomaly detection problem in which deviations from the expected system execution paths are to be flagged as anomalies. The Inference Engine shall therefore model the expected execution of a given system based on information contained in log files and shall flag deviations from this execution model as anomalies.

Input Data The Inference Engine shall be capable of processing log files that have been processed by the Data Miner as detailed in Chapter 4. The Inference Engine shall only process parsed log files that have a corresponding log key or event template for each log entry. The Inference Engine shall receive these pre-processed and parsed log files as a *comma separated values* (CSV) file.

Output Data Depending on whether the Inference Engine is being used to train a model or to perform inference, the output of the Inference Engine will vary. During Inference Engine training, the Inference Engine shall output a list of parameters describing a deep learning model trained to detect anomalies within log files of a given system. During inference, the Inference Engine shall output a dataset containing log records that have been flagged as anomalies by the Inference Engine.

Interfaces The Inference Engine shall have an interface to the Data Miner. This interface shall be described by the format of the data passed from the Data Miner to the Inference Engine. The format of this data is described during the System Integration Phase in Chapter 6. When performing inference, the Inference Engine shall have an interface to the ALFAF. This interface shall be described by the format of the data passed from the Inference Engine to the framework.

Features The Inference Engine shall extract appropriate features from the parsed log dataset to represent the desired relationships between the log records. The Inference Engine shall use these features to represent the data when training the deep learning model and when performing inference. Given the nature of log files, i.e. a sequence of events, and the unlikelihood of having a labelled system log file dataset available for model training, the sequence of log events, as

described by the extracted log keys, shall be used as the basis for feature extraction. The features extracted shall represent the expected execution path for a given system.

Deep Learning Algorithm As detailed in Section 3.2.4 research toward automating log file analysis has shown promise when employing various machine learning or deep learning algorithms. Considering the time series, sequenced nature of log files, Recurrent Neural Networks, adept at sequence modelling and retaining information across sequences, are particularly promising. The Inference Engine shall employ a Recurrent Neural Network architecture for performing log file analysis. The Recurrent Neural Network shall be used to model the expected system execution path which will be used to detect anomalies in the form of deviations from this expected path. The Recurrent Neural Network shall ingest a sequence of log events and shall predict the next likely log event. The prediction problem shall be modelled as a multi-class classification problem, in which every possible log key event, in the given system, is a possible class.

Deep Learning Framework The Inference Engine shall make use of the *PyTorch* Deep Learning Framework [74] for implementing machine learning and deep learning models. *PyTorch* facilitates rapid prototyping and can readily leverage additional compute capability, e.g. GPUs, if and when available.

Usage The Inference Engine shall be a modular, stand-alone tool that may be run independently of the ALFAF. This will facilitate tuning and testing of the anomaly detection algorithm independently.

Processing Mode Given that the aim of this research project is to demonstrate a proof-of-concept framework for automated log file analysis, the Inference Engine shall only be designed to perform offline log file analysis and anomaly detection on pre-generated log files and shall not consider the processing of log files in real-time.

Performance Metrics The Inference Engine shall output various performance metrics during both the training and inference/prediction modes of operations. These metrics are considered during the design but are further detailed in Chapter 7.

Subsequent sections in this chapter, informed by the aforementioned design considerations and guiding system requirements, outline the design and development of the Inference Engine.

5.2 High-Level Design: Inference Engine

This section presents the high-level design of the Inference Engine. This design is derived by considering the processing path of the incoming parsed log file data through the Inference Engine and by taking into account the major objectives and design goals of the Inference Engine. This high-level design is used to identify the major subcomponents and processes of the Inference Engine, to completely describe the functional behaviour and to outline the complete end-to-end design of the Inference Engine.

The main objective of the Inference Engine is two-fold. Firstly, the Inference Engine shall ingest a parsed log file and train an anomaly detection algorithm to identify failures and errors in a

system from these log files. Secondly, the Inference Engine shall again ingest a parsed log file, and using a previously trained model, shall identify, detect and flag potential erroneous events within the provided log files.

In order for the Inference Engine to perform both objectives, two modes of operation are defined. A *Training Mode* in which the Inference Engine trains an anomaly detection model and an *Anomaly Detection Mode* in which the Inference Engine loads a previously trained model and detects anomalies within system log files.

With these design considerations and the main objectives in mind, the high-level design of the Inference Engine is illustrated in Figure 5.1.

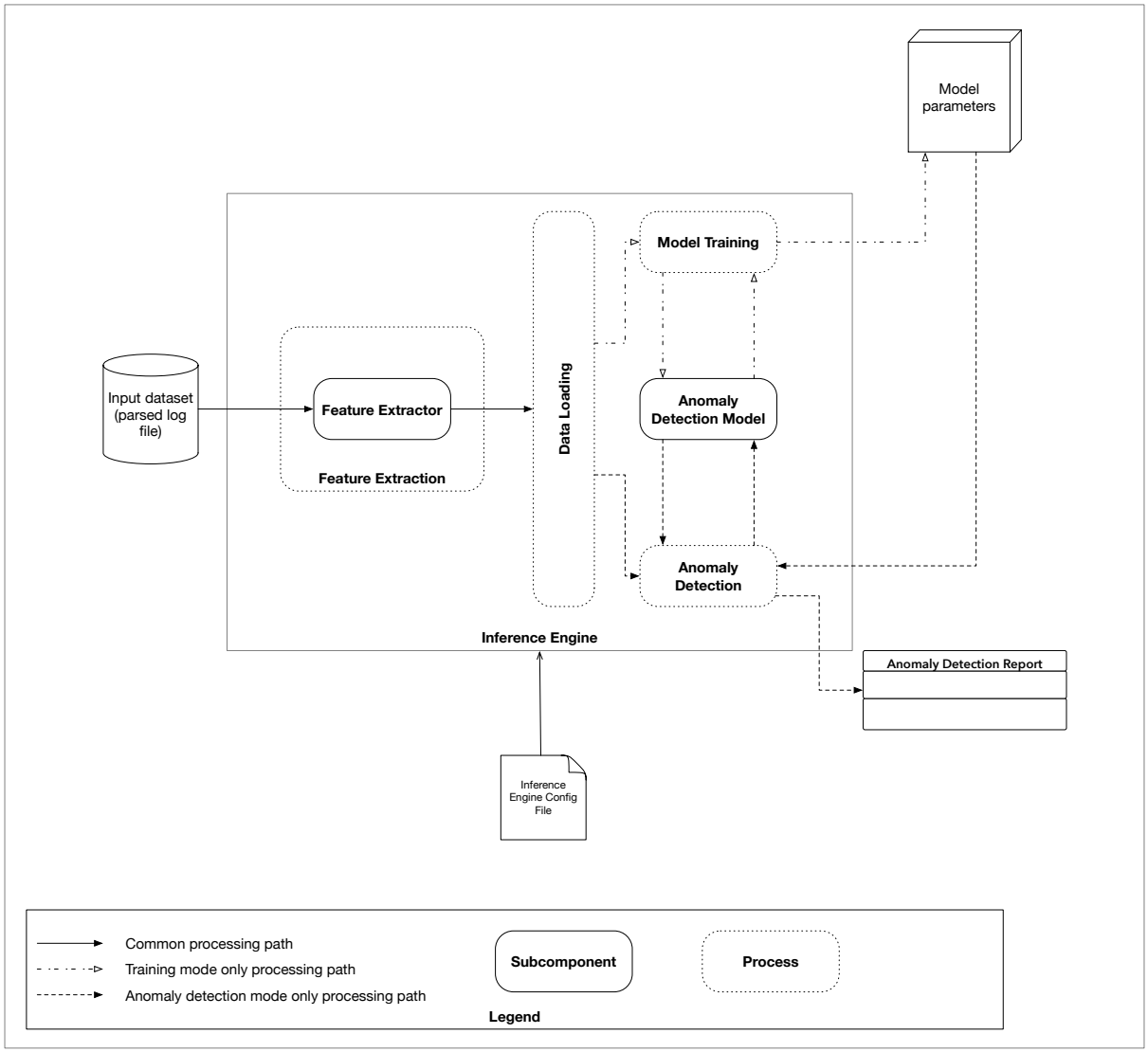


Figure 5.1: High-Level Design of the Inference Engine component of the Automated Log File Analysis Framework. The major subcomponents of the Inference Engine, the *Feature Extractor* and *Anomaly Detection Model*, are illustrated alongside the inputs to and outputs from the Inference Engine. Also shown are the different data path flows for model training and for model inference

As shown in Figure 5.1, the Inference Engine has two data processing pipelines, one corresponding to each mode of operation. The Inference Engine consists of two main subcomponents, the *Feature Extractor* and the *Anomaly Detection Model*, as well as a number of processes. Both subcomponents are used during both operational modes but are configured differently. The processes used vary depending on the mode of operation.

There are four major processing stages that are performed by the Inference Engine. These are *Feature Extraction*, *Data Loading*, *Model Training* and *Anomaly Detection*. These are illustrated in Figure 5.1.

The first processing stage, *Feature Extraction*, is responsible for ingesting the parsed log file and extracting useful, representative features that can aid in the objective of anomaly detection using a deep learning algorithm. Both modes of operation require *Feature Extraction* but the way in which the final data representation is derived in either case differs. The *Feature Extractor* subcomponent implements the *Feature Extraction* process and is detailed further in Section 5.3.

After *Feature Extraction*, the next processing stage in the pipeline is *Data Loading*. During this stage, the data, a feature-rich dataset representing the parsed log files, is loaded into the correct data structures and format in preparation for feeding into a deep learning model. During this process, the data is also copied to memory on the applicable compute device that will be used for model training or inference. Both modes of operation require this process. The *Data Loading* process is detailed in Section 5.5.

Once the *Data Loading* process is complete, the processing pipeline diverges depending on the mode of operation. In *Training Mode*, after loading the data, the Inference Engine initiates the *Model Training* processing stage. During this processing stage, the *Anomaly Detection Model* is trained to model relationships and dependencies among the input features in the data to detect and flag anomalous log events within the log files. This processing stage produces a trained model described by a list of model parameters that can be loaded to use the model for inference at a later stage.

In *Anomaly Detection Mode*, after the data is loaded, the Inference Engine executes the *Anomaly Detection* process. During this processing stage, the Inference Engine attempts to detect and identify anomalies within the provided log file. This processing stage uses the *Anomaly Detection Model* subcomponent, but instead of training a new model, model parameters learned during *Model Training* are loaded. The output of this processing stage is a report identifying potential anomalous log events.

The *Model Training* and *Anomaly Detection* processes are detailed in Section 5.5.

The *Anomaly Detection Model*, utilized by the *Model Training* and *Anomaly Detection* processes is a subcomponent that implements the deep learning model that is to be trained to perform anomaly detection. The design of this model is discussed in Section 5.4.

As shown in Figure 5.1, there are three main inputs to the Inference Engine. The first is a dataset containing parsed log files as generated by the Data Miner detailed in Chapter 4. The format of this dataset is discussed and detailed in Chapter 6. The second input is a configuration

file that contains configuration information for the various subcomponents and processes of the Inference Engine. The details of the configuration file is described in Section 5.5. Lastly, when used in *Anomaly Detection Mode*, a set of parameters describing a previously trained deep learning model is required to instantiate a version of the model capable of performing anomaly detection.

The main outputs of the Inference Engine are the model parameters when used in *Training Mode* and a report detailing detected anomalies when used in *Anomaly Detection Mode*.

The integration of all subcomponents, and the implementation of all processes and sub-processes making up the Inference Engine, is detailed in Section 5.5.

5.3 Subcomponent Design: Feature Extractor

As discussed in Section 5.2, the *Feature Extractor* subcomponent of the Inference Engine performs the *Feature Extraction* process. During this process, features that are useful and representative of what is to be modelled by a deep learning algorithm are extracted from the given dataset. These features are stored in a dataset that is ready to be ingested by a deep learning model.

Research reviewed in Chapter 3 showed that the types of features extracted are dependent on the nature of problem, what is to be modelled and the deep learning model or algorithm that is to be used. Log files have been seen to contain numerous pieces of information about the system that generated them. One such piece of information is the system's execution path that is embedded in the sequence of log messages. For this project, the problem of system failure detection is modelled as anomaly detection in which the system's execution path is analysed to detect anomalous, or unexpected events, which likely indicates a failure within system operation. The system's execution path can be modelled by considering the sequence of log events. Since one of the objectives of the Data Miner is to reduce a sequence of unstructured log events into a sequence of log event keys, this sequence is representative of the system's execution path. Using log files that are generated during normal or expected system operation, it is possible to use this sequence of log event keys to create a dataset of features that can be used to model the expected system behaviour.

The expected system execution behaviour is modelled by constructing features in the following way: a sequence of w log event keys is considered as input features, and the log event key appearing after the selected sequence, i.e. at index $w+1$, is considered the target. The deep learning model can use these features and targets to learn what the expected next log event key is for a given sequence of keys and in doing so, can learn and model the expected system behaviour. Extracting features in this way eliminates the need for a labelled dataset which is desired since this is often unavailable in the context of log files. These features are also generalisable across log files generated by different systems as they rely on one of the inherent properties of log files i.e. a sequence of events. During anomaly detection, features are extracted in a similar way, however the manner in which they are used differs slightly. The extracted input features are used as inputs to the deep learning model and the model then predicts what the

next expected log event key is. This is then compared to the actual target key, extracted from the log file, and if they are different, an anomaly is flagged. This process is detailed further in Section 5.5.

The *Feature Extractor* subcomponent implements the necessary processes to transform an incoming parsed log file into a feature-rich dataset consisting of a window of size w of log event keys and a corresponding target log event key. The processes performed by the *Feature Extractor* are shown in Figure 5.2.

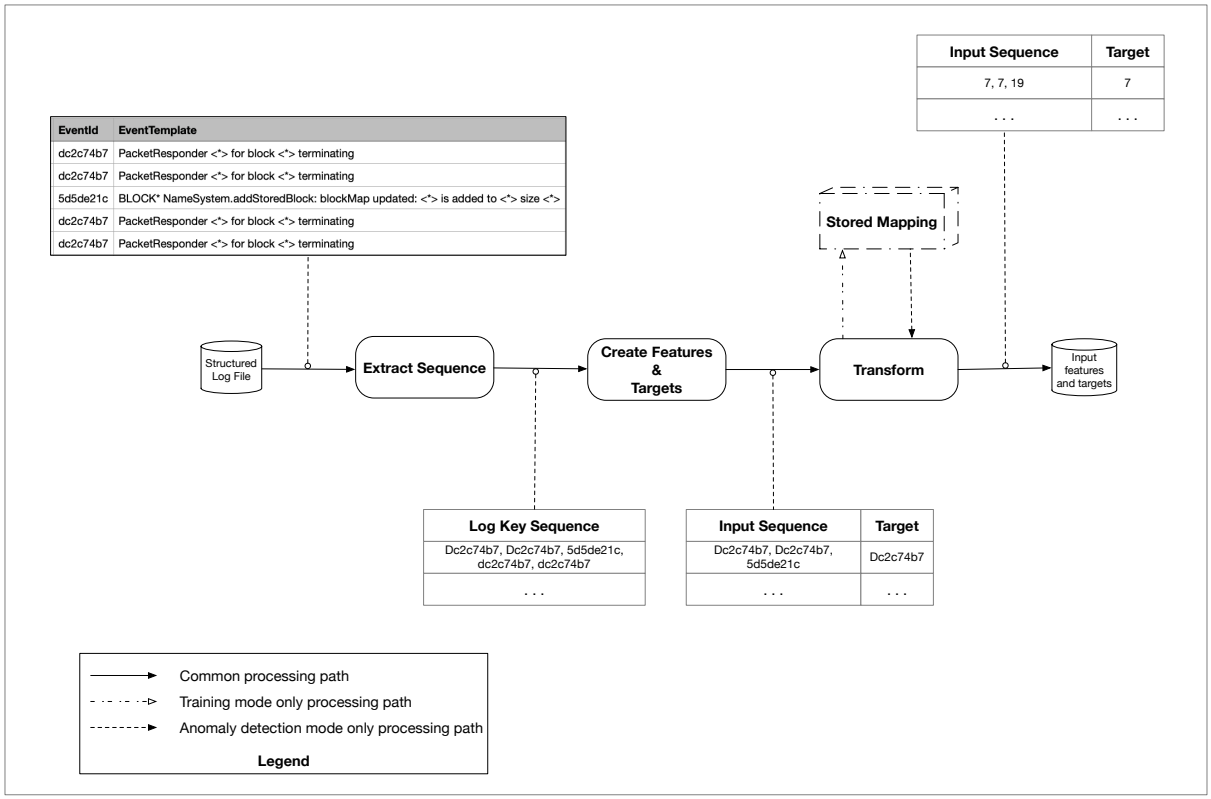


Figure 5.2: Processes of the *Feature Extractor* subcomponent. Also illustrated is how the data is transformed between processes

As shown in Figure 5.2, the *Feature Extractor* consists of three main processes: *Extract Sequence*, *Create Features and Targets*, and *Transform*.

The *Extract Sequence* process, implemented as a Python method, extracts a sequence of log event keys from the provided parsed log file. When received, the parsed log file consists of a structured dataset in which each row entry corresponds to a log message. Each log message is represented by multiple data fields, including a log event key. The *Extract Sequence* process extracts the log keys from all messages, while ensuring that the sequence in which the log keys appear is preserved, to generate a sequence of log keys that is representative of the system's execution path that was recorded by the original log file. Some systems, particularly those that have multiple concurrent processes running, generate log files that contain log messages from multiple processes or sessions. In systems where this is known to be the case, the *Extract Sequence* process is able to group sequences by session, provided that a session identifier is provided. The output of the *Extract Sequence* process is a sequence of log event keys as shown

in Figure 5.2.

Once a sequence of log event keys representing the system's execution path has been extracted, the *Create Features and Targets* process applies a sliding window to the sequence of log keys to extract pairs of input sequences and target log event keys. This process is implemented as a Python method and the size of the window, or the number of log event keys to be used in the input sequence, is tunable. The sliding window is applied to the sequence of log event keys until the entire sequence has been considered. In the event that there are insufficient log event keys remaining to satisfy the window size, then no further windows may be created. In instances where log key sequences have been grouped by session, the sliding window is only applied within each session group, and not across sessions. The output of the *Create Features and Targets* process is a dataset consisting of a number of input sequence - target key pairs generated across the entire log file.

Once the input features and labels have been created, the final process of the *Feature Extractor* transforms the data such that it can be fed into a deep learning model. At this stage of the processing pipeline, both the input sequence and targets are alphanumeric text strings. Deep learning algorithms perform better on numerical data and as such, the features need to be transformed. The *Transform* process performs this transformation. When a new model is being trained, the *Transform* process learns a new transformation for the data. This transformation is in the form of a mapping in which each unique log key, in the entire log file dataset, is mapped to a unique numeric integer value. This mapping is stored in a *yaml* file such that the same mapping used for model training may be used when the model is performing inference. During inference, the *Transform* process loads a previously stored model corresponding to the log files that are being processed. The *Transform* process is implemented as a Python method.

The output of the *Transform* process, and the final output of the *Feature Extractor*, is a transformed dataset consisting of input sequence - target key pairs. This dataset is ready to be ingested by a deep learning model for either model training or inference.

The entire *Feature Extractor* subcomponent is implemented as a Python class, named *FeatureExtractor* and its UML Class Diagram is shown in Figure 5.3.

An instance of *FeatureExtractor* is instantiated with the following parameters which are stored as class attributes:

1. *sample_by_session* - a flag specifying whether the log messages in the log file are to be sampled by session
2. *window_size* - an integer specifying the size of the sliding window i.e. how many log event keys to include per window
3. *training_mode* - a flag specifying whether the Feature Extractor is to be run in training mode or not
4. *data_transformation* - a path to a previously generated data transformation, required when running in Anomaly Detection mode

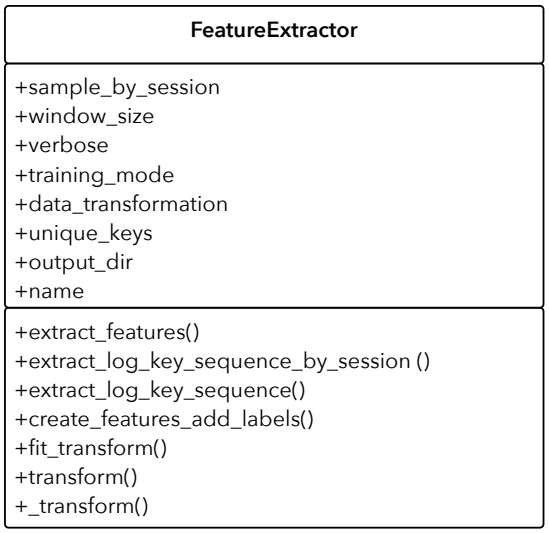


Figure 5.3: UML Class Diagram of the *FeatureExtractor* class.

5. *verbose* - a flag specifying whether the Feature Extractor will output information regarding its operation
6. *output_dir* - a string specifying a path to a directory where all feature extractor outputs are to be stored
7. *name* - a unique name for identifying a particular instance of the Feature Extractor

An additional class attribute, *unique_keys*, is used to store the set of unique log event keys present in the log dataset that is to undergo Feature Extraction. The methods of the class implement the various processes as previously described.

