Enhancing Robustness of Software Log Parsers Using Large Language Models

Prabhu, Shashank S.

Award date:
2024

Awarding institution:
Tilburg University

Enhancing Robustness of Software Log Parsers Using Large Language Models

Bachelor's End Project Thesis

Candidate: Shashank Prabhu
1668471 / 2087567

Supervisor: Lina Ochoa Venegas

Technical University of Eindhoven and Tilburg University
BSc Data Science
2024

Contents

Abstract

As software systems advance and scale, generating more logs, there is a growing need for robust and versatile log parsers that work without any application-specific tuning. Existing parsers only focus on the robustness or versatility of applications but not both. This thesis aims to create a Large Language Model (LLM) based robust log parser and evaluate its performance and resilience on a custom-made dataset simulating software evolution. The parser uses a few shot-prompting approaches paired with a clustering sampler and the augmented log templates were created using LLMs to generate various "evolved" log datasets. The LLM parser experienced a relatively low 6% to 10% performance degradation on heavily augmented logs and a higher shot count resulted in a better-performing and consistent parser. 5-shot, the highest tested for this study, achieved a parsing accuracy of 0.920 and an F1 template accuracy of 0.857 on the original dataset, in the ballpark of state-of-the-art (SOTA) parsers. The findings suggest that LLMs can be leveraged to build robust and versatile parsers, requiring minimal to no modifications for different use cases. However, more research is needed to create augmented datasets with consistent magnitudes of change that simulate software evolution using LLMs.

1 Introduction

For every 58 lines of source code, there is one line of logging code [6]. Considering the original space shuttle had $\sim$ 400k lines of code and modern operating systems such as Windows 7 have over 40 million lines, the importance of logs is undeniable [19]. Log parsing and analysis is a central part of the software development workflow. Software logs can help monitor the state of software systems, be used to predict imminent failures and are part of many DevOps workflows. As systems become increasingly more complex and automated, more logs are generated.

Software is ever-evolving and as a result, so are the logs. One example is Service X, an internal development tool from Microsoft. Within service X changed logs account for over 30% of logs when compared to previous versions [6]. There are two main sources for these changes:

1. Evolution of logging statements as the software develops.

   This involves adding new logs, removing existing ones, or altering current logs by rephrasing the messages and adjusting variables.

2. Added noise introduced by processing the log data.

   During data collection and pre-processing logs can be duplicated, disordered or go missing, impacting the quality and reliability of logs for downstream tasks.

Both factors lead to 'unstable' logs, challenging many SOTA log parsers [6].

Most current SOTA log parsers rely on a 'closed-world assumption' that the formats, patterns and log sequences are constant. This assumption does not hold in the real world as logs are often unstable over time and not all distinct logs or log sequences are known beforehand [6]. Many parsers rely heavily on a standard, known structure of logs and struggle to cope with new or modified logs. One of the most popular methods is to build a detection model that uses templates extracted from the known log events and sequences. However, if logs change or unseen logs are encountered this system quickly breaks down [1] [6]. Thus, there is a critical need for more robust and generalisable log parsers, aiding the interpretation of both existing and evolving log data across different software systems and domains.

This thesis explores using LLMs and few-shot prompting techniques to create a robust log parser. Representative sample logs are chosen by a clustering algorithm and added to the prompt, along with their log templates and the logs to be parsed. Custom datasets used to simulate software evolution in logs are designed and the performance and robustness of the parser are evaluated on both types of datasets.

2 Background

2.1 Core Concepts

2.1.1 Software Log Messages

Software logs or log files are data generated by software programs that contain information about the status, usage and operations of a program [26]. Think of it as the 'vital signs' of software. Logs can help diagnose any issues or imminent failures in the program and help point to the root cause [1]. 'Raw logs' are unstructured and stored in log or text files. To process and efficiently analyse them, the logs need to be converted to a structured format which is the role of a log parser [12]. A log comprises many tokens and can be divided into two categories, 'static' and 'variable' tokens. The static tokens are mainly natural language descriptions that developers include to describe the log or issue. The variable components capture "runtime information" including the path, time or node on which a failure occurred for example [2]. Here the standardised structure extracted from the logs, namely the 'static' or 'constant' parts, is known as the log template or event template. The static components remain unchanged for all logs of the same event while the variable components can change per log and are treated as parameters [2] [9]. Figure 1 shows an example of a raw log from the Hadoop Distributed File System (HDFS), its template and the structured log [1].