5.4 Subcomponent Design: Anomaly Detection Model

The *Anomaly Detection Model* subcomponent implements the deep learning model that will be trained to model system behaviour from provided system-generated log files in order to be able to detect and flag errors and failures that occurred during system execution. As discussed in Chapter 3, the most promising results with regards to using deep learning for Log File Analysis were found when using Long Short-Term Memory Recurrent Neural Networks (LSTMs). Based on the results in recent literature, a similar deep learning model, LSTM-based, is used for the Inference Engine of this project.

The architecture of the deep learning model implemented by the *Anomaly Detection Model* is shown in Figure 5.4.

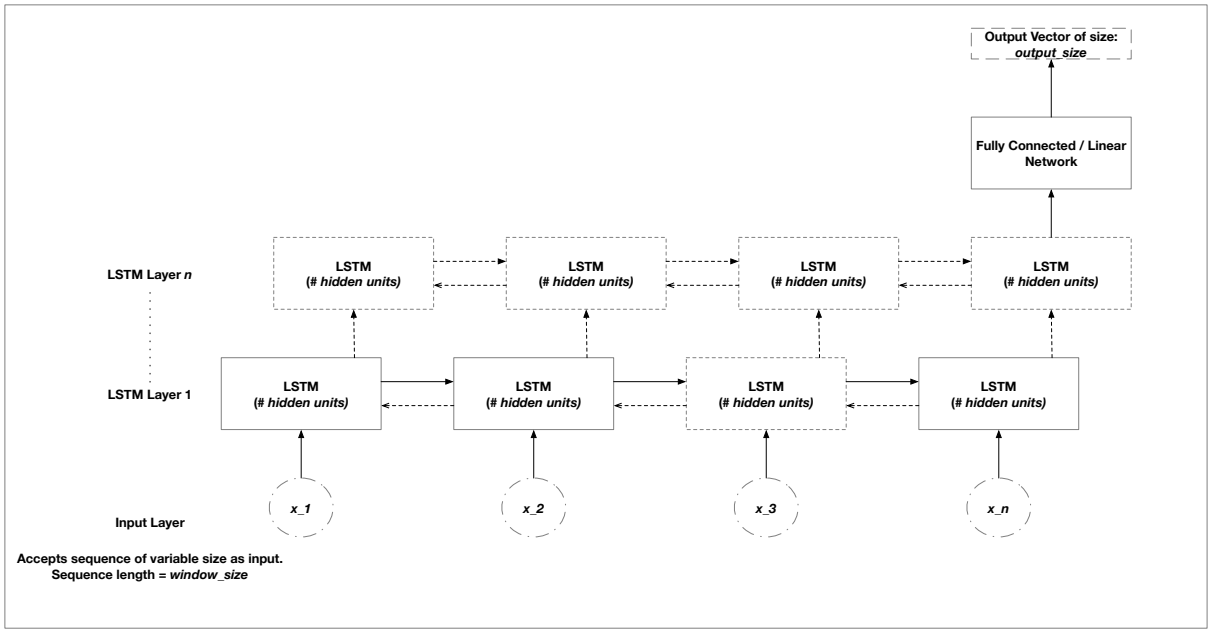


Figure 5.4: Architecture of the Long Short-Term Memory Recurrent Neural Network implemented by the *Anomaly Detection Model*. Configurable hyperparameters may be used to change the model architecture

Figure 5.4 shows a generic LSTM neural network architecture, rolled out in time, that consists of multiple LSTM-layers and that is bidirectional. Research into the use of LSTM neural networks for the problem of log file analysis in the form of anomaly detection has illustrated various LSTM architectures in terms of the number of LSTM layers and whether or not the LSTM is bidirectional. The number of hidden units in the LSTM layers also varies. These various architectures result in varying degrees of performance across various datasets. Since log files generated by different systems can differ substantially in structure and content, proposing a single, defined LSTM architecture for the Anomaly Detection Model would be inefficient and would lead to less than favourable performance in most cases. For the best results, a unique LSTM architecture is to be modelled and trained for each unique dataset. To facilitate this, the following hyperparameters, also shown in Figure 5.4, are tunable for the implementation of the Anomaly Detection Model:

1. *input_size* - the number of features describing each input. For the features extracted by the Feature Extractor, each input is only described by a single feature i.e. the log event key
2. *hidden_size* - the number of hidden units the LSTM layer of the network should have
3. *output_size* - the size of the output of the overall model. For the Inference Engine, this is equal to the number of unique log event keys for a given system
4. *num_layers* - the number of LSTM layers that may be implemented in the network architecture

5. *bidirectional* - flag indicating whether a bidirectional LSTM should be implemented or not

These hyperparameters are provided to the Anomaly Detection Model and are used to define the architecture of the LSTM model that is to be implemented. When the Inference Engine is to be used on a new system, these hyperparameters must be tuned and selected to achieve optimal performance for that particular system. The impact of these hyperparameters on Inference Engine performance, and their tuning, shall be investigated and detailed in Chapter 7.

It should be noted that while the architecture of the LSTM model may vary, design decisions are made regarding the structure of the input and output layers of the network. As shown in Figure 5.4, the LSTM network is designed to accept an input sequence of log event keys and the size of this sequence may vary. Each element in the sequence, corresponding to a single log key, is fed into the LSTM network sequentially. Since the problem is being modelled as a multi-class classification task, only the final output of the LSTM network, after the entire input sequence has been considered, is of interest. This final output, of the LSTM layers, is then fed into a Fully Connected Linear Neural Network Layer to covert the output into a vector containing probability scores for each of the classes. Each index in this vector represents one of the classes and the score represents the likelihood of that class being the target class for the given input sequence.

The *Anomaly Detection Model* is implemented as a Python class, named *AnomalyDetectorLSTM*, using the PyTorch Framework. The UML Class Diagram for this class is shown in Figure 5.5.

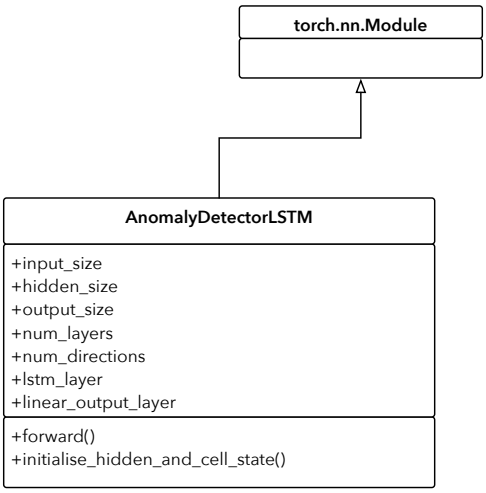


Figure 5.5: UML Class Diagram for the *AnomalyDetectorLSTM* class that implements the Anomaly Detection Model

The *AnomalyDetectorLSTM* class inherits from PyTorch's *nn.Module* class which is a base class for PyTorch Neural Netorks. When an instance of *AnomalyDetectorLSTM* is initialised, it is provided with the hyperparameters, as previously discussed, as parameters. During initialisation, the LSTM and Linear layers are created, with the provided hyperparameters, using the *nn.LSTM* and *nn.Linear* classes of PyTorch. Two methods are defined for the *AnomalyDetectorLSTM* class:

1. *forward* - this implements the forward pass function in which the input data is propagated in a forward direction through the layers of the network model to predict the output values
2. *initialise_hidden_and_cell_states* - this initialises the hidden and cell states of the LSTM layers. This is performed for every new input sequence that is fed into the model

The use of PyTorch enables the model to be run, for both training and inference, on either CPU or GPU hardware if available, without any changes to the model implementation. After initialising an instance of the *AnomalyDectorLSTM* class to implement the Anomaly Detection Model, the *forward* method is used to run the model on a given set of input data to generate the desired output.

5.5 Component Design: Inference Engine

The complete Inference Engine is illustrated in Figure 5.1 and the high-level design is described in Section 5.2. This section describes the integration of the subcomponents and the design and implementation of the various processes that make up the Inference Engine. Where applicable, it also details how the processes facilitate the two modes of operation of the Inference Engine - *Training Mode* and *Anomaly Detection Mode*.

The Inference Engine is implemented as a Python class named *InferenceEngine*. This class is illustrated in the UML Class Diagram in Figure 5.6.

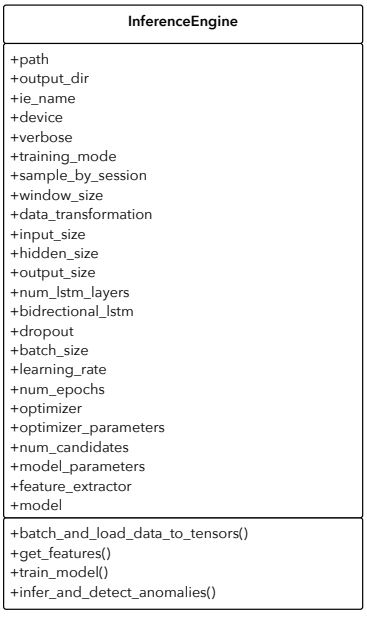


Figure 5.6: UML Class Diagram of the *InferenceEngine* class

Instantiating an instance of the *InferenceEngine* class requires four arguments:

1. *config_file* - a path to the configuration file for the Inference Engine
2. *name* - a unique identifier for this instance of the Inference Engine
3. *device* - a string specifying which device the Inference Engine will run the deep learning

model on

4. *verbose* - a flag that enables detailed output and statistics from the Inference Engine to be printed to the terminal
5. *output_dir* - a path to a directory where all Inference Engine-generated outputs are to be stored

Most of the attributes of the *InferenceEngine* class are parameters for the Feature Extractor and Anomaly Detection Model subcomponents. The values of these attributes are parsed from the configuration file provided to the Inference Engine. The configuration file is a *yaml* file and defines configuration parameters for the subcomponents and processes of the Inference Engine. These parameters are described in subsequent sections.

The *sample_by_session*, *window_size* and *data_transformation* attributes are used to configure the Feature Extractor. The *input_size*, *hidden_size*, *output_size*, *num_lstm_layers*, *bidirectional_lstm* and *dropout* attributes are used to configure the Anomaly Detection model. The configuration of these subcomponents, and how these parameters are used, is detailed in Sections 5.3 and 5.4.

The *path* attribute is used to create a unique working directory for each instance of the Inference Engine and *ie_name* is used to create a unique prefix for all files generated by this instance of the Inference Engine. The *device* attribute specifies which device is to be used for model training and model inference. This is used by PyTorch and enables models to be run on either CPU or GPU hardware. *verbose* is a flag that enables detailed output during the operation of the Inference Engine.

The *training_mode* attribute is set depending on which mode of operation is selected. This attribute is used to control the processing pipeline through the Inference Engine depending on whether the Inference Engine is to train a new model or whether it is to use an existing model to perform inference.

Depending on which mode of operation is used, a unique set of attributes are defined to facilitate each mode. These are detailed in Subsections 5.5.3 and 5.5.4.

As shown in Figure 5.1, the Inference Engine consists of two major subcomponents - the Feature Extractor and the Anomaly Detection Model. These are implemented by the *FeatureExtractor* and *AnomalyDetectorLSTM* as described in Sections 5.3 and 5.4 respectively. The Inference Engine implements these subcomponents by instantiating instances of each class. The *feature_extractor* attribute stores an instance of the *FeatureExtractor*, configured with the parameters parsed from the configuration file. The *model* attribute stores and instance of the *AnomalyDetectorLSTM* that instantiates an LSTM-model configured with the model parameters defined in the configuration file.

These instances of the *FeatureExtractor* and *AnomalyDetectorLSTM* are used by the various processing stages of the Inference Engine. The processing stages of the Inference Engine are implemented as methods of the *InferenceEngine* class. By calling and invoking the various

methods, the different processing pipelines of the inference Engine, as illustrated in Figure 5.1 can be executed.

The following subsections describe the main processing stages of the Inference Engine. These are Feature Extraction, Data Loading, Model Training and Anomaly Detection, and are detailed in Subsections 5.5.1, 5.5.2, 5.5.3 and 5.5.4 respectively. These processing stages were also illustrated in Figure 5.1.

5.5.1 Feature Extraction

The Feature Extraction process is implemented by the *get_features* method of the *InferenceEngine* class. This method uses the previously instantiated Feature Extractor to extract features from an input dataset consisting of structured, parsed log files. This method returns a feature-rich dataset consisting of log event key input sequences and target log event keys.

5.5.2 Data Loading

The Data Loading process varies depending on the Inference Engine's mode of operation and is implemented by the *batch_and_load_data_to_tensors* method for *Training Mode* and is incorporated into the *infer_and_detect_anomalies* method for *Anomaly Detection Mode*. In either case, the output of the Feature Extraction process is provided as input to the Data Loading process.

In *Training Mode*, the Data Loading process extracts the log event key sequences and the target log event keys from the dataset and loads them into PyTorch *tensors* [84]. These are multi-dimensional arrays that can be stored and used in operations on either CPU or GPU hardware. After loading the data into tensors, the input dataset, to be used for training a model, is split into a number of batches, governed by the *batch_size* attribute. The training data is split into batches to reduce memory usage and to speed up model training. The batched training dataset is then stored in an iterable object that is returned by the *batch_and_load_data_to_tensor* method. This iterable object is provided as input to the Model Training processing stage.

In *Anomaly Detection Mode*, the Data Loading process is simplified and is implemented by the *infer_and_detect_anomalies* method. The feature-extracted dataset is provided as input to this method and for each input sample in the dataset, the log event key sequence and the target log event key are extracted. These are then loaded into tensors in preparation for the rest of the Anomaly Detection process. In *Anomaly Detection Mode*, each input sample is retrieved individually and all samples are fed to the Anomaly Detection process sequentially. Batch processing of the input samples is not performed in *Anomaly Detection Mode*.

5.5.3 Model Training

The Model Training process is only executed when the Inference Engine is run in *Training Mode*. This process is implemented by the *train_model* method and is responsible for training the Anomaly Detection Model using the provided input training data that has been loaded into an iterable object by the Data Loading process.

The Model Training process is implemented as a Python method and is illustrated through the UML Activity Diagram shown in Figure 5.7.

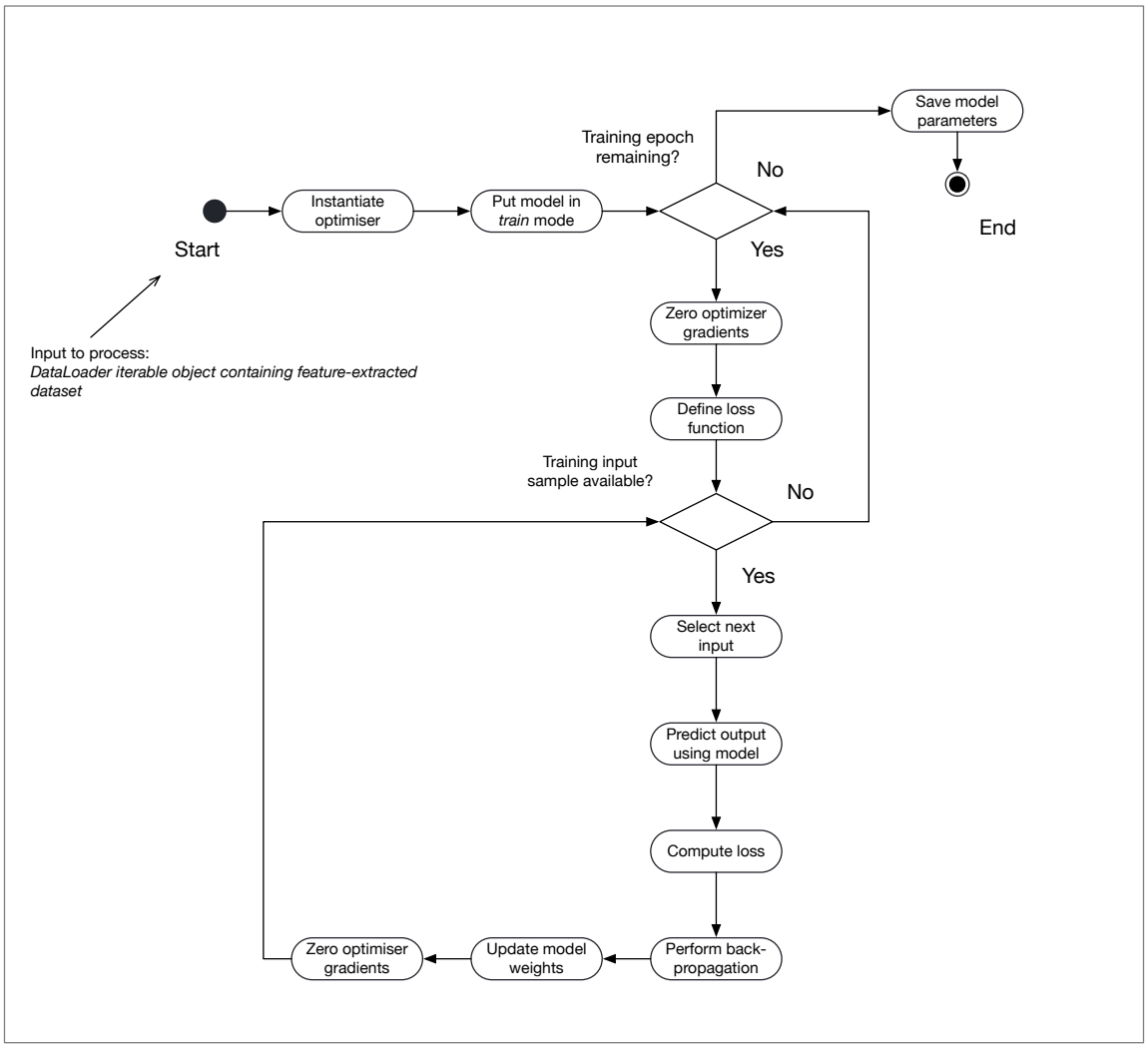


Figure 5.7: UML Activity Diagram illustrating the functionality of the Model Training process

As seen in Figure 5.7, the input to the Model Training process is the training data, encoded in PyTorch tensors and loaded into an iterable object. The Model Training process begins by instantiating an optimizer with the strategy to use to update the model parameters through the gradient decent process. Then, the previously instantiated Anomaly Detection Model is configured to be in *train* mode to enable the parameters of the neural network to be learned. The Anomaly Detection Model is trained on the entire training dataset for a given number of epochs. During each training epoch, the optimizer's gradients are zeroed at the start to avoid accumulation with previously computed gradients and a loss function is also defined. For the Inference Engine, the Cross Entropy Loss function is used as the anomaly detection problem is framed as a multi-class classification problem in which the next log key in the sequence is the classification target [85].

After the initialisation for each epoch, the training process is run for each batch of training samples in the training dataset. For each batch of input sequences, the next log key is predicted

using the Anomaly Detection Model. The actual target log keys and the predicted log keys are then used to compute the loss using the loss function. Back-propagation is then performed to propagate the loss backwards through the model to compute the gradients of the various model parameters. Once all the gradients have been computed, the optimizer is used to update the model weights accordingly. After updating the weights, the optimizer's gradients are zeroed again.

This training process continues until all batches in the dataset have been processed and until the model has been trained for the specified number of epochs. Once Model Training has been completed, the learned model parameters are saved and stored in a file. These model parameters are used to load a previously learned model during the Anomaly Detection process.

During Model Training, additional class attributes are required to control the training process:

1. *learning_rate* - this factor is used to control the rate at which the model parameters are updated using the computed gradients
2. *num_epochs* - specifies the number of epochs for which to train. During each training epoch, the model is trained on the entire training dataset
3. *optimizer* - specifies which optimization strategy to use to update the weights of the neural network after the gradients have been computed
4. *optimizer_parameters* - specifies the configurable parameters for the optimizer

5.5.4 Anomaly Detection

The Anomaly Detection process is executed when the Inference Engine is run in *Anomaly Detection Mode*. This process is implemented by the *infer_and_detect_anomalies* method and loads previously learned parameters into an instance of the Anomaly Detection Model before then using this model to detect anomalies within a given log file that is provided as a parsed, feature-extracted dataset.

The Anomaly Detection process is implemented as a Python method and is illustrated in the UML Activity Diagram shown in Figure 5.8.

The *Anomaly Detection* process requires two inputs. Firstly, a set of learned parameters describing the Anomaly Detection Model as generated by the Model Training process, and secondly, an input dataset corresponding to a parsed, feature-extracted log file.

As shown in Figure 5.8, the *Anomaly Detection* process begins by loading the learned model parameters into the instantiated Anomaly Detection Model and then setting the Anomaly Detection Model to operate in *evaluation* mode to prevent learning and updating of model parameters. As previously mentioned, in *Anomaly Detection Mode* the input data is not batched, and instead, each input sample is provided to the Anomaly Detection Model individually and the dataset of all samples is passed to the model sequentially.

Each input sample in the dataset consists of a sequence of log event keys, the input, and the actual next log event key, following the sequence. Both are derived and extracted from the

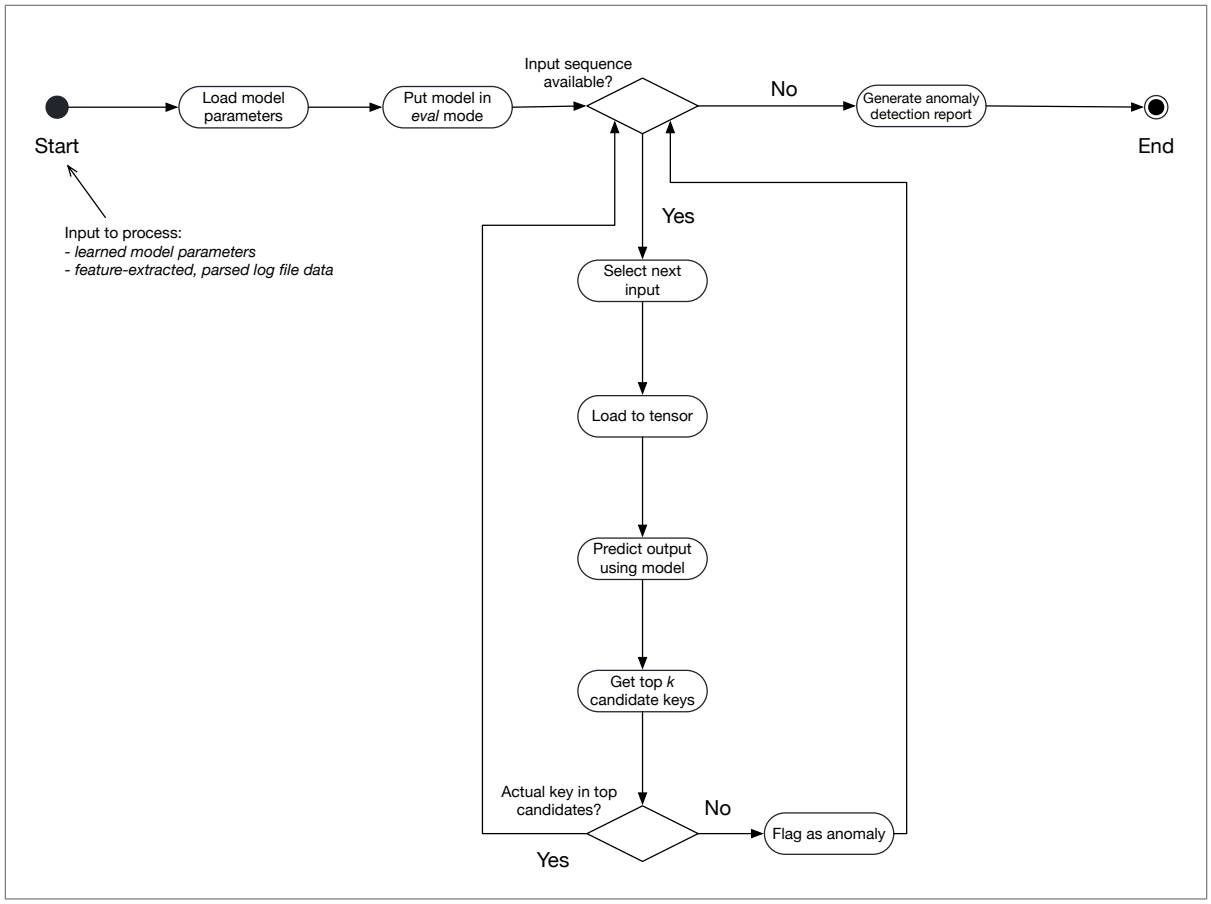


Figure 5.8: UML Activity Diagram illustrating the functionality of the Anomaly Detection process

given log file. For each input sample, the sequence of log event keys is loaded to a tensor and is then provided as input to the Anomaly Detection Model. The model then performs inference and generates an output vector containing scores for each of the possible classes. The classes represent the complete set of possible log event keys in the given dataset, corresponding to the set of all unique log event keys for a system, and the scores indicate the likelihood of each class or log event key being the next one in the sequence after the input sequence. From the vector of scores, a number of top candidate classes, based on their scores, is selected. To detect whether an anomaly is present, the *Anomaly Detection* process checks whether the actual log event key is contained within the set of top candidate log event keys. A set of candidate keys is used in the anomaly decision making process as there may be multiple different valid log event keys for a given input sequence.

If the actual log event key is present in the set of candidate keys, no anomaly is detected and the *Anomaly Detection* process runs again for the next input sample. If the actual log event key is not present in the set, an anomaly is detected, signifying a possible error or failure in the system's operation. In this case an anomaly is flagged with the input sequence, actual log event key and candidate log keys being recorded. After flagging the anomaly, the *Anomaly Detection* process continues for the next input sample until all input samples have been processed.

Once all input samples have been processed, an Anomaly Detection Report is generated. This

report consists of all input log event key sequences, the actual targets, the expected candidate targets and whether or not an anomaly was detected by the model. This report is later used by the Automated Log File Analysis Framework to provide assistive debugging information to the user. This process is detailed in Chapter 6.

For the *Anomaly Detection* process, the following additional class attributes are required:

1. *num_candidates* - this specifies the number of top candidate log event keys to consider during anomaly detection
2. *model_parameters* - this contains a path to a file containing previously learned model parameters

5.6 Inference Engine Design Conclusion

This chapter described the detailed design and development of the Inference Engine component of the Automated Log File Analysis Framework (ALFAF).

The high-level design of the Inference Engine was described and the subcomponents, namely the *Feature Extractor* and *Anomaly Detection Model*, were designed in detail to achieve the objectives of the Inference Engine as presented at the start of this chapter.

The various processing stages and processing pipelines of the Inference Engine, depending on the mode of operation, were also detailed to describe how the Inference Engine is realised and how it performs its two major functions of model training and anomaly detection through inference.

Interfaces to the ALFAF, and to other components of the framework, were considered, but are further detailed in Chapter 6.

The Inference Engine is integrated into the ALFAF in Chapter 6, and is tuned, tested and evaluated in Chapter 7. The next chapter of this dissertation describes the integration of the various components of the ALFAF and details the design and development of the complete framework.

6

System Integration and Framework Design

This chapter finalises the design and development of the Automated Log File Analysis Framework (ALFAF). It details the integration of the previously designed Data Miner and Inference Engine subcomponents, alongside the design of the overall ALFAF that implements the end-to-end automated log file analysis processing pipeline.

The design considerations and system requirements put forward for the development of the subcomponents are applicable to the ALFAF. In addition, framework-specific considerations and requirements that guide system integration and the development of the complete framework at the top level are also put forward. These are once again based on the objectives of this project and are detailed in Section 6.1.

Before detailing the subcomponent integration and design of the ALFAF, attention is directed toward the system interfaces in Section 6.2. In this section, the various internal and external interfaces of the system are identified and detailed.

Finally, in Section 6.3, the design and development of the complete ALFAF is detailed. This section describes the integration of the previously designed subcomponents and the design of additional processes required to realise the overall ALFAF.

6.1 Design Considerations and System Requirements

The Automated Log File Analysis Framework (ALFAF) consists of the two previously designed subcomponents, the *Data Miner* and the *Inference Engine*, as presented in Chapter 1. During the design and development of these subcomponents, design considerations and system requirements were taken into account to ensure that these subcomponents are designed to enable the ALFAF to perform its intended functions. These considerations are discussed in Chapters 4 and 5. At the top level, i.e. the ALFAF itself, additional design considerations and requirements are put forward to account for the additional functionality required at the framework level as well as the integration of the subcomponents. These design considerations are based on the

research questions put forward in Chapter 1 and the project objectives. They are used to guide the integration of the subcomponents and the design of the ALFAF and are presented below:

Automated Log File Analysis Framework Objective The main objective of the ALFAF is to provide an end-to-end processing pipeline that implements automated log file analysis to identify possible points of system failure and errors from given system log files. The framework shall perform the following processing stages required for automated, machine learning-based log file analysis as discussed in Chapter 3: *Log Parsing*, *Feature Engineering*, and *Log File Analysis*.

Input The ALFAF shall receive raw, system-generated log files as input. These log files shall contain log messages stored as raw text.

Output The ALFAF shall make the following outputs available after performing automated log file analysis:

- *Debugging Report* - a comma separated values (CSV) file containing the entire, original log file in a structured format with suspicious or potential anomalous log messages flagged. This report will also indicate *why* a given log message is flagged as an anomaly. This report may be used by system operators or developers to understand what may have gone wrong during system operation.
- *Suspicious Lines Report* - a CSV file containing only the suspicious or potential anomalous log messages. This report may be used by system operators to quickly identify anomalous lines and cross-reference these with the *Debugging Report* to speed up analysis.

Operational Modes The ALFAF shall have two modes of operation. A training mode in which the framework is trained on log files generated by a given system, and an inference mode in which the framework is used to detect failures and errors within new log files generated by the same system.

Data Stores The ALFAF is designed to be robust and able to perform log file analysis on the log files from any given system. However, the framework does need to be trained on a particular system before being able to be effective. Part of this training involves creating two data stores. The first contains a history of log event key to log event template mappings. This data store is generated during framework training and is used to ensure that all subsequent log files, of the same system, are parsed and mapped using a consistent log event key to log event template mapping. The second data store contains the parameters describing a previously trained deep learning-based Anomaly Detection Model that was trained to detect anomalous log events from log files for the system under test.

Interfaces The ALFAF has an internal interface between the Data Miner and Inference Engine subcomponents. It shall also have external interfaces to the system under test and the end-user i.e. operator or developer.

6.2 System Interfaces

This section identifies and describes the major internal and external interfaces of the ALFAF. The ALFAF, and its main subcomponents, is shown in the context of its operational environment in Figure 6.1.

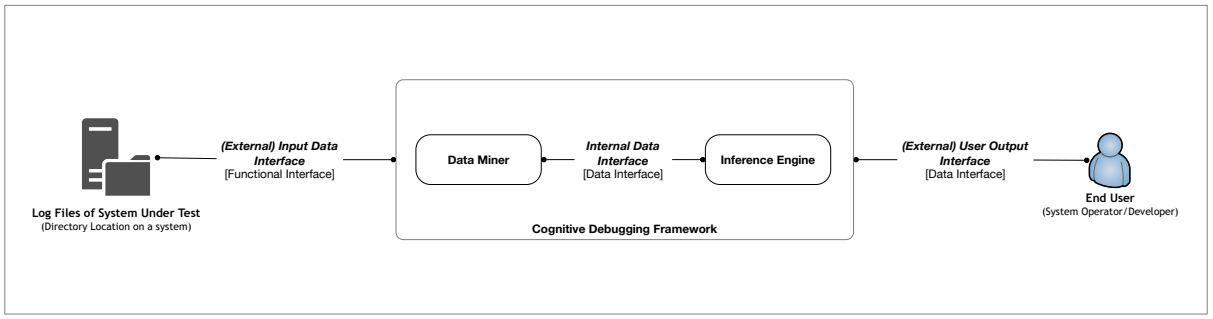


Figure 6.1: Diagram illustrating the major components of the Automated Log File Analysis Framework and the internal and external interfaces

Figure 6.1 also identifies the major internal and external interfaces of the ALFAF. As shown, there are three major interfaces in total: two external interfaces, and one internal interface. These are detailed in the following subsections.

6.2.1 Internal Interfaces

There is only one major internal interface of the ALFAF. This is the interface between the Data Miner and Inference Engine subcomponents of the framework as illustrated in Figure 6.1 and is called the *Internal Data Interface*.

This is a data interface and is defined by the structure and format of the data that is passed from the Data Miner to the Inference Engine. As discussed in Chapter 4, the Data Miner parses log files into a structured dataset in which each log message in the file is ultimately described by a log event template, a corresponding log event key and a list of variable parameters. In addition, each log message is also described by the metadata contained in the log message preamble. This dataset is structured in a pandas DataFrame data structure and is stored as a csv file.

As discussed in Chapter 5, the Inference Engine is designed to ingest a log file that was parsed by the Data Miner. During the Inference Engine's operation, the csv file containing the parsed log file is loaded into a pandas DataFrame and the columns of interest are extracted for further processing.

This interface is defined not only by the format of the data, but also the data fields describing each log message that are present in the parsed log file data structure. Figure 6.2 shows the mandatory data fields that are required in the parsed log file structure.

LineId	< . . .Preamble . . .>	Content	EventId	EventTemplate	ParameterList
Log *message* line number	*Various Metadata*	*Log Message Content*	*Log Key - unique log key representing the log event template*	*Event Template - template that describes the static part of the log message*	*List of variable parameters occurring in the log message*

Figure 6.2: Data structure and format of the data passed from the the Data Miner to the Inference Engine. Mandatory fields are also illustrated

The rows of the data structure in Figure 6.2 represent individual log messages from a given log file while the columns represent data fields describing the log message. The mandatory data fields required for this data interface, as shown in Figure 6.2, are:

- *LineId* - contains the line number of the log message
- *Content* - contains the raw log message content
- *EventId* - contains the log event key corresponding to the log message content
- *EventTemplate* - contains the log event template corresponding to the log message content
- *ParameterList* - contains a list of variable parameters extracted from the log message content

Depending on the structure of the log messages, additional data fields may be generated as described in Chapter 4). These are shown as the *Preamble* in Figure 6.2. These fields contain metadata information about the log messages, e.g. timestamps, source file names, and may be used for further processing or analysis but are not considered by the Inference Engine in this study.

6.2.2 External Interfaces

There are two major external interfaces of the ALFAF as shown in Figure 6.1. This includes the *Input Data Interface* to the *Log Files of System Under Test* and a *User Output Interface* to the *End-User*.

Input Data Interface

The ALFAF is designed to be tunable and adaptable to be able to perform automated log file analysis on log files produced by different systems. Most systems generate log files using raw text and these files are stored on disk. The ALFAF assumes this of all log files provided for analysis.

The *Input Data Interface* is a functional interface that provides access to the log files to be analysed to the ALFAF. The framework may be deployed on hardware forming part of the actual system under test that generates the log files, or it may be deployed on a separate, stand-alone system. To accommodate either scenario, the *Input Data Interface* specifies that access to log files be provided by the following:

- *path to directory* - an absolute path to a directory that contains the log files to be analysed by the ALFAF. Regardless of where the framework is deployed, this directory shall be accessible with *read* file privileges
- *log file name* - the file name, including extension, of the log file that is to be analysed or ingested by the framework. This log file shall exist in the directory specified above.

User Output Interface

The objective of the ALFAF is to perform automated log file analysis, with the aim of identifying possible anomalous events, errors and failures that have occurred during system operation, to provide information that operators and/or developers may use to assist in their efforts to debug the system failure.

The *User Output Interface* is a data interface that describes the data format, structure and required data fields of the output generated by the framework, that may be used by end-users to assist in debugging. The ALFAF generates two outputs after analysing a log file. These are the Debugging Report and the Suspicious Lines Report. Both reports are structured in pandas DataFrames and are stored as CSV files. The structure of these reports is shown in Figure 6.3. It should be noted that all data fields are applicable to all row entries in the data structure - the separation in Figure 6.3 is for visualisation purposes.

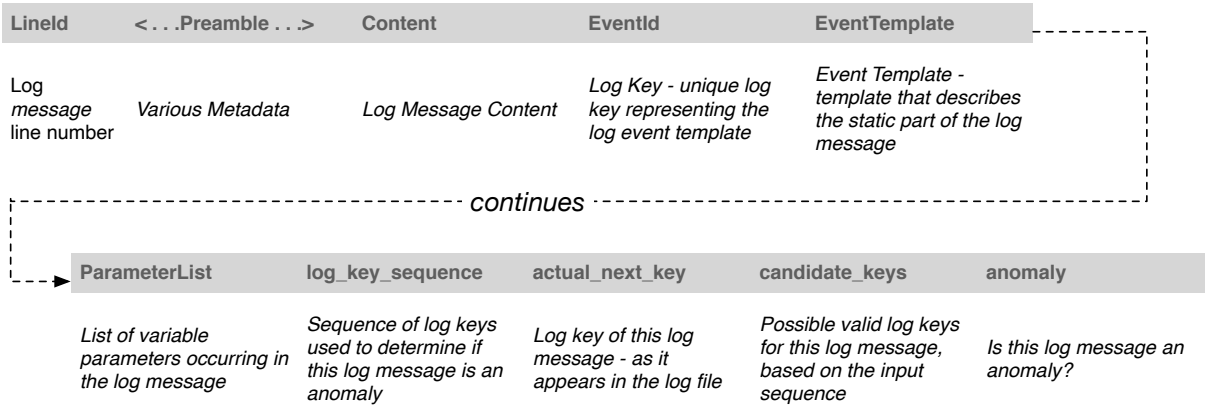


Figure 6.3: Data structure and format of the Debug and Suspicious Lines Reports

As shown in Figure 6.3, the data structure is based on the parsed log data structure shown in Figure 6.2, with additional data fields. Each row of the data structure again represents a single log message from a given log file. The additional fields are as follows:

- *log_key_seq* - contains the input sequence of log messages, represented as log event keys, used to determine whether a particular log message is anomalous or not
- *actual_next_key* - contains the actual log key that appeared in a given line in the log file
- *candidate_keys* - contains a list of the likeliest log messages, represented as log keys, that follow the given input sequence of log messages
- *anomaly* - contains a flag that specifies whether a particular log message is anomalous or

not. *1* represents an anomalous event, *0* represents a normal event.

Both the Debugging Report and the Suspicious Lines Report share the same data structure shown in Figure 6.3. The only difference between the two reports is that the Suspicious Lines report only contains the lines that have been flagged as anomalous, while the Debugging Report contains all lines contained in the entire log file. This approach enables faster debugging as operators or developers can consult the Suspicious Lines Report to identify which lines, as indicated by their line numbers in the original log file, are anomalous, and then cross-reference these with the Debugging Report to analyse the events leading to, and following, the anomalous event.

6.3 Automated Log File Analysis Framework Design

The Automated Log File Analysis Framework (ALFAF) is the complete, end-to-end framework capable of performing automated log file analysis on given log files to generate information that can aid in system fault and failure debugging. The ALFAF integrates the two previously designed subcomponents, the Data Miner and the Inference Engine, and provides additional processes to facilitate the entire automated log file analysis processing pipeline.

Given that the ALFAF employs data-driven approaches for the purposes of log file parsing and failure detection through anomaly detection, the framework has two modes of operation:

- *training mode* - the Data Miner and Inference Engine are trained on a set of system logs to learn the log event template to log key mapping and to train a model capable of detecting anomalies within log files
- *inference mode* - outputs, in the form of models and data stores, generated during *training mode* are used to run the framework in *inference mode* in which new, unseen log files, from the same system as used in *training mode*, are analysed to identify and detect failures in system operation.

It should be noted that while the ALFAF is designed to be capable of operating on log files generated by any system, the framework must be trained on log files that are generated by the same system on which the framework is to perform log file analysis. This holds true for both the Data Miner and Inference Engine subcomponents of the framework.

The design of the ALFAF is shown in Figure 6.4.

Figure 6.4 shows that the ALFAF consists of two major subcomponents, the Data Miner and Inference Engine, and two data stores: the *Log Key Data Store* and the *Model Data Store*. The Data Miner and Inference Engine subcomponents are detailed in Chapters 4 and 5 respectively. As shown in Figure 6.4, both subcomponents require pre-populated configuration files. These configuration files were previously detailed in the design chapters of the subcomponents.

The Log Key Data Store stores the log event key to log event template mappings derived by the Data Miner when operating in training mode. During inference mode, this data store is used to retrieve previously learned log event key to log event template mappings to ensure that log

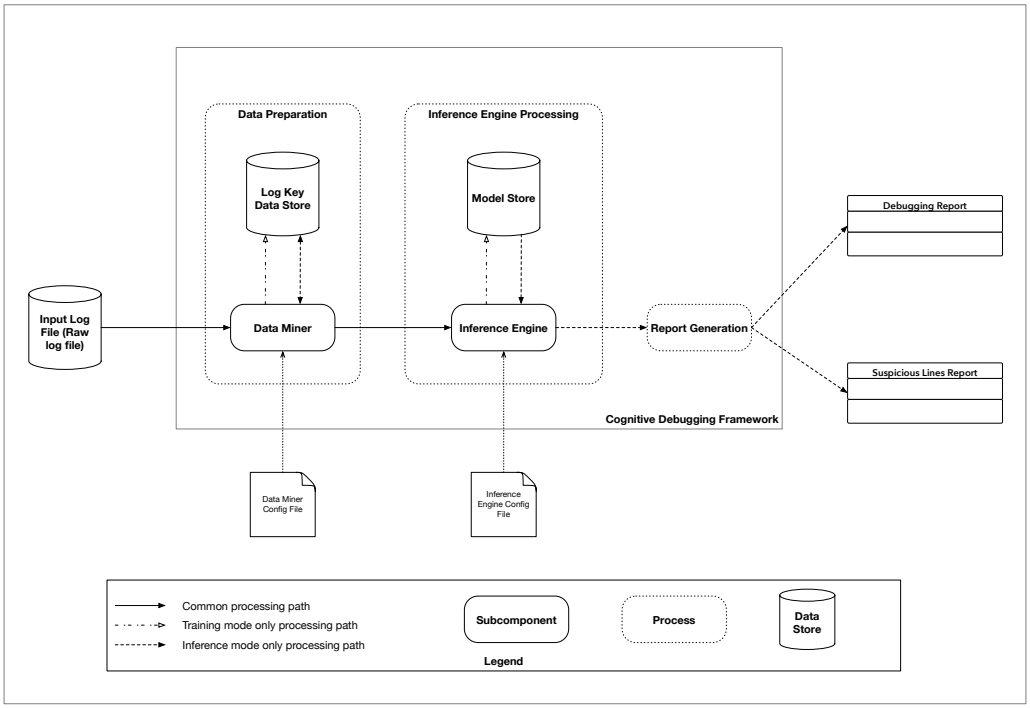


Figure 6.4: Diagram illustrating the design of the Automated Log File Analysis Framework. Major subcomponents and processes are illustrated. Training Mode and Inference Mode execution paths are also shown

event templates are consistently mapped to the same log event keys across log files for a given system.

The Model Data Store stores the learned model parameters describing the Anomaly Detection Model of the Inference Engine. When operating in training mode, the Anomaly Detection Model is trained on a given dataset to model the system behaviour. This model is described by a set of parameters stored in the Model Data Store. When operating in inference mode, the framework loads a previously learned model, corresponding to log files of the same system as the one under test, into the Anomaly Detection Model and uses this to detect anomalies in given log files.

As shown in Figure 6.4, the ALFAF has three major processing stages: *Data Preparation*, *Inference Engine Processing* and *Report Generation*. The first two processes are common to both modes of operation, while the *Report Generation* process only executes when the framework is operating in inference mode.

Data Preparation is the first processing stage and ingests the raw log files of the system under test. During this processing stage, the raw log files are parsed by the Data Miner into a structured dataset. If operating in training mode, the Log Key Data Store is created and the mapping of all unique log event key to log event templates in the structured dataset is stored in the Log Key Data Store.

If operating in inference mode, after parsing the log files into a structured dataset, a previously generated Log Key Data Store is queried to retrieve existing log event keys for any log events

occurring in the log file that are already contained in the Log Key Data Store. This is done to ensure that log events, of a given system's log files, are consistently mapped to the same log event keys across log files. This is important as the deep learning algorithm extracts features from these log event keys and these features are crucial to the ability of the Inference Engine to accurately detect anomalies within the log files. When operating in inference mode, the possibility exists that new log events, not already contained in the Log Key Data Store, are identified. To address this, an option to update the the Log Key Data Store when operating in inference mode is made available.

After the *Data Preparation* processing stage, the parsed, structured log file dataset is then passed to the next processing stage, *Inference Engine Processing*. During this stage, the ALFAF uses the Inference Engine to either train an anomaly detection model, or to detect anomalies within log files using a previously trained model, depending on whether it is run in training mode or inference mode.

In training mode, after training the Anomaly Detection Model, the Model Data Store is created and the learned model parameters are stored. In inference mode, previously learned model parameters are retrieved from the Model Data Store and are loaded into the Anomaly Detection Model to be used for inference.

If the framework is operated in training mode, the processing pipeline is complete after the *Inference Engine Processing* stage. If the framework is operated in inference mode, then an additional processing stage, *Report Generation*, is executed.

During the *Report Generation* processing stage, the output of the Inference Engine, an Anomaly Detection Report, is used to generate the Debugging Report and the Suspicious Lines Report as described in Section 6.2.

6.3.1 Implementation

The Automated Log File Analysis Framework (ALFAF) is implemented as a Python class named *AutomatedLFAFramework*. The UML Class Diagram for this class is shown in Figure 6.5.

Instantiating an instance of the *AutomatedLFAFramework* class requires the following arguments:

- *data_miner_config* - path to configuration file for the Data Miner
- *inference_engine_config* - path to configuration file for the Inference Engine
- *input_dir* - path to the directory where the log files to be considered are located
- *output_dir* - path to a directory where framework outputs are to be stored
- *name* - a unique identifier for an instance of the framework
- *device* - specifies which compute device, CPU or GPU, to use for model training and inference
- *mode* - specify the mode of operation. Either *training* or *inference*

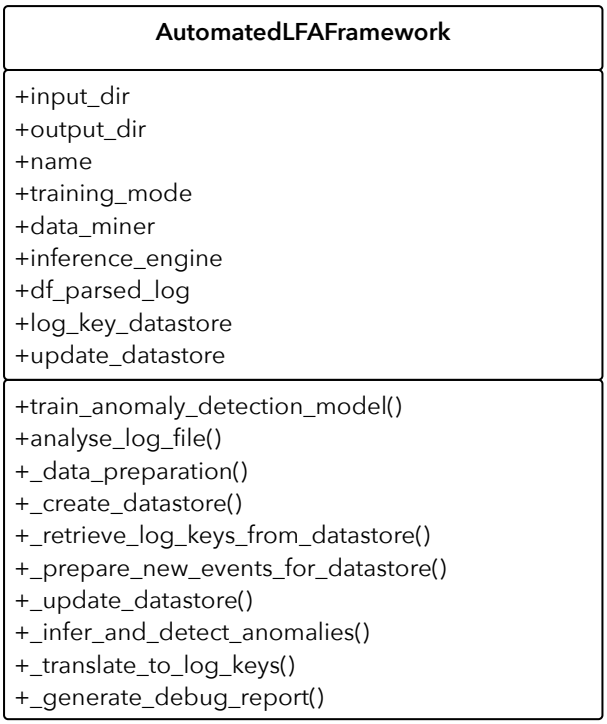


Figure 6.5: UML Class Diagram for the *AutomatedLFAFramework* class that implements the Automated Log File Analysis Framework

- *log_key_datastore* - path to a previously generated log key data store. Only required for inference mode
- *update_datastore* - flag to enable or disable updating of the log key store when operating in inference mode

The *input_dir*, *output_dir*, *name*, *log_key_datastore* and *update_datastore* attributes map to and store the corresponding class initialisation arguments. The *training_mode* attribute is a boolean flag that is set based on the value of the *mode* input argument. *df_parsed_log* is initialised to store a copy of the parsed log file, from the Data Miner, once available. The *data_miner* and *inference_engine* attributes are used to store instances of the *DataMiner* and *InferenceEngine* classes that implement the Data Miner and Inference Engine subcomponents respectively. It should be noted that the instantiation of both subcomponents requires the appropriate configuration files. These are described in Chapters 4 and 5.

The two modes of operation of the framework are invoked by two class methods, namely *train_anomaly_detection_model* for training mode and *analyse_log_file* for inference mode. These methods are provided with the file name of the log file to be considered as input, and implement the entire end-to-end processing pipeline for each mode of operation. A number of helper functions facilitate the various processes of the ALFAF.

The *data_preparation* method implements the *Data Preparation* processing stage. The *create_datastore*, *retrieve_log_keys_from_datastore*, *prepare_new_events_for_datastore* and *update_datastore* methods perform the various operations required to create the Log Key Data Store when in training mode and to retrieve log keys from the data store and update the Log Key Data Store

when in inference mode.

When operating in inference mode, the *analyse_log_file* method makes use of the *infer_and_detect_anomalies* method to generate the Anomaly Detection Report, and then subsequently calls the *translate_to_log_keys* and *generate_debug_report* methods to create the Debug and Suspicious Lines Reports.

All outputs generated by the ALFAF, whether operating in training mode or inference mode, are stored to disk in the directory specified by the *output_dir* class initialisation argument.

6.4 Framework Design Conclusion

This chapter described the integration of the Data Miner and Inference Engine subcomponents and the design of additional processes to realise, and complete, the design and development of the Automated Log File Analysis Framework.

The interfaces, both internal and external, of the ALFAF were identified and their implementation was detailed. Of particular importance to the end-to-end processing pipeline is the data interface between the Data Miner and Inference Engine that was described in Section 6.2.

The ALFAF's two modes of operation, namely *training mode* and *inference mode* were also described, and the various outputs generated were detailed. The structure of the Debug Report and the Suspicious Lines Report, and how they were designed to aid in system debugging, was presented.

The development and implementation of the ALFAF as a Python class was detailed and the inclusion of the previously designed Data Miner and Inference Engine component classes was discussed.

With the design and development of the Automated Log File Analysis Framework complete, the System Verification and Analysis Phase of the project, as described in Section 2.3, can now be considered. This is detailed in Chapter 7.

7

Algorithmic Tuning, Evaluation and Verification

With the Automated Log File Analysis Framework (ALFAF) and its subcomponents, the Data Miner and Inference Engine, now completely designed, this chapter details the tuning, evaluation and verification of the framework and its components.

The designs of the Data Miner, Inference Engine and the Framework, are firstly verified against the design and functional considerations and requirements put forward in Chapters 4, 5 and 6 respectively. These considerations and requirements shall be verified intrinsically as they are realised through the development of the subcomponents.

The Data Miner and Inference Engine, both having data-driven implementations for some of their subcomponents, require algorithmic tuning on suitable data to ensure they perform optimally. During this tuning, the maximising of certain performance metrics is considered. Once tuned, both the Data Miner and Inference Engine are then evaluated in terms of their performance to determine how well they perform their respective functions.

In addition to the design verification, the tuning and evaluation of the Data Miner is detailed in Section 7.2 and that of the Inference Engine is detailed in Section 7.3. The design verification of the overall Framework is detailed in Section 7.4. No performance evaluation of the Framework is considered as this is addressed through evaluation of the subcomponents.

The results and findings presented in this chapter are used as the basis for deeper analysis and discussion in Chapter 8.

7.1 Experimental Setup

This section describes the experimental setup used to perform algorithmic tuning and performance evaluation of the Data Miner and Inference Engine components. Section 7.1.1 details the compute platform used for all experiments and evaluation and Section 7.1.2 describes the datasets used through the various stages of tuning and evaluation.

7.1.1 Compute Platform

All algorithmic tuning and performance evaluation for this project was performed on a desktop computer that was made available, with the configuration shown in Table 7.1.

Table 7.1: *Specifications of the Compute Platform used for experiments in this study*

CPU	AMD Ryzen 7 5800X, 3.8 GHz
RAM	32 GB 3600 MHz DDR4
GPU	Nvidia RTX 3080
Storage	250 GB SATA III SSD

In these experiments, all Data Miner tuning and processing is run using the CPU, while the Inference Engine tuning and processing is run using the GPU.

7.1.2 Datasets

In order to be able to tune and quantitatively evaluate the performance of the Data Miner and Inference Engine components, suitable datasets, containing raw log files along with associated ground truths for log event templates, log event keys and system anomalies, are required.

A dataset consisting of system logs generated by various systems and stored using the Hadoop Distributed File System (*HDFS*) [86] was used as the dataset for tuning and evaluation of both components.

For Data Miner tuning and evaluation, a ground truth containing manually extracted log event templates and log event keys for a given log file is required. The developers of *Logparser*[7] have generated such datasets from system logs from various software systems, including HDFS. This dataset consists of a log file, from the HDFS, containing 2000 log messages, along with a manually-parsed version of this log file in which each log message is represented by a log event template and a log event key.

For the tuning and evaluation of the Inference Engine, a ground truth explicitly identifying anomalous log messages in a given log file is required. Suitable datasets, consisting of raw log files and an anomaly-identifying ground truth, are made available in *Loghub*[87] which is a collection of system log files made available for research purposes. One of the available datasets consists of log files from HDFS and consists of just over 11 million log messages spanning across 38.7 hours of system operation. An accompanying ground truth identifies which of the log messages were identified to be indicative of anomalous system behaviour. For the tuning and evaluation of the Inference Engine, this dataset is split into separate training, validation and testing sets. This is detailed in Section 7.3.

7.2 Data Miner Tuning and Verification

This section firstly details the design and functional verification of the Data Miner. Following that, the Data Miner is tuned to achieve optimal performance. The tuning process yields the optimal set of parameters for each of the available log parsing algorithms. The optimal set of

parameters is then finally used to evaluate the performance of the Data Miner across all of the available log parsing algorithms.

7.2.1 Design and Functional Verification

With reference to Section 4.1 in Chapter 4, the following design and functional considerations and requirements proposed for the Data Miner are considered and intrinsically verified:

Input Log Files The Data Miner is designed to ingest log messages produced by any system, stored as raw text in a file. The maximum size of log file that may be ingested is limited by the available compute capability and not the design of the Data Miner. This is detailed in Sections 4.3 and 4.6

Output Data After processing a given log file, the Data Miner generates two outputs: a structured log file in which each log message is represented by a log event template, corresponding log event key and extracted runtime parameters; and an event occurrence matrix that contains counts of each log event template occurring in the log file. This is detailed in Section 4.6.

Interfaces An interface that facilitates data transfer from the Data Miner to the Inference Engine exists and is detailed in Section 6.2.

Parsing Algorithm The Data Miner supports multiple, state-of-the-art log parsing algorithms including: *AEL*, *Drain*, *IPLoM*, *LenMa*, *LogMine* and *Spell*. An interface exposing the configurable parameters of each algorithm is also available and detailed in Section 4.6.

Usage The Data Miner was designed and developed as a stand-alone tool. It is run independently of the framework during the algorithm tuning and performance evaluation experiments detailed in this section.

Performance Metrics The Data Miner is capable of outputting the following performance metrics: *Parsing Accuracy* and *Parsing Time*. These are detailed further in this section.

7.2.2 Algorithmic Tuning

As detailed in Sections 4.5.1 and 4.7, each of the log parsing algorithms that the Data Miner supports has a set of configurable parameters that affect the performance of the algorithm on a given log file. In order to find the optimal set of parameters for each algorithm, algorithmic tuning of the Data Miner is required. The objective of this tuning is to find the optimal set of parameters, for a particular log parsing algorithm, that results in the highest parsing accuracy on log files from a given system. Parsing accuracy, detailed in Section 7.2.3, is the performance metric that best describes the ability of a log parsing algorithm to reduce a raw log to a structured, but still representative, dataset and has been shown to have the most impact on the performance of later log file analysis.

The log parsing algorithm tuning tool developed in Section 4.7 is used to tune the Data Miner. Tuning is performed for the Data Miner across all supported log parsing algorithms. The HDFS dataset with a ground truth structured log file is used.

The results of Data Miner algorithmic tuning, across all algorithms, are presented in Table 7.2.

Table 7.2: *Data Miner algorithmic tuning results*

	Log Parsing Algorithm					
Performance Parameter	**AEL**	**Drain**	**IPLoM**	**LenMa**	**LogMine**	**Spell**
Search Space Size	1919	2020	12648	101	600	101
Time Taken to Tune (seconds)	237.93	605.34	1849.35	28.89	1856.49	22.77
Minimum Parsing Accuracy	0.3500	0.3500	0.8420	0.4355	0.0000	0.3180
Maximum Parsing Accuracy	0.9975	0.9975	1.0000	0.9975	0.9975	1.0000

Table 7.2 shows the results of the algorithmic tuning of the Data Miner. As can be seen by the *Minimum and Maximum Parsing Accuracy*, the performance of the algorithms varies substantially as the configurable parameters change. All algorithms were capable of achieving a parsing accuracy greater than 99%, with the *IPLoM* and *Spell* algorithms able to achieve 100% parsing accuracy on the given dataset. The time required to tune each log parsing algorithm also varies substantially and is affected by both the total size of the parameter search space and the complexity and implementation of the log parsing algorithm.

The *Maximum Parsing Accuracy*, for each algorithm, is achieved when using the optimal set of parameters derived after tuning each algorithm. These optimal parameter sets are used in further performance evaluation of the Data Miner. It should be noted that these optimal parameters were derived through tuning the Data Miner on the HDFS dataset and these are not expected to perform favourably on log files produced by other systems. For reference, the optimal parameters that results in the best performance from each algorithm is shown in Table 7.3.

Table 7.3: *Optimal parameters for the various log parsing algorithms after tuning on the HDFS dataset*

Log Parsing Algorithm	**Optimal Parameters**
AEL	Merge Percentage: 0.25 Minimum Event Count: 2
Drain	Parse Tree Depth: 1 Similarity Threshold: 0.12
IPLoM	Cluster Goodness Threshold: 0.3 Partition Support Threshold: 0 Lower Bound: 0.01 Upper Bound: 0.9 File Support Threshold: 0
LenMa	Clustering Similarity Threshold: 0.54
LogMine	Clustering Parameter (k): 0.1 Clustering Hierarchy Levels: 2 Maximum Distance: 0.004
Spell	Longest Common Subsequence Threshold (Tau): 0.67

7.2.3 Performance Evaluation

The performance of the Data Miner is evaluated in terms of: *Parsing Accuracy*, *Parsing Time*, *Efficiency* and *Robustness*. These metrics, how they are measured and the results of the Data Miner's performance evaluation are detailed in this section. During the performance evaluation process, each log parsing algorithm is configured with the optimal set of parameters as derived during algorithm tuning in Section 7.2.2 and detailed in Table 7.3.

Parsing Accuracy

Parsing accuracy is defined as the ratio of correctly parsed log messages to all log messages in a given log file [40]. This is shown in Equation 7.1. Measuring parsing accuracy requires a ground truth of parsed log messages for a given log file.

$$\text{parsing accuracy} = \frac{\text{\# correctly parsed log messages}}{\text{\# total log messages}} \tag{7.1}$$

The log parsing algorithm parses each log message into an event template which consists of the static part of the log message and the variable parts represented by wildcards ($*$). The number of correctly parsed log messages is determined by considering groups of messages in the parsed log file generated by the log parsing algorithm. A group of log messages, represented by the same log event template, is considered to be correctly parsed if two conditions are satisfied. Firstly, there is only a single corresponding log event describing these log messages in the ground truth, and secondly, the number of log messages represented by the event template in the generated parsed log is equal to the number of log messages represented by the corresponding event template in the ground truth. If these conditions are satisfied then the total number of log messages, corresponding to this specific event template, is added to the number of correctly parsed log messages.

The parsing accuracy metric describes how well a given log parsing algorithm is able to correctly extract log event templates from a log file. This metric is of particular importance when the parsed log files are to be used for later log file analysis processes.

The results of the Data Miner Parsing Accuracy tests are shown in Table 7.4.

Table 7.4: *Data Miner Parsing Accuracy Test Results*

	Log Parsing Algorithm					
Performance Metric	**AEL**	**Drain**	**IPLoM**	**LenMa**	**LogMine**	**Spell**
Parsing Accuracy	0.9975	0.9975	1.0000	0.9975	0.9975	1.0000
Unique Events Extracted	16	16	14	16	16	14
Time Taken (seconds)	0.08	0.16	0.14	0.27	2.50	0.17

As previously noted, when tuned, all log parsing algorithms are capable of achieving a very high parsing accuracy. Two algorithms are able to achieve a perfect parsing accuracy score, and similarly have identified the same number of unique log event templates from the dataset.

Parsing Time and Efficiency

Parsing time is simply measured as the time taken for a log parsing algorithm to complete parsing of a given log file. The efficiency of each log parsing algorithm is evaluated by measuring the parsing time on log files of increasing size to determine how the parsing time scales.

The results of the Data Miner Parsing Efficiency Tests are shown in Table 7.5. These results are also illustrated in Figure 7.1

Table 7.5: *Data Miner Parsing Efficiency Test Results*

	Log Parsing Algorithm Parsing Time (s)					
Log Message Count	**AEL**	**Drain**	**IPLoM**	**LenMa**	**LogMine**	**Spell**
1k	0.05	0.09	0.07	0.13	0.61	0.12
10k	0.39	0.79	0.66	1.32	58.43	0.81
100k	3.86	8.08	6.78	13.14	DNF	8.12
1000k	47.01	81.47	71.66	129.61	DNF	84.68
10000k	778.20	808.75	712.36	1569.39	DNF	839.56

DNF = Did not finish within 1 hour

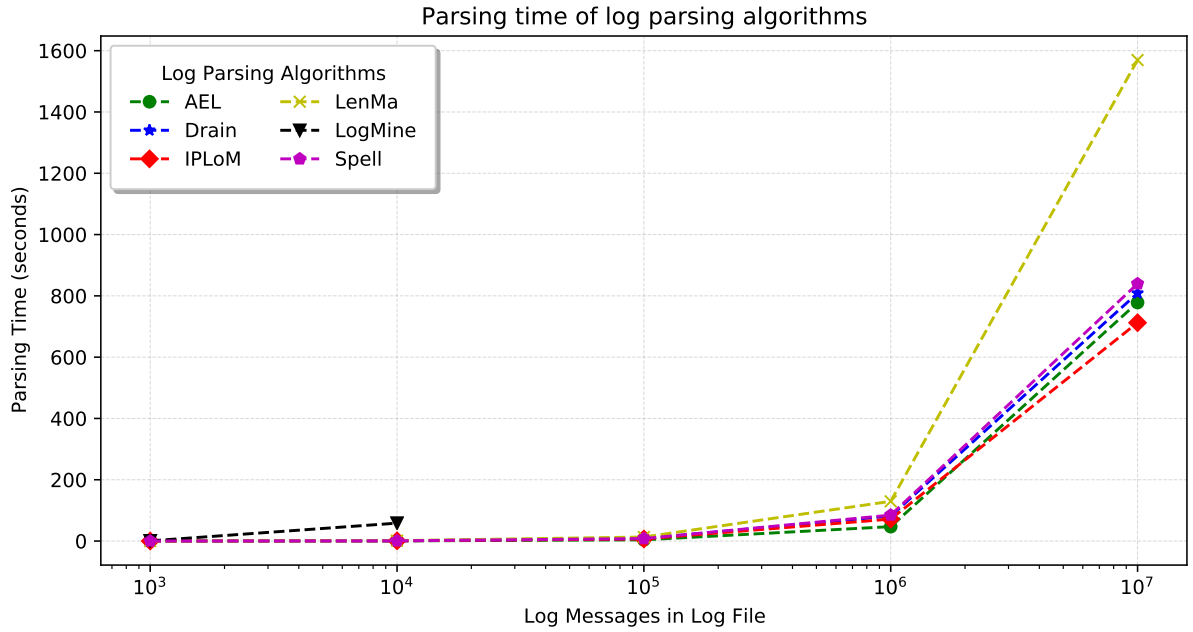


Figure 7.1: Plot showing efficiency of log parsing algorithms

As shown in Figure 7.1, up to a log files containing 1 million log messages, most of the log parsing algorithms' parsing times scale in a similar manner. The *LogMine* algorithm was unable to complete parsing a log file containing 100 000 log messages in under an hour. The *LenMa* algorithm's parsing time increases substantially beyond log files containing 1 million messages and shifts away from the scaling exhibited by the other algorithms.

Robustness

The robustness of the Data Miner is assessed in terms of how well the various log parsing algorithms, tuned toward log files from a particular system, perform on log files generated by

different systems.

To measure the robustness, three additional log file datasets from other systems were used. These systems were *OpenStack*, the *Blue Gene/L Supercomputer* and the *Thunderbird Supercomputer*. These datasets contained raw log files in addition to a ground truth of parsed log messages. These datasets were obtained from Logparser [7].

Each log parsing algorithm was configured to use the optimal parameters, derived through algorithmic tuning on the HDFS dataset and then run on the other three datasets to measure the achieved parsing accuracy.

The results of the Data Miner Robustness Tests are shown in Table 7.6.

Table 7.6: *Data Miner Robustness Test Results*

	AEL	**Drain**	**IPLoM**	**LenMa**	**LogMine**	**Spell**
HDFS Parsing Accuracy	0.9975	0.9975	1.0000	0.9975	0.9975	1.0000
BGL Parsing Accuracy	0.8215	0.8630	0.9375	0.6895	0.7805	0.7615
OpenStack Parsing Accuracy	0.2950	0.1850	0.3095	0.3305	0.2850	0.2520
Thunderbird Parsing Accuracy	0.6530	0.5135	0.6630	0.9430	0.6520	0.6445

The results from the robustness tests show that the performance of the log parsing algorithms on a particular system's log files is very sensitive to the parameters used to configure the algorithm. There is also no observable pattern in the performance reduction, which indicates this variance is attributed to the structure of the log messages themselves.

7.3 Inference Engine Tuning and Verification

This section details the verification, tuning and evaluation of the Inference Engine subcomponent. The design and functional considerations and requirements are first intrinsically verified to confirm the Inference Engine was designed as intended. The Inference Engine is then tuned on the HDFS dataset to achieve optimal performance. Once the optimal configuration of the Inference Engine has been found, various experiments and tests that evaluate the performance of the Inference Engine are conducted to understand its capability and behaviour.

7.3.1 Design and Functional Verification

With reference to Section 5.1 in Chapter 5, the following design and functional considerations and requirements proposed for the Inference Engine are considered and intrinsically verified:

Input Data The Inference Engine was designed to process log files that have been processed by the Data Miner. These log files contain log event keys representing each log message. This is described in Sections 5.5 and 6.2.

Output Data Depending on the mode of operation, the Inference Engine's output varies. In training mode, a list of parameters describing a trained anomaly detection model is generated. In inference mode, an *Anomaly Detection Report* that flags potential anomalous log messages is generated. The outputs of the Inference Engine are detailed in Section 5.5.

Interfaces The interface between the Data Miner and Inference Engine is detailed in Section 6.2. An interface that facilitates data transfer from the Inference Engine to the framework also exists and is described in the same section.

Features The Inference Engine extracts features from a given parsed log file that represents the expected execution path of a given system. The features are a sequence of log messages, represented by log event keys. This is detailed in Section 5.3.

Deep Learning Algorithm The Anomaly Detection Model subcomponent of the Inference Engine utilizes a Recurrent Neural Network. This network is trained to predict the next likeliest log event for a given input sequence of log events or messages. The design of this model is described in Section 5.4.

Deep Learning Framework The development of the Inference Engine utilizes the *PyTorch* framework for implementing the Anomaly Detection Model.

Usage The Inference Engine was designed and developed as a stand-alone tool. It is run independently of the framework to perform the algorithmic tuning and performance evaluation in this section.

Processing Mode The Inference Engine was designed to only process log files in an offline manner. Real-time stream processing of log messages is not considered.

Performance Metrics The Inference Engine was designed to output the following performance metrics: *Model Accuracy*, *Precision*, *Recall*, F_1 *Measure* and *False Alarm Rate*. These performance metrics are detailed later in this section.

7.3.2 Algorithmic Tuning

Before being able to detect anomalies through log file analysis, the Inference Engine must be trained to model the expected behaviour of the system under test. This modelling involves training and tuning the Inference Engine's LSTM-based Anomaly Detection Model using log files that are generated during normal system operation. During this tuning and training process, the accuracy of the model is optimized. This *Model Accuracy* represents the model's ability to correctly predict the next log event for a given input sequence of log events. A higher *Model Accuracy* means that the model more accurately represents the expected behaviour of the system. In addition, an anomaly-labelled ground truth is unlikely to exist for most production systems. Therefore, training the model to optimize anomaly detection specific metrics would limit the usefulness and practicality of the Inference Engine.

The LSTM-based Anomaly Detection Model is trained using a subset of the HDFS dataset described in Section 7.1.2. A total of 1 000 000 log messages are extracted from the raw HDFS log file and the ordering of the log messages is preserved. This subset of log messages is then parsed using the Data Miner, configured to use the *Spell* algorithm, to generate a parsed, structured log file that the Inference Engine can process. This parsed log file is finally split into three subsets, a *training set*, a *validation set* and a *testing set* using a ratio of *64:16:20*. These split datasets are used during different stages of algorithmic tuning and performance evaluation

of the Inference Engine.

Before training the LSTM-based model, the various hyper-parameters of the model are tuned. This tuning was performed by training various instances of LSTM models instantiated with various combinations of the hyper-parameters and identifying which combination yields the highest *Model Accuracy*. The search space considered for the model hyper-parameters is shown in Table 7.7.

Table 7.7: *Search space for the various model parameters*

Model Parameter	**Search Space**
Hidden Size	16, 32, 64, 128, 256
Window Size	[1, 15]
Bidirectional	Yes, No
Number of LSTM Layers	[1,5]
Batch Size	32, 128, 1024, 2048

Note: Number of epochs trained for was increased as Batch Size increased.

The LSTM-model was tuned using the *training set* of parsed log messages. After searching the entire hyper-parameter search space, the optimal combination of hyper-parameters was found and is shown in Table 7.8. This set of hyper-parameters was used for all further performance evaluation of the Inference Engine.

Table 7.8: *Optimal hyper-parameters for the LSTM model of the Inference Engine*

Model Parameter	**Optimal Value**
Hidden Size	256
Window Size	10
Bidirectional LSTM	Yes
Number of LSTM Layers	2
Batch Size	1024

7.3.3 Performance Evaluation

The performance of the Inference Engine is evaluated from two different perspectives. Firstly, the Inference Engine's performance in terms of accurately predicting the next log event key for a given input sequence of log event keys is determined by measuring the accuracy of the Anomaly Detection LSTM. This was the *Model Accuracy* metric that was optimized for during algorithmic tuning.

Secondly, the Inference Engine is also evaluated in terms of its Anomaly Detection Performance. This provides insight into how well the Inference Engine can detect anomalous log messages in given log files. The Anomaly Detection Performance is described by a number of metrics:

- True Positives (t_p) - the number of log messages correctly identified as being anomalous
- True Negatives (t_n) - the number of log messages correctly identified as being normal
- False Positives (f_p) - the number of log messages incorrectly identified as being anomalous

- False Negatives (f_n) - the number of log messages incorrectly identified as being normal
- Precision - the proportion of all *positive* identifications that is correct. Given by $Precision = \frac{t_p}{t_p+f_p}$
- Recall - the proportion of all actual positives that is correctly identified. Given by $Recall = \frac{t_p}{t_p+f_n}$
- F_1 Score - accuracy measure that considers both precision and recall. Given by $F_1 = 2 * \frac{precision*recall}{precision+recall}$
- False Alarm Rate - the proportion of all actual *negative* identifications that are incorrectly identified as positive. Given by False Alarm Rate $= \frac{f_p}{t_n+f_p}$

It should be noted that computing these metrics requires a ground truth dataset in which anomalous log messages are identified and labelled.

Anomaly Detection Performance

When performing Anomaly Detection, the Inference Engine has a tunable parameter that defines the criterion used to decide whether or not a log message is anomalous. This parameter is the *num_candidates* parameter described in Section 5.5. This parameter does not affect the LSTM-model and is therefore not considered during tuning. Instead, its impact is considered when evaluating the Anomaly Detection Performance.

To evaluate the performance of the Inference Engine, a model is first trained on the training dataset using the optimal set of hyper-parameters derived during algorithm tuning. This model is then used by the Inference Engine in *inference* mode to detect anomalies in the HDFS validation dataset. The *num_candidates* parameter is varied and the performance measurements are shown in Table 7.9

Table 7.9: *Anomaly Detection Performance of Inference Engine on Validation Dataset*

	Number of Candidates Log Keys								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Model Accuracy	0.9404	0.9404	0.9404	0.9404	0.9404	0.9404	0.9404	0.9404	0.9404
True Positives	66	48	39	39	38	38	37	37	37
True Negatives	8360	9469	9972	10051	10085	10114	10114	10114	10114
False Positives	1754	645	142	63	29	0	0	0	0
False Negatives	406	424	433	433	434	434	435	435	435
Precision	0.036	0.069	0.215	0.382	0.567	1.000	1.000	1.000	1.000
Recall	0.140	0.102	0.083	0.083	0.081	0.081	0.078	0.078	0.078
F1 Measure	0.058	0.082	0.119	0.136	0.141	0.149	0.145	0.145	0.145
False Alaram Rate	0.173	0.064	0.014	0.006	0.003	0.000	0.000	0.000	0.000

The *precision* and *recall* metrics are plotted in Figure 7.2.

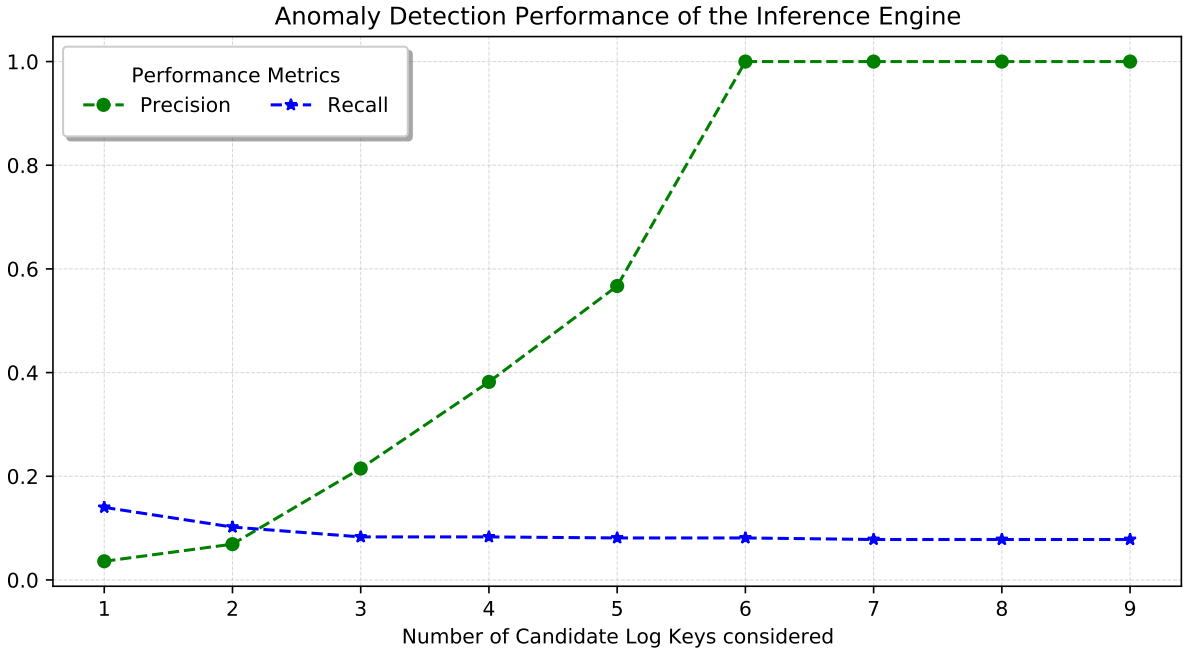


Figure 7.2: Plot showing the precision and recall performance of the Inference Engine on the 1000k HDFS dataset for various numbers of candidate log keys

The results of performance testing reveal that there is a trade-off to be made between achieving

maximum *precision* or maximum *recall* when varying the *num_candidates* parameter. It is also seen that while the *Model Accuracy* is high, the F_1 *Measure* performance is quite low. Further inspection into the generated Anomaly Detection Reports yield additional insights into this behaviour and is discussed further in Chapter 8.

For the purposes of finalising the performance evaluation, the optimal value for the *num_candidates* parameter is chosen as 6. Using this, the trained model is used to perform inference on the HDFS test dataset. The results are shown in Table 7.10.

Table 7.10: *Anomaly Detection Performance of Inference Engine on Test Dataset*

Model Accuracy	0.9392
True Positives	35
True Negatives	12638
False Positives	6
False Negatives	556
Precision	0.8537
Recall	0.0592
F1 Measure	0.1108
False Alarm Rate	0.0005

Note: num_candidates = 6

7.4 Framework Verification

The design and functional considerations and requirements proposed for the Data Miner and Inference Engine components are applicable to the framework and are realised and verified through the design and development of these subcomponents. In addition, to support the framework in achieving its main objective, additional considerations are proposed at the top level. These considerations are intrinsically verified in this section. Additionally, the case study conducted in Chapter 8 serves as evidence for the verification of the framework design and functional considerations and requirements.

7.4.1 Design and Functional Verification

With reference to Section 6.1 in Chapter 6, the following design and functional considerations and requirements proposed for the framework are considered and intrinsically verified:

Input The framework, through the Data Miner component is capable of ingesting raw, system-generated log files stored as raw text. The framework also provides an interface for accessing these stored log files as detailed in Section 6.2.

Output After the complete processing pipeline, the framework outputs two files. A *Debugging Report* containing the entire, parsed log file with anomalous messages identified, and a *Suspicious Lines Report* containing only the suspicious lines as identified by the framework. This is detailed in Section 6.3 and demonstrated in Section 8.1.3.

Operational Modes As detailed in Section 6.3, the framework provides a training mode in which the Data Miner and Inference Engine are trained on log files from a given system. And

an inference mode in which the Data Miner and Inference Engine are used to detect anomalies in new log files generated by the same system. The framework's operation in training mode is detailed in Section 8.1.2 and inference mode operation is demonstrated in Section 8.1.3.

Data Stores The framework provides data stores for storing previously learned log event key and log event template mappings to ensure consistency in log parsing and for storing previously trained anomaly detection models. The implementation of the data stores is detailed in Section 6.3. The generation of these data stores is demonstrated in Section 8.1.2.

Interfaces The interface between the Data Miner and Inference Engine was already previously verified. The external interfaces to the system under test and the end-user is detailed in Section 6.2. The functionality of the external interfaces is demonstrated in Section 8.1.

7.5 Conclusion

This chapter detailed the verification, tuning and performance evaluation of the Automated Log File Analysis Framework and its subcomponents.

Section 7.2 considered the design verification of the Data Miner and detailed the algorithmic tuning process required for this component. After tuning, the performance of the Data Miner was evaluated.

The Inference Engine was then considered in Section 7.3. After verifying the design, the tuning and training of the Inference Engine was detailed. Using a tuned and trained model, the Inference Engine was then evaluated it terms of its system modelling performance and its anomaly detection performance.

The framework's design was verified in Section 7.4. The performance evaluations of the Data Miner and Inference Engine are directly applicable to the Framework and no further performance testing was required for the Framework.

The next chapter presents a case study in which the framework is adapted to work on log files generated by a real-world, production system. The findings of this case study, and the results presented in this chapter are used to form the basis of further analysis and discussion in Chapter 8.

8

Analysis and Discussion

This chapter presents a detailed analysis of the design, operation and performance of the Automated Log File Analysis Framework (ALFAF) with the aim of gaining insight into how well it performs its objectives. The analysis and discussion presented in this chapter follow on from the verification and performance evaluation presented in Chapter 7, and together these findings form the basis of the answers to the research questions put forward for this dissertation in Section 1.2.

Before analysing and discussing the Automated Log File Analysis Framework, a case study is conducted in which the ALFAF is tuned toward and trained to detect anomalies on a real-world system currently in development. The case study system is the MeerKAT Radio Telescope's Correlator Beamformer that was introduced in Section 3.1.4. The objective of this case study is to gain insight into the operation and performance of the ALFAF on a real system, using actual log files generated by the system. Through this case study, the viability of the ALFAF, as an effective tool in system fault identification and debugging, shall be considered. It should be noted that given the absence of a ground truth dataset i.e. with anomalies labelled, the evaluation of the framework's performance in this case study will be partially subjective and perceptive. The case study is detailed in Section 8.1.

Following the case study, Section 8.2, supported by the findings presented in Chapter 7 and the case study in Section 8.1, examines the overall operation, performance and design of the ALFAF. This examination addresses the performance of the various components of the framework and considers which factors influence their performance and that of the overall framework. Shortcomings of the ALFAF are also considered and possible resolutions are also provided. The section concludes by discussing the viability of the ALFAF as an effective and robust end-to-end tool for implementing automated log file analysis with the objective of identifying system errors and failures.

8.1 Case Study: MeerKAT Correlator Beamformer

This section details a case study that was conducted in which the designed ALFAF is used on log files generated by the MeerKAT Radio Telescope's Correlator Beamformer (CBF) subsystem.

This subsystem was previously detailed in Section 3.1.4 of the literature review. This case study aims to accomplish three objectives. The first objective is to test and validate the end-to-end operation of the ALFAF on real-world system logs to determine whether or not the framework performs its main objective of providing a pipeline for automated log file analysis including log parsing and anomaly detection. The second objective is to evaluate the viability of using the ALFAF on the CBF log files. And lastly, the final objective is to understand the behaviour of the ALFAF and gain insight into how log messages and files should be structured for optimal anomaly detection performance.

The case study is laid out as follows: Section 8.1.1 briefly introduces the dataset of log files considered during this case study, Section 8.1.2 details the application and operation of the ALFAF in *training* mode to learn to model the CBF system, Section 8.1.3 evaluates the ALFAF when operating in *inference* mode and finally, Section 8.1.4 analyses and discusses the findings of the case study.

8.1.1 Dataset

The log files considered for this case study were generated by the *corr2* software package that runs as part of the software stack for the CBF. A new log file is generated by this software package for every new instance of the *corr2 servlet* that is started. Typically, a new instance is started for each unique observation that the telescope is to perform. The *corr2* log file, for each instance of the correlator or similarly each observation, details the *instrument start-up* or initialisation process of the CBF. It logs all activities involved in configuring the SKARAB processing nodes, assigning IP addresses to input and output data streams and configuring the various parameters of the CBF.

For this case study, a selection of ten *corr2* log files were considered. These log files were selected as they are representative of the various modes of operation the CBF may be configured for. The size of these log files range from hundreds to thousands of log messages, depending on the size of the instrument instantiated. Since each log file represents a different instance of the CBF, each log file, in its entirety, is considered as a unique dataset. A selection of the log files were used during training and the rest were used for inference.

The log files contain raw text describing each log message and are stored with the *.log* file extension. An extract from the CBF log files in shown in Figure 8.1.

```
2020-04-22 16:14:38,879 - INFO - array_1.wide.bc8n856m4k instrument.py:65 - FxCorrelator array_1.wide.bc8n856m4k created.
2020-04-22 16:14:38,890 - INFO - array_1.wide.bc8n856m4k fxcorrelator.py:120 -
========================================
Successfully created Instrument: array_1.wide.bc8n856m4k
========================================

2020-04-22 16:14:38,895 - INFO - array_1.wide.bc8n856m4k.katcp.server server.py:567 - Reading loop for client 127.0.0.1:40577 completed
2020-04-22 16:14:38,898 - INFO - array_1.wide.bc8n856m4k instrument.py:283 - Set synch epoch to 1587450614.00000.
2020-04-22 16:14:38,898 - INFO - array_1.wide.bc8n856m4k.katcp.server server.py:567 - Reading loop for client 127.0.0.1:40578 completed
2020-04-22 16:14:38,900 - INFO - array_1.wide.bc8n856m4k fxcorrelator.py:145 - Sync epoch pre-set to 1587450614.0.
2020-04-22 16:15:39,254 - INFO - array_1.wide.bc8n856m4k.skarab02042e-01 transport_skarab.py:2753 - Skarab is rebooting from SDRAM.
2020-04-22 16:15:39,254 - INFO - array_1.wide.bc8n856m4k.skarab02060a-01 transport_skarab.py:2753 - Skarab is rebooting from SDRAM.
```

Figure 8.1: Example log file from the MeerKAT Correlator Beamformer

8.1.2 Training the Framework

Before the ALFAF may be used to perform automated log file analysis, its two subcomponents, the Data Miner and the Inference Engine, must be properly prepared for, and tuned toward, the log files of the system under test. The tuning and training of the ALFAF subcomponents, in preparation for using the ALFAF for inference on new, unseen log files, is explained in the following subsections.

Data Miner Preparation

Preparing the Data Miner subcomponent of ALFAF for training requires three key pieces of information to be specified in the Data Miner configuration file. First, the log message format identifying the *log message preamble* and log message content component, as described in Section 4.3, is required. Second, an optional list of regular expressions describing certain runtime variable components expected in the log messages may be provided, and finally, the desired log parsing algorithm and its associated configurable parameters must be specified.

Log Message Format The log message format of the CBF log messages is shown in Figure 8.2. This figure clearly distinguishes between the *log message preamble* and the log message content. The format is captured in the Data Miner configuration file, as the *log_format* parameter, as a string consisting of:

<Date><Time>- <Level>- <Instrument><Package>:<Line Number>- <Content>

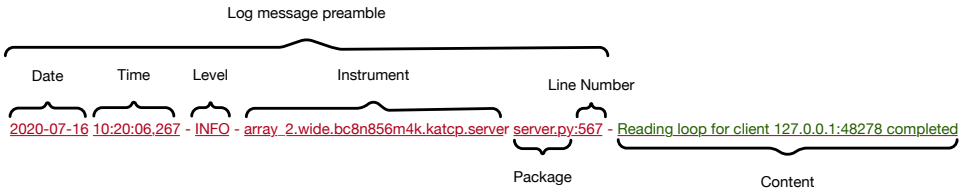


Figure 8.2: Log message format of the CBF log messages

Preprocessing Regular Expressions An examination of the CBF log files reveals numerous repeating runtime variable formats. Regular expressions are derived for matching these variables in log messages to assist with log parsing. The identified runtime variable formats, their associated regular expressions, and an example of each variable, are presented below:

- IP Addresses: *(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}:\d+)*, e.g. 127.0.0.1:33613
- Processing Node Identifier: *(skarab\w+-\d+)*, e.g. skarab020606-01
- CBF Input Labels: *([md]\d\d\d[hv])*, e.g. m002v
- Data Stream Addresses: *(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d1,3\+\d+:\d+)*, e.g. 239.10.8.40+7:7148
- Ethernet Interfaces: *(gbe\d+)*, e.g. gbe0
- X-Host Identifier: *(Xhost\s+\d+)*, e.g. Xhost 1
- Beam Name: *(tied-array-channelised-voltage\.\d+[xy])*, e.g. tied-array-channelised-voltage.0x

These regular expressions are provided as a list for the *preprocess* parameter in the Data Miner configuration file.

Log Parsing Algorithm The final piece of information that is to be provided for the Data Miner is the desired log parsing algorithm and its configurable parameters. As evidenced by the results obtained in Section 7.2.3, log parsing algorithms tuned toward a particular system's log files do not generalise well across log files from different systems. Therefore, to select the optimal log parsing algorithm, and its set of configurable parameters, for the CBF log files, the log parsing tuning tool is used to tune the various log parsing algorithms toward the CBF log files. To enable this tuning to take place, a ground truth dataset consisting of manually extracted log event templates is created using regular expressions and the Regex-based log parser developed in Chapter 4. The results of tuning the log parsing algorithms toward the CBF log files is shown in Table 8.1.

Table 8.1: *Performance of the various log parsing algorithms on the CBF log files*

	Log Parsing Algorithm					
Performance Metric	**AEL**	**Drain**	**IPLoM**	**LenMa**	**LogMine**	**Spell**
Accuracy	0.9024	1.0000	0.9390	0.9024	0.9390	0.9756
Precision	0.9727	1.0000	0.9652	1.0000	0.9652	0.9911
Recall	0.9640	1.0000	1.0000	0.7477	1.0000	1.0000
F1 Measure	0.9683	1.0000	0.9823	0.8557	0.9823	0.9955
Unique Events	39	40	37	47	37	39
Time Taken	0.011881	0.014591	0.011747	0.012331	0.016125	0.014876

As shown in Table 8.1, the *Drain* algorithm, after being tuned, is able to parse the CBF log files with perfect accuracy. The *AEL* and *Spell* algorithms also perform well and inspection of the parsed logs generated by these algorithms revealed that these algorithms merged two static log messages that only differed by a single token, and treated this token as a variable parameter. These particular log messages, although very similar in structure, were in fact not generated by the same event. This behaviour can be addressed with additional regular expressions for these particular log messages. For the purposes of this case study however, the *Drain* log parsing algorithm is selected as the log parsing algorithm to use.

Before concluding the Data Miner preparation the impact of the optional pre-processing regular expressions was evaluated. All available log parsing algorithms were again used to process the CBF log file but this time, the pre-processing regular expressions were omitted. With the exception of *Drain*, all log parsing algorithms performed worse without pre-processing regular expressions. *Drain* was still able to achieve perfect parsing accuracy for this particular log file, but some runtime variable components of log messages were not extracted correctly. This was particularly evident for log messages that didn't occur very often. This emphasises the importance of the pre-processing regular expressions, for identifying common variable components in the log messages, when aiming to achieve optimal log parsing algorithm performance.

Inference Engine Preparation

Before the Inference Engine component of the ALFAF can be trained for the CBF log files, the optimal model parameters for the Inference Engine's LSTM-based anomaly detection model must be found. The same process, and model hyper-parameter search space, as outlined in Section 7.3 was used, except in this instance, the CBF log files were considered. The optimal model parameters for the CBF log files that resulted in the highest model accuracy is shown in Table 8.2.

Table 8.2: *Optimal hyper-parameters for the LSTM model of the Inference Engine for the CBF log files*

Model Parameter	**Optimal Value**
Hidden Size	256
Window Size	7
Bidirectional LSTM	Yes
Number of LSTM Layers	2
Batch Size	32

These model hyper-parameters were captured in the Inference Engine configuration file. Given the comparatively small number of log messages in the CBF log files, the number of epochs to train for was set to 500.

Output of the ALFAF in Training Mode

With the Data Miner and Inference Engine subcomponents fully prepared and configured - by means of their respective configuration files - the ALFAF was then ready to be run in *training* mode to generate the Log Event Key Data Store and train the LSTM-based anomaly detection model for the CBF logs.

The ALFAF was run in *training* mode and extracts of the training process, showing the various outputs of the ALFAF, are shown in the following figures. Figure 8.3 shows the outputs generated after the feature extraction stage of processing.

As seen in Figure 8.3, the transformation mapping and Log Event Key Datastore are generated as outputs by the ALFAF in training mode. These are saved to *yaml* and *csv* files respectively for later use by the ALFAF in *inference* mode. The number of unique classes, representing the number of unique log events, is also derived for the system during training and is outputted. This value is used when configuring the Inference Engine for *inference* mode.

Figure 8.4 shows the outputs generated after the Inference Engine is trained.

After Inference Engine training, as shown in Figure 8.4, the learned model parameters are generated as outputs by the ALFAF and saved to a file. These outputs are required for running the ALFAF in *inference* mode at a later stage to detect anomalies within given log files.

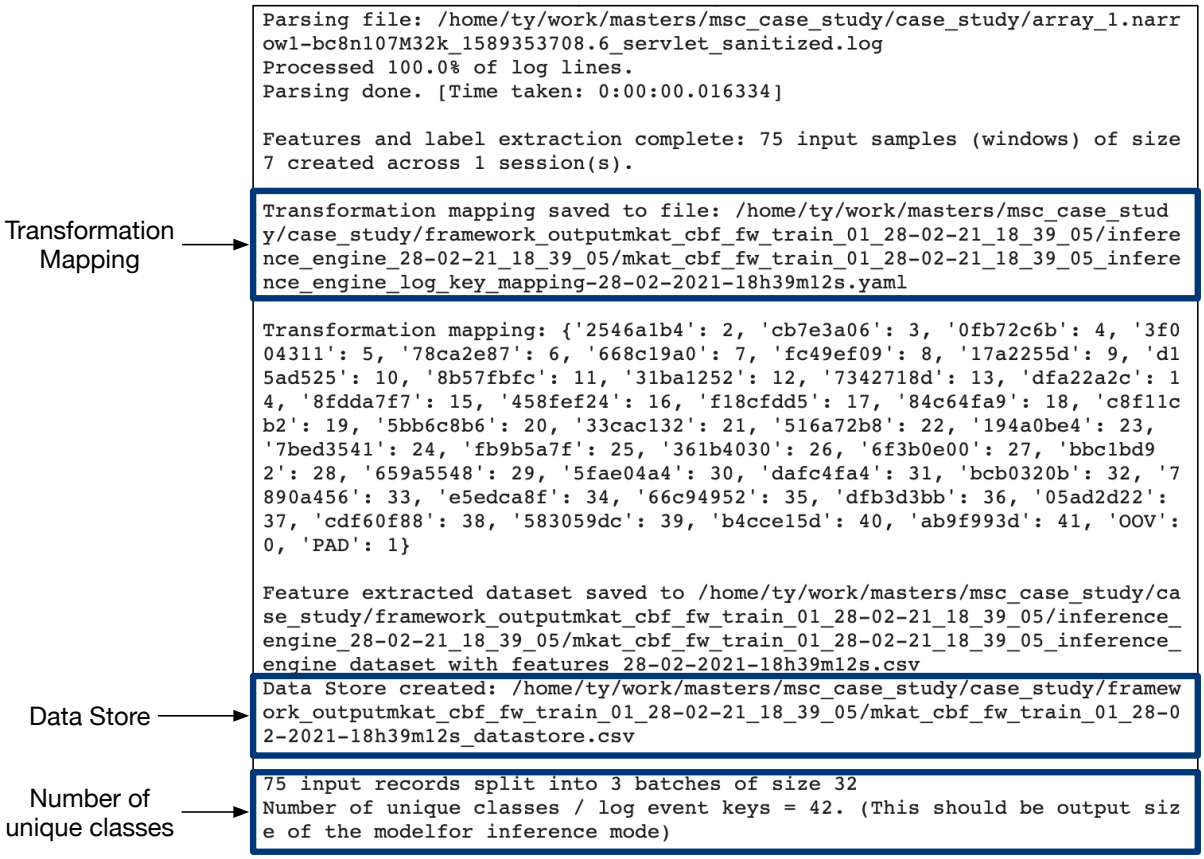


Figure 8.3: Extract from the ALFAF's operation in *training* mode. The outputs generated after feature extraction are shown

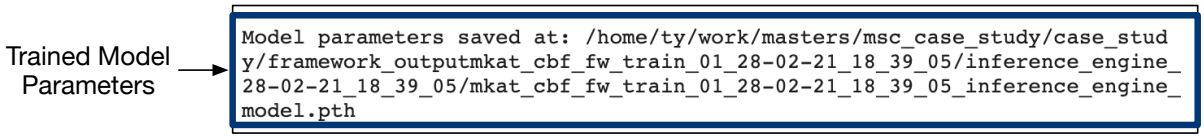


Figure 8.4: Extract from the ALFAF's operation in *training* mode. The trained model parameters are saved to a file

8.1.3 Using the Framework to Detect Anomalies

After running the ALFAF in *training* mode on the CBF log files, the required data stores and models, for operating the ALFAF in *inference* mode to detect potential anomalous lines in new log files, were generated. This subsection explains how the ALFAF was configured to run in *inference* mode, using these data stores and models, and also describes the outputs generated by the ALFAF when operated in this mode.

Data Miner Configuration

When running the ALFAF in *inference* mode, no additional preparation for the Data Miner was required. The same configuration file for the Data Miner, as used in *training* mode, was used again for *inference* mode.

Inference Engine Configuration

The Inference Engine requires additional configuration information when the ALFAF is to be operated in *inference* mode. The information required is as follows: a transformation mapping, the output size of the LSTM-anomaly detection model, a Log Event Key Data Store and the learned model parameters. This information is generated by the ALFAF when operating in *training* mode as shown in Figures 8.3 and 8.4.

The configuration file of the Inference Engine was updated with this new information and the Inference Engine mode was set for *inference* operation. All other configuration file parameters, except those mentioned here, were configured to the same values as in *training* mode.

Outputs of Inference Mode

With *inference* mode-ready configuration files for both the Data Miner and Inference Engine subcomponents, the ALFAF was then run in *inference* mode to attempt to identify potential anomalous log messages within CBF log files. An extract from the output of the ALFAF's operation in *inference* mode is shown in Figure 8.5.

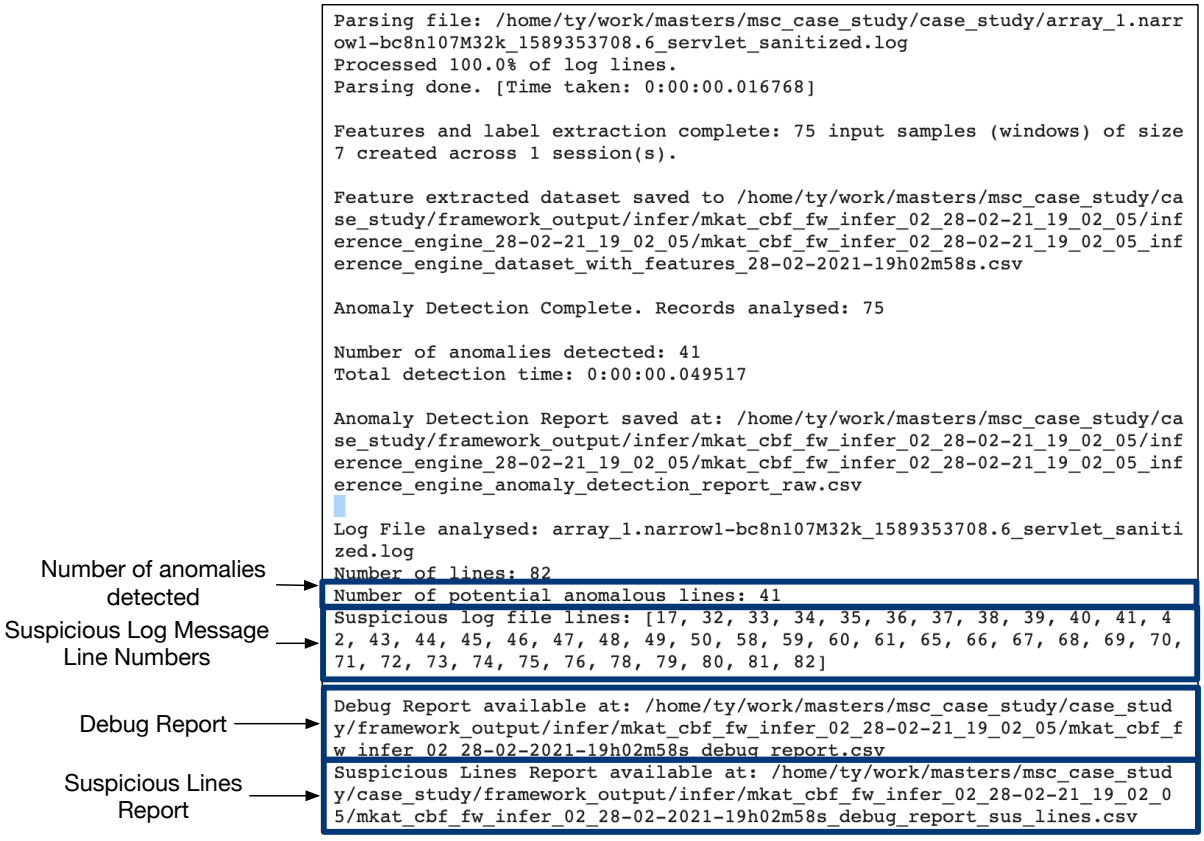


Figure 8.5: Extract from the ALFAF's operation in *inference* mode. Shows outputs generated after performing Anomaly Detection

Figure 8.5 illustrates the outputs of the ALFAF when operating in *inference* mode to perform anomaly detection on a given log file. The ALFAF generates two main outputs. Firstly, the *Debug Report* which contains the entire log file, in structured, parsed format with potential anomalous log messages identified, and secondly, the *Suspicious Lines Report* that only contains

the log messages that were identified as potential anomalies. These log messages are identified by their line numbers in the original log file and can be cross-referenced with the *Debug Report* during in depth debugging activities.

The ALFAF also outputs the number of anomalies detected and a list of line numbers identifying potential anomalous log messages in the analysed log file.

8.1.4 Subjective Evaluation

This subsection provides an evaluation of the performance and usefulness of the ALFAF in the context of the CBF log files, highlights key operational aspects and concludes the case study. Without an available ground truth dataset, with labelled anomalies or failures, a subjective and perceptive evaluation of the ALFAF's performance on the CBF log files is performed.

Data Miner and Log Parsing Evaluation

The Data Miner subcomponent performed quite well on the CBF log files in terms of accurately parsing the log files into a structured dataset. All of the supported log parsing algorithms were able to achieve very high parsing accuracy once tuned toward the CBF log files. Given that the nature of the CBF log files is such that certain events do not occur frequently, the provision of regular expressions for matching common runtime variable parts of log events is imperative to achieving high parsing accuracy. Manually extracting these regular expressions was simple given the relatively low number of unique log messages in the CBF logs.

It was found that certain CBF log files contained log messages that did not conform to the expected log message format. These messages were found to be relating to Git version information of various software packages. The implementation of logging these events was found to not use the same logging infrastructure as the rest of the log messages. For these log messages, the Data Miner was unable to correctly parse the log messages into a structured format. This challenge is more attributed to the presence of these unexpected log formats in the log files than the capability of the Data Miner. Since these log messages did not contribute to the CBF initialisation sequence, they were removed from the log files when encountered. This behaviour highlights the importance of a consistent pseudo-structure throughout the log file in which the actual log message content, the free-form text component describing the event, is preceded by the same set of metadata.

Overall, the Data Miner was able to effectively implement automated log parsing and was able to perfectly identify and extract log message templates from the CBF log files. The automated log parsing was also much more efficient when compared to the rigorous process followed to generate the ground truth by manually inspecting log files and creating regular expressions for extracting event templates.

Inference Engine and Anomaly Detection Evaluation

The performance of the Inference Engine on the CBF log files was mixed. In terms of model accuracy and being able to correctly predict the next log event key for a given input sequence,

the Inference Engine performed well. In terms of anomaly detection performance however, the framework experienced a number of challenges that impacted its ability to accurately detect and flag anomalous log messages.

Firstly, the CBF log file dataset used in this case study was not consistent. At the time of writing this dissertation, the MeerKAT CBF subsystem is simultaneously deployed for use and still under active development. New features, bug fixes and code refactors are regularly rolled out to the deployed system and there are often numerous differences between the events that are logged between releases. This inconsistency results in inaccurate system modelling and causes new log events to be erroneously classified as anomalies.

As discussed in Section 8.1.1 a new log file is generated for each instance of the CBF and each instance can be configured to run in a different mode of operation. This results in an additional layer of inconsistency as each possible CBF mode of operation may be considered to be an entirely unique system with its own sequence of events. This makes it challenging to train a single Inference Engine that is capable of accurately detecting anomalies across all CBF log files.

CBF log files generated for larger instruments were found to have a large number of repeating log messages. These log messages described the configuration status of the various processing nodes during CBF initialisation. This repetition often resulted in input sequences, for the LSTM-model, that consisted of the same log message. This created an inaccurate model of system behaviour that did not translate well to CBF log files generated for smaller instruments.

Finally, the CBF log files used in this case study were not representative of normal system behaviour. The log files contained a mix of normal system events alongside system errors and failures. This impacted the accuracy of the trained model as actual failures were often regarded as normal log messages. Due to the fact that the MeerKAT CBF is still under development, log files that are representative of the system's complete, normal operational behaviour are not presently available.

In this case study then, the poor performance of the Inference Engine on the CBF log files was found to be mainly caused by inconsistencies in the log files which reduced the quality of the dataset available for training.

Overall ALFAF Evaluation

The overall performance of the ALFAF on the CBF log file depends on how well both the Data Miner and Inference Engine subcomponents perform.

As discussed, the Data Miner is able to perfectly parse the CBF log files into a structured dataset and is able to accurately identify and extract most runtime variable parameters from log messages even when preprocessing regular expressions are not provided. When these regular expressions are provided, the extracted log message templates more closely matched the ground truth event templates. These findings indicate that the CBF log files can be automatically parsed by a log parsing algorithm to generate a dataset that is more suitable for later log file analysis activities.

The Inference Engine was found to perform less favourably on the CBF logs. The Inference Engine had no difficulty in terms of modelling the sequence of log events, but failed to perform anomaly detection reliably. Deep inspection of the CBF log files revealed shortcomings in the content, structure and generation of the log files that attributed to poor anomaly detection performance.

The operation of the ALFAF was demonstrated to be simple and efficient with only a few key inputs required for both the training and inference modes of operation. The ALFAF was able to quickly parse log files containing thousands of log messages and was able to analyse these log files quickly. The outputs generated after inference, the Debug Report and Suspicious Lines Report, could potentially aid developers in tracking down the exact point of failure during runtime. Conceptually then, assuming the shortcomings of the CBF log file contents can be addressed and overcome, the Automated Log File Analysis Framework could prove useful to developers and system operators tasked with debugging system failures.

The insights gained from this case study are used to address the research questions in Chapter 9.

8.2 Analysis

Using the results from performance evaluation on the two ALFAF subcomponents and the findings presented in Chapter 7, as well as that of the case study detailed in Section 8.1, this section analyses and discusses the performance, operation and design of the ALFAF and its subcomponents. This section proceeds as follows: Section 8.2.1 presents a discussion centred around the Data Miner subcomponent while Section 8.2.2 analyses the Inference Engine. Section 8.2.3 closes off the analysis by considering the ALFAF in its entirety and discusses its viability as an end-to-end, automated log file analysis pipeline for debugging system failures.

8.2.1 Data Miner Analysis

The Data Miner subcomponent of the ALFAF is analysed in terms of its log parsing performance, tuning, efficiency, and its robustness and generalisability. Additionally, attributes of the log files under test that may influence the performance and efficiency of the Data Miner are also considered.

Log Parsing Performance

The results of verification and performance evaluation presented in Section 7.2.3 show that all log parsing algorithms supported by the Data Miner are able to achieve a high parsing accuracy on the HDFS log files. Similarly, the findings of the case study in Section 8.1 also indicated that the log parsing algorithms were able to achieve high parsing accuracy on the CBF log files. These results indicate the viability of using data-driven, automated techniques and algorithms for parsing unstructured log files into a structured dataset in which log messages are represented by log event templates and log event keys. Further inspection of the generated parsed datasets show that the log parsing algorithms perform largely similarly in terms of

extracting event templates from the log messages. At most, the number of event templates extracted by the various algorithms differs by two for the HDFS logs and by ten for the CBF logs. The differences were found to be largely caused by the way in which the algorithms handle log messages that do not appear as often. it should be noted that the log parsing algorithms were only able to perform optimally after being specifically tuned toward log files for a given system.

Tuning

As mentioned, the log parsing algorithms were able to achieve high parsing accuracy once tuned toward a particular log message structure. The tuning of the log parsing algorithms involved searching through all possible combinations of tunable parameters for each algorithm to determine which combination resulted in the highest parsing accuracy. The results of log parser tuning, presented in Table 7.2, illustrate just how much impact the tunable parameters can have on the performance of each log parsing algorithm. Tuning the log parsing algorithms toward the log files under consideration is imperative for achieving the best results. The developed log parsing tuning tool, detailed in Chapter 4.7, proved to be useful in enabling and streamlining the tuning of the various log parsing algorithms. The only caveat around log parser algorithm tuning is that a ground truth dataset containing extracted event templates is required. Such a dataset is not often available for most systems. As a means around this challenge, it is recommended that the log parsing algorithm be applied to a given log file and that the generated event templates be used as the basis for generating the exact event templates. This process, while still requiring manual involvement, significantly reduces the number of log messages that need to be considered to extract all event templates. Two log parsing algorithms, *IPLoM* and *LogMine*, took considerably longer to tune compared to the other algorithms. This was found to be due to the much larger configurable parameter search space for *IPLoM* and the general longer parsing time of *LogMine*.

Efficiency

The results presented in Table 7.5 indicate that, in most cases, the log parsing algorithms are able to parse log files quickly. Excluding the results from the *LogMine* algorithm, as its parsing time was significantly higher comparatively, the average parsing time for a log file containing 10 000 log messages was 0.8 seconds. This is significantly faster than manual extraction and parsing. While the parsing time of each algorithm does increase as the size of the log files grow, the impact of this must be considered in context. During the training of the ALFAF, it is expected that larger datasets would be used, but during training, parsing time is inconsequential since the training of the Inference Engine is expected to contribute the majority toward the total training time. When the ALFAF is operated in inference mode, parsing time is of more relevance as the outputs of the ALFAF as desired as quickly as possible to assist with debugging and reduce system downtime. When performing inference, however, it is assumed that smaller log files will be staged for analysis in most cases, so any of the available log parsing algorithms would perform sufficiently in this regard.

Robustness and Generalisability

Another attribute that motivates toward the tuning of log parsing algorithms is the robustness and generalisability across log files of different systems. Table 7.6 shows how the performance of the log parsing algorithms, tuned toward the HDFS dataset, perform on log files from other systems. The results vary significantly in most cases. The structure of the log messages themselves are identified to be the cause of this variation. When the structure of the log messages vary significantly, the parsing algorithms require different configurations to be able to extract event templates accurately. In the case study conducted, the different CBF log files, generated by different modes of operation, were all parsed using the same log parsing algorithm configured with the same parameters. In this case, although the log files were different, the messages were of similar enough structure and the log parsing algorithm was still able to effectively extract the event templates.

Dataset Considerations

During tuning, evaluation and the conducting of the case study, an observation regarding the dataset itself and how it affects log parsing were made: as the size of the log files for a particular system grew significantly, the number of unique event templates extracted only increased marginally. This indicates that the size of the log files is not as important as the content of the log files. For optimal log parsing performance, the log files must contain a fully representative set of log messages that pertain to all possible event templates that occur during normal system operation.

8.2.2 Inference Engine Analysis

The Inference Engine subcomponent of the ALFAF is analysed in terms of its anomaly detection performance. Following this, considerations around the dataset used for training and inference are discussed.

Anomaly Detection Performance

The anomaly detection performance of the Inference Engine was found to be mixed as evidenced by the results in Table 7.9. Although a model accuracy of 94% was achieved, the Inference Engine struggled to perform well in the task of identifying anomalies from the given log files. Depending on the number of candidate log keys considered, an many as 435 anomalous events are classified as being normal and as many as 1754 normal log events were incorrectly classified as being anomalous. As the number of candidate log keys increases, the number of false positives (normal events incorrectly classified as anomalous) reduces to 0 and the Inference Engine is able to avoid misclassifying normal events as anomalies. Regardless of the number of candidate keys considered, the Inference Engine is unable to achieve a high recall score, higher than 14%, and as a result many of the anomalous events are not detected.

The high model accuracy indicates that the Inference Engine is able to correctly predict the the next likeliest log event for a given input sequence of events for the majority of input sequences.

The poor anomaly detection performance then is deduced to be attributed to the quality of the dataset used. The performance of the Inference Engine was evaluated using the HDFS ground truth dataset in which anomalies are labelled per session. Further inspection into the Anomaly Detection Report generated by the Inference Engine revealed that performance was being impacted by anomalous log entries that were absent from the dataset. The case study, as discussed in Section 8.1.3, also revealed that poor Inference Engine performance was attributed to the available dataset.

Despite the poor anomaly detection performance, the results also revealed the impact of the number of candidate log keys to be considered when making the decision about whether a particular log event is anomalous or not. A given sequence of log events may have multiple valid next log events that all describe normal system behaviour. The number of candidate log keys must be selected to take this into account. Setting the number of candidates too low results in too many false positives and a very low precision score. Setting it too high can potentially reduce the number of true positives identified, resulting in a lower recall score. The number of candidates must therefore be tuned to find a balance between precision and recall. Another factor that influences the setting of this parameter is the total number of unique log keys in a given system. The number of candidate keys must at all times be less than the total number of unique log keys otherwise the Inference Engine will be unable to detect any anomalous events.

Dataset Considerations

As mentioned in the analysis of the anomaly detection performance and in the case study, the poor performance of the Inference Engine was found to be attributed to the quality of the datasets used. This section discusses how certain characteristics of the log files can affect the performance of the Inference Engine and what characteristics are required for optimal performance.

The Inference Engine's anomaly detection model is based on modelling system behaviour and flagging unexpected event occurrences that deviate from the expected behaviour as anomalies. If the Inference Engine is trained on log file data that contains anomalous events, then the Inference Engine will regard the occurrence of these events as normal in later analysis activities. A separate test in which the performance of the Inference Engine was evaluated after being trained using mixed normal and anomalous log message and using only normal log messages, was conducted. The results are shown in Table 8.3.

As shown in Table 8.3, when trained using normal events only, the Inference Engine is able to detect 5 times more anomalies than if it were trained using mixed events. Therefore, for optimal anomaly detection performance, the Inference Engine must be trained to model system behaviour using only normal or expected sequences of events.

The dataset used for training the Inference Engine must also be representative of the log files that will be analysed by the Inference Engine at a later stage. As shown through the case study, even different modes of operation of the same system can result in vastly different sequences of log events. These differences have been shown to have a significant impact on the performance

Table 8.3: *Anomaly Detection performance of the Inference Engine trained on a mixed events dataset and a normal events only dataset*

	Training Data	
	Mixed Events	**Normal Events**
Model Accuracy	0.934	0.939
True Positives	7	35
True Negatives	12644	12638
False Positives	0	6
False Negatives	584	556
Precision	1	0.854
Recall	0.012	0.059
F1 Measure	0.023	0.111

of the Inference Engine.

Finally, the dataset used for training must also be *complete*. For the Inference Engine to accurately model system behaviour, the available log files must describe the *entire* normal operation of the system. If the log files only describe a subset of the system's operation, normal events will be misclassified leading to a higher false alarm rate.

8.2.3 Overall Framework Analysis

With the analysis of the Data Miner and Inference Engine subcomponents complete, this section concludes the analysis by considering the entire Automated Log File Analysis Framework and discussing how it performs as a robust, end-to-end pipeline for automating log file analysis.

The case study, presented in Section 8.1, demonstrated the operation of the ALFAF and validated the design of the overall framework. It was shown that the ALFAF is capable of ingesting log files generated by a system and is then able to, through the Data Miner and Inference Engine subcomponents, perform log parsing and anomaly detection-based log file analysis on these log files.

The usage of the ALFAF is simple and streamlined, and only requires two configuration files. The only system-specific information that is required is the log message format and the optional list of regular expressions that match variable parameters in the log messages.

The ALFAF performs log parsing well, provided that the log parsing algorithms can be tuned toward the system under consideration. Anomaly Detection performance is currently mixed at best, and is largely dependent on characteristics of the data available for training. It was, however, shown that the ALFAF is capable of training an LSTM model for predicting expected log events that achieves a high model accuracy.

In terms of robustness and generalisability, as is the case with most data-driven approaches, tuning and training are highly recommended in order to achieve the optimal results. Tuning of the Data Miner does require a ground truth dataset containing log event templates, but the creation of this can be supplemented through use of the Data Miner itself. The developed log parser tuning tool proved to be invaluable in finding the optimal configuration for each of the

log parsing algorithms.

While the Inference Engine does require tuning and training for each system that is to be considered, it does not require a labelled dataset as features and target labels are extracted automatically, by the Inference Engine, from the available data. This dataset must, however, be representative of the complete, normal expected system behaviour.

In terms of providing a robust, end-to-end automated log file analysis framework, the ALFAF meets this objective. While anomaly detection performance does need to be improved before the framework is viable and useful, it does provide a robust and easy-to-use pipeline for implementing, and automating, both log parsing and log file analysis processes to enable more efficient system debugging using log files.

8.3 Analysis Conclusion

This chapter provided a detailed analysis of the design, performance, and operation of the Automated Log File Analysis Framework using the results presented in Chapter 7 and the case study conducted in Section 8.1 as a basis for discussion.

Section 8.1 details the case study conducted in which the ALFAF was used on log files generated by the MeerKAT Correlator Beamformer. This case study validated the design and operation of the framework and also highlighted challenges that real-world systems could pose.

Section 8.2 then considered the results of Chapter 7 and the case study, and discussed the ALFAF in the more general sense. The Data Miner and Inference Engine subcomponents were analysed individually before the framework as a whole was considered.

The analysis and discussion presented in this chapter is next used to finally address the research questions and conclude the dissertation in Chapter 9.

9

Conclusions and Recommendations

This final chapter of the dissertation reflects on the work carried out during this research project and draws insights from previous chapters to address the research questions put forward in Section 1.2.

Chapter 1 presented the main objective of the project and put forward six research questions to drive the development of the dissertation toward meeting the stated objective. Chapter 2 then described the methodology followed during this research project and presented the three main phases of this project: the Concept Exploration phase, the System Development phase and finally, the System Verification and Analysis phase.

Chapter 3, the literature review, unpacked the challenges associated with log file-based system debugging and investigated recent data-driven approaches that have been proposed to address these challenges. It also explored the inner workings of the MeerKAT Correlator Beamformer. The research conducted in Chapter 3 informed the design and development of the Automated Log File Analysis Framework (ALFAF). The two major subcomponents of the ALFAF, the Data Miner and the Inference Engine, were designed and developed in detail in Chapters 4 and 5 respectively. Chapter 6 described the integration of the subcomponents to implement the end-to-end processing pipeline and concluded the design of the ALFAF.

Once all the necessary development was completed, Chapter 7 considered the verification, evaluation and tuning of the ALFAF. The designs of the subcomponents and the framework were verified, and the performance of both the Data Miner and Inference Engine was evaluated. Chapter 8 presented a detailed analysis of the design, performance and operation of the ALFAF and its subcomponents. This included a case study in which the ALFAF was put to work on log files generated by the MeerKAT Correlator Beamformer system. The results of this case study, and the results of performance testing presented in Chapter 7, were then analysed to determine how well the ALFAF performed its intended function.

The following sections in this chapter provide answers to the research questions (Section 9.1), reflect on the project objectives (Section 9.2) and provide recommendations for future work in terms of continuing this research and improving the performance of the ALFAF on the MeerKAT Correlator Beamformer (Section 9.3).

9.1 Research Question Answers

This section uses the insights gained throughout this research project to address the six research questions put forward in Section 1.2 to drive the development of the dissertation toward meeting its primary objective.

9.1.1 Research Question 1

The first research question was presented as:

Research Question 1: *What data-driven log parsing techniques can be used to build a Data Miner capable of parsing raw, unstructured log files into a structured dataset? How well do these techniques perform?*

Research studied in Section 3.2.2 revealed various data-driven techniques for implementing automated log parsing. The most recent, state-of-the-art techniques were examined in greater detail in Section 4.5.1. Through this examination, the most promising log parsing techniques were identified to be *AEL* [48], [49], *Drain* [44], *IPLoM* [43], [42], *LogMine* [45], *LenMa* [80] and *Spell* [50]. Previous research has shown that these log parsing techniques are able to achieve a high average parsing accuracy across different datasets. The Data Miner was designed, as described in Chapter 4, to support all six of these log parsing algorithms. The Data Miner ingests an unstructured log file and, using one of the supported log parsing algorithms, parses the log file into a structured dataset in which each log message is represented by a log event template, corresponding to the static part of the log message, a log event key and a list of parameters corresponding to the variable parts of the log message.

Evaluating the performance of the Data Miner in Section 7.2 revealed three key insights:

- the performance of the log parsing algorithms was sensitive to the tunable parameters used to configure each algorithm,
- all the supported log parsing algorithms are able to achieve a high parsing accuracy when tuned, in terms of identifying the optimal set of tunable parameters, toward a particular dataset,
- automated log parsing, using an algorithm, was much faster than manually generating regular expressions for matching log messages, and
- none of the log parsing algorithms generalised well across different log files without prior tuning specifically on the log files in question.

The conclusion, then, is that the automated log parsing techniques are able to perform well, in terms of parsing accuracy, only when they are tuned toward log files of a particular system. The log parsing algorithm tuning tool developed in Section 4.7 was demonstrated to be invaluable for tuning the Data Miner for optimal performance.

9.1.2 Research Question 2

The second research question was presented as:

Research Question 2: *Which machine learning and/or deep learning technique is best suited for an Inference Engine to perform failure detection using the log files generated by a system? How well does the Inference Engine perform?*

Initial research conducted in Section 3.2.4 considered various machine learning and deep learning based techniques for automating log file analysis. More specifically, log file analysis modelled as an anomaly detection problem was considered. Previous research pointed toward LSTM neural networks being the most promising technique for two main reasons. Firstly, log messages in a log file may be considered to be a sequence of events, with various dependencies between the elements in the sequence. LSTM networks are capable of modelling long-range dependencies between elements in a sequence. The second reason was the fact that since LSTM networks can accept sequences as an input, feature engineering could be used to create a dataset of input features and target labels without the need for an explicitly labelled dataset.

The Inference Engine was designed, as described in Chapter 5, to use an LSTM neural network to model a system's execution path using the sequence of events that can be extracted from log files. This model is then used to identify log messages that deviate from this expected path. The Inference Engine considers a sequence of input log events and predicts the likeliest candidates for the next log event. If the actual log event is not among the likeliest candidates, then an anomaly is flagged for that particular log message.

The performance of the Inference Engine was evaluated in Section 7.3 and was considered from two aspects: model accuracy, describing how well the Inference Engine was able to model the system's execution path, and anomaly detection performance that considers how accurately anomalies are detected. The performance evaluation of the Inference Engine revealed the following:

- the Inference Engine is able to accurately model the system's execution path from a given log file, achieving a model accuracy as high as 94% in certain instances,
- the Inference Engine's anomaly detection performance is mixed at best. While the Inference Engine was able to detect anomalies, the number of false positives and false negatives was high and the Inference Engine was neither able to achieve good precision nor recall scores,
- the mixed anomaly detection performance was identified to be largely attributed to the structure and contents of the datasets available for training the Inference Engine, and
- tuning the number of likeliest candidate log events does impact the performance of the Inference Engine and there is a trade-off between precision and recall that must be made when tuning this parameter.

9.1.3 Research Question 3

The third research question was presented as:

Research Question 3: *How can the Inference Engine be tailored to generate assistive debugging information to aid in the debugging of detected failures? How can the Inference Engine be given a degree of interpretability?*

As discussed in Chapter 5, the Inference Engine was designed to detect anomalies by first modelling the system's expected execution path and then identifying events, described by log messages, that deviate from this expected path. When performing inference on a given log file, the Inference Engine generates an Anomaly Detection Report as detailed in Section 5.5. This report contains, for each log message, the input sequence of log event keys considered, the likeliest expected log keys for the given input sequence, the actual occurring log event key and a flag indicating whether or not the log message is anomalous. The contents of this report provide some insight into the anomaly decision criteria used by the Inference Engine, but the contents are still represented as numerical features at this stage. This Anomaly Detection Report is then used by the ALFAF to generate the Debug Report and Suspicious Lines Report. Generating these reports translates the content of the Anomaly Detection Report into a human-readable dataset consisting of the parsed log file and the Inference Engine's anomaly decision criteria. The Suspicious Lines Report, specifically, only contains log messages that have been flagged as being anomalous. These messages are identified by their line number in the original log file and may be cross-referenced with the Debug Report to analyse the events that occurred before and after the anomalous entry. The Debug and Suspicious Lines Reports, and the generation thereof, is detailed in Chapter 6.

The Debug and Suspicious Lines Reports do not only assist in failure debugging, but also provide clear insight into *why* the Inference Engine flagged any particular log message as an anomalous event.

9.1.4 Research Question 4

The fourth research question was presented as:

Research Question 4: *How can the Data Miner and Inference Engine be combined into an automated, end-to-end, log file analysis-based failure detection framework that is capable of ingesting raw system log files and outputting assistive debugging information?*

The Data Miner and Inference Engine were designed and developed as stand-alone components in Chapters 4 and 5 respectively. In order to combine them into an end-to-end framework, an interface between the two components was required. This interface was formally defined in Section 6.2 and was considered during the design of each component. This interface is a data interface and defines the structure of the data that is to be passed from the Data Miner, after performing log parsing, to the Inference Engine for performing further log file analysis. The most important data field of this data structure was the log event key, as the Inference Engine

used the sequence of log event keys to model the system's execution path.

The Data Miner and Inference Engine components were combined into an end-to-end framework through a custom Python class that instantiates instances of each component. This framework was designed to have external interfaces to the system under test and the end-user as described in Section 6.2. The framework is capable of receiving raw log files, generated by the system under test, and outputs assistive debugging information to the end-user in the form of the Debug and Suspicious Lines Reports.

9.1.5 Research Question 5

The fifth research question was presented as:

Research Question 5: *What requirements or guidelines can be imposed on the generation of system log files to make them more amenable to automated log file analysis using the developed framework?*

The evaluation of the Data Miner and Inference Engine in Chapter 7, and the findings of the case study in Chapter 8, revealed insights that were used to form requirements to guide the generation of log files, to be used for ALFAF training, to improve the overall performance of the ALFAF. These requirements are listed below.

The log files used to train the ALFAF shall:

- be generated by the same system on which the ALFAF is to perform inference,
- be representative of the system's complete, normal execution behaviour,
- not contain any anomalous log events, and
- only consist of log messages that conform to the same log message format with a clearly discernable log message content component.

The accuracy of the log parsing performed by the Data Miner may be improved through the following recommendations:

- regular expressions matching variable components of log messages should be provided where possible, and
- the Data Miner should be tuned toward a particular dataset. This requires a ground truth of extracted log event templates which may be generated semi-manually using the Data Miner.

9.1.6 Research Question 6

The sixth and final research question was presented as:

Research Question 6: *What is the predicted impact of this framework on the process of debugging system failures, system uptime, operation, reliability and usability in the context of*

the MeerKAT Correlator Beamformer? How can the performance of this framework be tested at scale?

Section 8.1 detailed a case study that was conducted in which the ALFAF was applied to log files generated by the MeerKAT Correlator Beamformer. This case study sought to understand how the ALFAF performs on a real-world system and to determine the usefulness of ALFAF in terms of debugging system failures occurring within the CBF.

The results of the case study indicate that while the ALFAF is able to accurately model the system execution path from the provided log files, the anomaly detection performance is lacking. This was discovered to be as a result of the log files available for training. These log files did not meet all of the criteria as specified in Section 9.1.5. As a result, a number of recommendations for improving the performance of the ALFAF, in the context of the CBF system, is put forward in Section 9.3.1.

Given that the CBF log files were parsed accurately by the Data Miner, and taking into consideration that the Inference Engine was able to accurately model the execution path present in the log files, there is potential for the ALFAF to be useful in terms of accurately identifying failures, provided that suitable training data, i.e. log files, is available. The general operation and performance of the ALFAF indicate that it is capable of processing a large volume of log messages in a very short time. This efficient processing, coupled with improved anomaly detection performance, can make the ALFAF an indispensable tool for identifying and debugging system failures. The generated Debug Report and Suspicious Lines Report will allow operators and engineers to more quickly identify erroneous execution events as they will be able to narrow their search space of the log files significantly.

Before testing the performance of the ALFAF at scale on the CBF system, the recommendations, regarding the CBF log files, made in Section 9.3.1 must be considered. Once the challenges with the log files have been overcome, the ALFAF can be re-trained, and be integrated into the CBF system's software stack to continuously and automatically analyse log files as they are generated. This will result in the generation of multiple Debug Reports, each pertaining to a different observation, and engineers and operators will be able to review the contents of the report and assess the performance of the ALFAF over a period of time.

9.2 Reflection on Project Objectives

The main objective of this research project, as put forward in Chapter 1, was the design and development of a robust, end-to-end automated log file analysis framework and the evaluation of this framework on the MeerKAT Correlator Beamformer system.

While a robust, end-to-end automated log file analysis framework was designed and developed in the form of the ALFAF, the case study conducted in Chapter 8 indicates that there are additional considerations required, mainly regarding the CBF log files, before the ALFAF can be deemed to be useful on the CBF system in terms of performing automated log file analysis. The ALFAF was able to accurately parse the CBF log files and model the system's execution

path, but was limited in its ability to accurately detect and identify anomalous events. The structure and content of the CBF log files was found to be the factor that was hindering the anomaly detection performance of the ALFAF. Considerations for improving the CBF log files in this regard are put forward as recommendations in Section 9.3.1.

The ALFAF design is modular, robust and tunable, and as such, can be used to perform automated log file analysis on log files generated by any system. The ALFAF is released as an open-source tool that may be extended with additional features, and recommendations for future enhancements are provided in Section 9.3.2. The source code for the developed ALFAF is available at: `https://github.com/tyronevb/alfaf`

It can be concluded, then, that the research project was successful in achieving its main objective.

9.3 Recommendations for Future Work

The recommendations presented here include considerations to be made to improve the performance and usefulness of the ALFAF on the CBF in Section 9.3.1, as well as additional avenues of research to further the design of the ALFAF in Section 9.3.2.

9.3.1 Improving ALFAF Performance on CBF

The mixed anomaly detection performance of the ALFAF on the CBF system was attributed to the structure, format and content of the CBF log files available. The following recommendations should be considered to make the CBF log files more amenable to automated log file analysis using the ALFAF:

1. The CBF log files contain numerous duplicates of events describing node configuration. For larger instruments, this can lead to poor feature generation for the Inference Engine. The CBF should only log one single event, to the main log file, if all nodes are successfully configured. The individual node events may be logged separately to a different file.
2. The CBF log files currently contain a mix of normal and anomalous log events. Training the inference engine requires that the log files only contain normal log events. A log file containing only normal events should be generated through the use of a CBF simulator that runs the CBF software in a simulated environment to generate the sequence of events expected during normal system operation.
3. The CBF can operate in a number of different modes and configurations. While the structure of the log messages is similar, the overall content of the log files can differ greatly. When using the ALFAF on the CBF, a separate instance of the framework should be trained for each mode or configuration, and the corresponding framework should be used when performing inference on a given log file.
4. All log messages contained in the CBF log files should conform to the same log message format.

9.3.2 ALFAF Research Recommendations

The recommendations listed below describe potential areas of research for future work to improve upon the design of the ALFAF:

1. Evaluate the generalisability of the ALFAF by considering its performance and operation on log files generated by multiple different systems.
2. Investigate the addition of a feature for extracting log message templates from software source code to assist with ground truth generation for Data Miner tuning.
3. Add support for processing log messages in an online, streaming manner.
4. Investigate alternative log file analysis objectives, and their implementation in the Inference Engine, that can utilise the outputs from the Data Miner.
5. Package the ALFAF in an easily-deployable format such that it is system and environment agnostic e.g. by using containerisation.
6. Investigate options for incorporating user-supplied feedback, in terms of false positives, into the ALFAF for improved anomaly detection performance.
7. Add support for handling systems with multiple modes of operation. Multiple models would need to be generated and the ALFAF should be capable of using the correct model for a given log file.

Bibliography

[1] A. Mirgorodskiy, N. Maruyama, and B. Miller, "Problem Diagnosis in Large-Scale Computing Environments," *ACM/IEEE SC 2006 Conference (SC'06)*, pp. 11–11, 2006. [Online]. Available: http://ieeexplore.ieee.org/document/4090185/

[2] T. Yang and V. Agrawal, "Log File Anomaly Detection," 2017.

[3] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2007, pp. 575–584.

[4] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *ICML 2010 - Proceedings, 27th International Conference on Machine Learning*, 2010, pp. 37–44.

[5] "The project – SARAO." [Online]. Available: https://www.sarao.ac.za/about/the-project/

[6] S. Dennehy, "M1200-0000-003 Correlator-Beamformer Design Document," South African Radio Astronomy Observatory, Tech. Rep., 2015.

[7] "GitHub - logpai/logparser: A toolkit for automated log parsing [ICSE'19, TDSC'18, DSN'16]." [Online]. Available: https://github.com/logpai/logparser

[8] C. Haskins, K. Forsberg, M. Krueger, D. Walden, and R. D. Hamelin, *Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities; INCOSE-TP-2003-002-03.2.2*, 4th ed. Hoboken, NJ: John Wiley and Sons, Inc, 2011. [Online]. Available: http://www.incose.org/ProductsPubs/products/sehandbook.aspx

[9] E. Fong, "Introduction to systems engineering," pp. 23–50, 2016. [Online]. Available: https://www.sebokwiki.org/wiki/Introduction{_}to{_}Systems{_}Engineering

[10] A. Sparrius, *Acquisition Mangement*, 2015.

[11] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, and C. He, "Latent error prediction and fault localization for microservice applications by learning from system trace logs," in *ESEC/FSE 2019 - Proceedings of the 2019 27th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, vol. 19, 2019, pp. 683–694. [Online]. Available: https://doi.org/10.1145/3338906.3338961

[12] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer, "Failure diagnosis using decision trees," in *Proceedings - International Conference on Autonomic Computing*, 2004, pp. 36–43. [Online]. Available: https://people.eecs.berkeley.edu/{~}brewer/papers/icac2004{_}chen{_}diagnosis.pdf

[13] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pp. 595–604, 2002. [Online]. Available: http://radlab.cs.berkeley.edu/people/fox/static/pubs/pdf/c21.pdf

[14] J. Stearley, "Towards informatic analysis of syslogs," in *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, 2004, pp. 309–318. [Online]. Available: http://www.loganalysis.com

[15] L. Mariani and F. Pastore, "Automated Identification of Failure Causes in System Logs Mariani-LogFile-ISSRE-2008.pdf."

[16] S. Xu and V. Rajlich, "Cognitive process during program debugging," *Proceedings of the Third IEEE International Conference on Cognitive Informatics, 2004.*, pp. 176–182, 2004. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1327473{%}5Cnhttp://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1327473

[17] A. Babenko, L. Mariani, and F. Pastore, "Ava," *Proceedings of the eighteenth international symposium on Software testing and analysis - ISSTA '09*, p. 237, 2009. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1572272.1572300

[18] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *Proceedings of the 3rd IEEE Workshop on IP Operations and Management, IPOM 2003*, 2003, pp. 119–126.

[19] D. Cotroneo, R. Pietrantuono, L. Mariani, and F. Pastore, "Investigation of Failure Causes in Workload-driven Reliability Testing," *Fourth International Workshop on Software Quality Assurance: In Conjunction with the 6th ESEC/FSE Joint Meeting*, pp. 78–85, 2007. [Online]. Available: http://doi.acm.org/10.1145/1295074.1295089

[20] J. Andrews and Y. Z. Y. Zhang, "Broad-spectrum studies of log file analysis," *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*, pp. 105–114, 2000.

[21] "grep(1) - Linux manual page." [Online]. Available: https://man7.org/linux/man-pages/man1/grep.1.html

[22] P. D. Admin, "Logging facility for Python." [Online]. Available: https://docs.python.org/3/library/logging.html

[23] P. Bodík, G. Friedman, L. Biewald, H. Levine, G. Candea, K. Patel, G. Tolle, J. Hui, A. Fox, M. I. Jordan, and D. Patterson, "Combining Visualization and Statistical Analysis to Improve Operator Confidence and Efficiency for Failure Detection and

Localization." [Online]. Available: https://www.cs.drexel.edu/{~}spiros/teaching/CS576/papers/Bodik{_}ICAC05.pdf

[24] LIGO, "What is an Interferometer? — LIGO Lab — Caltech." [Online]. Available: https://www.ligo.caltech.edu/page/what-is-interferometer

[25] "Radio Astronomy — COSMOS." [Online]. Available: https://astronomy.swin.edu.au/cosmos/R/Radio+Astronomy

[26] "KAT-7 – SARAO." [Online]. Available: https://www.sarao.ac.za/science/kat-7/

[27] "About MeerKAT – SARAO." [Online]. Available: https://www.sarao.ac.za/science/meerkat/about-meerkat/

[28] T. van Balla, "M1200-0000-0000 MeerKAT Correlator-Beamformer Requirement Specification Rev 4," South African Radio Astronomy Observatory, Tech. Rep., 2018.

[29] T. van Balla, "M1000-0001-002 MeerKAT Functional Interface Control Document - Correlator Beamformer to Control and Monitoring Rev 6," South African Radio Astronomy Observatory, Tech. Rep., 2019.

[30] "Correlators - AstroBaki." [Online]. Available: https://casper.ssl.berkeley.edu/astrobaki/index.php/Correlators

[31] T. van Balla, "M1200-0000 Correlator-Beamformer Block Diagram Revision 8," South African Radio Astronomy Observatory, Tech. Rep., 2020.

[32] "RADIO ASTRONOMY - Peralex - Dynamic Electronic Engineering." [Online]. Available: https://www.peralex.com/radio-astronomy/

[33] "SKARAB - Casper." [Online]. Available: https://casper.ssl.berkeley.edu/wiki/SKARAB

[34] S. Cross, R. Crida, T. Bennett, M. Welz, L. van der Heever, N. Marais, and A. Joubert, "Guidelines for Communication with Devices," South African Radio Astronomy Observatory, Tech. Rep., 2019.

[35] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "An evaluation study on log parsing and its use in log mining," in *Proceedings - 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016*, 2016, pp. 654–661. [Online]. Available: https://github.com/cuhk-cse/logparser

[36] R. Vaarandi, B. Blumbergs, and M. Kont, "An unsupervised framework for detecting anomalous messages from syslog log files," in *IEEE/IFIP Network Operations and Management Symposium: Cognitive Management in a Cyber World, NOMS 2018*, 2018, pp. 1–6.

[37] S. He, J. Zhu, P. He, and M. R. Lyu, "Experience Report: System Log Analysis for Anomaly Detection," in *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, 2016, pp. 207–218. [Online]. Available: https://github.com/cuhk-cse/logparser

[38] A. Géron, *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow (2019, O'reilly)*. O'Reilly Media.

[39] Q. Fu, J. G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *Proceedings - IEEE International Conference on Data Mining, ICDM*, 2009, pp. 149–158.

[40] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and Benchmarks for Automated Log Parsing," in *Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2019*, 2019, pp. 121–130. [Online]. Available: https://github.com/logpai/logparser

[41] R. Vaarandi, "SEC - A lightweight event correlation tool," *2002 IEEE Workshop on IP Operations and Management, IPOM 2002*, vol. 00, no. 4067, pp. 111–115, 2002.

[42] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "A lightweight algorithm for message type extraction in system application logs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 11, pp. 1921–1936, 2012.

[43] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "Clustering event logs using iterative partitioning," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2009, pp. 1255–1263.

[44] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An Online Log Parsing Approach with Fixed Depth Tree," in *Proceedings - 2017 IEEE 24th International Conference on Web Services, ICWS 2017*, 2017, pp. 33–40.

[45] H. Hamooni, B. Debnath, J. Xu, H. Zhang, G. Jiang, and A. Mueen, "LogMine: Fast pattern recognition for log analytics," in *International Conference on Information and Knowledge Management, Proceedings*, vol. 24-28-Octo, 2016, pp. 1573–1582. [Online]. Available: http://dx.doi.org/10.1145/2983323.2983358

[46] L. Tang, T. Li, and C. S. Perng, "LogSig: Generating system events from raw textual logs," in *International Conference on Information and Knowledge Management, Proceedings*, 2011, pp. 785–794.

[47] R. Vaarandi and M. Pihelgas, "LogCluster - A data clustering and pattern mining algorithm for event logs," in *Proceedings of the 11th International Conference on Network and Service Management, CNSM 2015*, 2015, pp. 1–7.

[48] M. J. Zhen, A. E. Hassan, P. Flora, and G. Hamann, "Abstracting execution logs to execution events for enterprise applications," in *Proceedings - International Conference on Quality Software*, 2008, pp. 181–186. [Online]. Available: https://www.researchgate.net/publication/4366728

[49] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "An automated approach for abstracting execution logs to execution events," in *Journal of Software Maintenance and Evolution*, vol. 20, no. 4, 2008, pp. 249–267. [Online]. Available: www.interscience.wiley.com

[50] M. Du and F. Li, "Spell: Streaming parsing of system event logs," in *Proceedings - IEEE International Conference on Data Mining, ICDM*, 2017, pp. 859–864.

[51] C. Monni and M. Pezze, "Energy-based anomaly detection a new perspective for predicting software failures," in *Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results, ICSE-NIER 2019*, 2019, pp. 69–72. [Online]. Available: https://webscope.sandbox.yahoo.com/

[52] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the ACM Conference on Computer and Communications Security*. Association for Computing Machinery, oct 2017, pp. 1285–1298.

[53] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun, and R. Zhou, "Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs," in *IJCAI International Joint Conference on Artificial Intelligence*, vol. 2019-Augus, 2019, pp. 4739–4745.

[54] K. Zhang, J. Xu, M. R. Min, G. Jiang, K. Pelechrinis, and H. Zhang, "Automated IT system failure prediction: A deep learning approach," in *Proceedings - 2016 IEEE International Conference on Big Data, Big Data 2016*, 2016, pp. 1291–1300. [Online]. Available: https://www.researchgate.net/publication/313456329

[55] M. Rhode, P. Burnap, and K. Jones, "Early-stage malware prediction using recurrent neural networks," *Computers and Security*, vol. 77, pp. 578–594, aug 2018.

[56] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, J. Chen, X. He, R. Yao, J. G. Lou, M. Chintalapati, F. Shen, and D.-M. Zhang, "Robust log-based anomaly detection on unstable log data," in *ESEC/FSE 2019 - Proceedings of the 2019 27th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, vol. 19, 2019, pp. 807–817. [Online]. Available: https://doi.org/10.1145/3338906.3338931

[57] G. Qi, L. Yao, and A. V. Uzunov, "Fault detection and localization in distributed systems using recurrent convolutional neural networks," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10604 LNAI. Springer Verlag, 2017, pp. 33–48.

[58] A. Brown, B. Hutchinson, A. Tuor, and N. Nichols, "Recurrent neural network attention mechanisms for interpretable system log anomaly detection," in *Proceedings of the 1st Workshop on Machine Learning for Computing Systems, MLCS 2018 - In conjunction with HPDC*. New York, New York, USA: Association for Computing Machinery, Inc, jun 2018, pp. 1–8. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3217871.3217872

[59] M. Astekin, H. Zengin, and H. Sözer, "Evaluation of Distributed Machine Learning Algorithms for Anomaly Detection from Large-Scale System Logs: A Case Study," in *Proceedings - 2018 IEEE International Conference on Big Data, Big Data 2018*. Institute of Electrical and Electronics Engineers Inc., jan 2019, pp. 2071–2077.

[60] T. F. Yen, A. Oprea, K. Onarlioglu, T. Leetham, W. Robertson, A. Juels, and E. Kirda, "Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks," in *ACM International Conference Proceeding Series*, 2013, pp. 199–208. [Online]. Available: http://dx.doi.org/10.1145/2523649.2523670

[61] M. Landauer, M. Wurzenberger, F. Skopik, G. Settanni, and P. Filzmoser, "Dynamic log file analysis: An unsupervised cluster evolution approach for anomaly detection," *Computers and Security*, vol. 79, pp. 94–116, 2018.

[62] Q. Lin, H. Zhang, J. G. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems," in *Proceedings - International Conference on Software Engineering*, 2016, pp. 102–111. [Online]. Available: http://dx.doi.org/10.1145/2889160.2889232

[63] Z. Liu, T. Qin, X. Guan, H. Jiang, and C. Wang, "An integrated method for anomaly detection from massive system logs," *IEEE Access*, vol. 6, pp. 30 602–30 611, 2018.

[64] C. Bertero, M. Roy, C. Sauvanaud, and G. Tredan, "Experience Report: Log Mining Using Natural Language Processing and Application to Anomaly Detection," in *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, vol. 2017-Octob, 2017, pp. 351–360. [Online]. Available: https://hal.laas.fr/hal-01576291

[65] J. G. Lou, Q. Fu, S. Yang, J. Li, and B. Wu, "Mining program workflow from interleaved traces," in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2010, pp. 613–622.

[66] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "SherLog: Error diagnosis by connecting clues from run-time logs," in *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 2010, pp. 143–154.

[67] A. B. Sharma, H. Chen, M. Ding, K. Yoshihira, and G. Jiang, "Fault detection and localization in distributed systems using invariant relationships," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2013.

[68] J. G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, "Mining invariants from console logs for system problem detection," in *Proceedings of the 2010 USENIX Annual Technical Conference, USENIX ATC 2010*, 2019, pp. 231–244.

[69] Python, "Welcome to Python.org," 2014. [Online]. Available: https://www.python.org/

[70] "NumPy." [Online]. Available: https://numpy.org/

[71] "pandas - Python Data Analysis Library." [Online]. Available: https://pandas.pydata.org/

[72] V. Jake, *Python Data Science Handbook*, 2019, vol. 53, no. 9.

[73] F. Pedregosa FABIANPEDREGOSA, V. Michel, O. Grisel OLIVIERGRISEL, M. Blondel, P. Prettenhofer, R. Weiss, J. Vanderplas, D. Cournapeau, F. Pedregosa, G. Varoquaux, A. Gramfort, B. Thirion, O. Grisel, V. Dubourg, A. Passos, M. Brucher, M. Perrot andÉdouardand, A. Duchesnay, and F. Duchesnay EDOUARDDUCHESNAY,

"Scikit-learn: Machine Learning in Python Gaël Varoquaux Bertrand Thirion Vincent Dubourg Alexandre Passos PEDREGOSA, VAROQUAUX, GRAMFORT ET AL. Matthieu Perrot," Tech. Rep., 2011. [Online]. Available: http://scikit-learn.sourceforge.net.

[74] "PyTorch." [Online]. Available: https://pytorch.org/

[75] Jupyter Team, "The Jupyter Notebook — Jupyter Notebook 6.0.0 documentation," 2019. [Online]. Available: https://jupyter-notebook.readthedocs.io/en/stable/examples/Notebook/WhatistheJupyterNotebook.html{#}Introduction{%}0Ahttps://jupyter-notebook.readthedocs.io/en/stable/examples/Notebook/WhatistheJupyterNotebook.html{%}0Ahttps://jupyter-notebook.readthedo

[76] "re — Regular expression operations — Python 3.8.6 documentation." [Online]. Available: https://docs.python.org/3/library/re.html

[77] S. Messaoudi, A. Panichella, D. Bianculli, L. Briand, and R. Sasnauskas, "A search-based approach for accurate identification of log message formats," in *Proceedings - International Conference on Software Engineering*. ACM, 2018, pp. 167–177. [Online]. Available: https://doi.org/10.1145/3196321.3196340

[78] M. Nagappan and M. A. Vouk, "Abstracting log lines to log event types for mining software system logs," in *Proceedings - International Conference on Software Engineering*, 2010, pp. 114–117.

[79] M. Mizutani, "Incremental mining of system log format," in *Proceedings - IEEE 10th International Conference on Services Computing, SCC 2013*, 2013, pp. 595–602.

[80] K. Shima, "Length Matters: Clustering System Log Messages using Length of Words," *arXiv preprint*, vol. arXiv:1611, 2016. [Online]. Available: http://www.wide.ad.jp/http://arxiv.org/abs/1611.03213

[81] "SciPy.org — SciPy.org." [Online]. Available: https://www.scipy.org/

[82] "The Official YAML Web Site." [Online]. Available: https://yaml.org/

[83] "sklearn.model_selection.ParameterGrid — scikit-learn 0.24.1 documentation." [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.model{_}selection.ParameterGrid.html

[84] "PyTorch: Tensors — PyTorch Tutorials 1.7.0 documentation." [Online]. Available: https://pytorch.org/tutorials/beginner/examples{_}tensor/two{_}layer{_}net{_}tensor.html

[85] "CrossEntropyLoss — PyTorch 1.8.0 documentation." [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html

[86] "HDFS Architecture Guide." [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs{_}design.html

[87] S. He, J. Zhu, P. He, and M. R. Lyu, "Loghub: A Large Collection of System Log Datasets towards Automated Log Analytics," *arXiv*, aug 2020. [Online]. Available: http://arxiv.org/abs/2008.06448