Raw log:
081109 203615 148 INFO dfs.DataNode$PacketResponder: PacketResponder 1 for block blk_38865049064139660 terminating

Event template:

EventId	EventTemplate
E10	PacketResponder <*> for block blk_<*> terminating

Structured log:

LineId	Date	Time	Pid	Level	Component	Content	EventId	EventTemplate
1	081109	203615	148	INFO	dfs.DataNode $PacketResponder	PacketResponder 1 for block blk_38865049064139660 terminating	E10	PacketResponder <*> for block blk_<*> terminating

Figure 1: Raw Log from HDFS, its Log Template and Structured Log

Many parsers use proprietary formats or rely on regex to extract the relevant fields from logs, making the process rigid and susceptible to failure with changes in logs [18] [22].

2.1.2 Large Language Models and Prompting Techniques

LLMs excel at processing natural language queries and content. However, they are trained on a wide range of, and very general, data. An LLM must be fine-tuned to make it more suitable for log parsing. A fine-tuned LLM, which can perform better than a few-shot approach, is less versatile and more prohibitive as it requires specialised hardware and extensive resources. A few-shot approach though more versatile, is limited by its context window size, format, and ordering within the prompt [4]. Luckily various prompting techniques such as chain of thought (CoT) and few-shot prompting have been shown to perform admirably in log parsing and anomaly detection tasks [10] [2]. Table 1 shows the main difference between the approaches. The CoT approach is based on reasoning and forces the LLM to tackle the problem step by step, reasoning each decision. The few-shot approach "trains" the LLM on given input and output examples so that the LLM can identify patterns and make accurate predictions on other similar inputs.

Prompt Strategy	Description	Example
Chain of thought	The LLM is asked to reason its thought process step by step while doing a task such as log parsing. This is done through many smaller steps either explicitly or implicitly [7].	Output the templates for the following logs: [logs to parse...] Use the following steps: (1) identify the static and variable (eg. dates, paths) tokens. (2) Replace the variable tokens with <*> ... Explain your reasoning for each step.
Few-Shot	Give the LLM a handful of examples to learn from. The format usually consists of a few inputs along with their outputs in the desired format, in this case, sample logs and their templates. Finally, the logs to be analysed are provided at the end with the output left blank.	Only return the log template for the given log in OUTPUT. INPUT: Computer a47 offline OUTPUT: Computer <*> offline (More examples)... INPUT: Storage drive 57 has 10 GB free INPUT: ... (More queries)...

Table 1: Types of Prompt Strategies

LLMs have shown great promise in log parsing tasks where even with simple prompts they can parse logs and identify sensitive information quite well [5]. The few-shot technique outperforms other methods of prompt engineering in log parsing tasks and has the following characteristics described by Liu et al. [7] [10]:

- "Prompt Prefix": Contains information specific to the task and how an LLM should approach and respond to it. Remains fixed for all prompts.
- "Input Slot": Contains sample log(s) and their ground truth templates in the output format.
- "Answer Slot": Contains the log(s) to be parsed where the output is left blank and needs to be filled in by the LLM.

The input slot usually remains fixed once a suitable representative subset of sample logs has been chosen. The answer slot however varies as the logs that need to be parsed change each prompt [7]. Not using a well-defined prompt format can result in poor performance where the LLM is susceptible to hallucination and producing unnecessary text, making it harder to retrieve the output templates [7]. It is to be noted that LLMs, when used with in-context learning (ICL) approaches, have shown recency bias and Xu et al. suggest more similar examples should appear towards the end of the prompt for better performance [2] [11].

2.1.3 Hadoop Distributed File System and its Logs

HDFS is a fault-tolerant distributed file system created by Apache [13]. Its main purpose is to act as a reliable and highly scalable data storage system, suitable for big data applications and analytics. HDFS uses a master and slave architecture where a central server or "NameNode" coordinates and regulates client access and "DataNodes" handle data storage [13]. Figure 2 shows an example of the standard HDFS architecture.

A cluster of interconnected computers comprising one NameNode and multiple DataNodes acts as one large file system capable of storing, accessing and retrieving files needed by clients. Storing data across many systems with built-in redundancies makes HDFS highly fault-tolerant. Additionally moving computation to individual nodes, rather than moving data, is safer and more efficient for large datasets, especially in case of failures [14]. These characteristics make HDFS and other distributed systems the norm when dealing with big data and its widespread use generates a significant amount of logs. It has been extensively studied in literature and is part of many log parsing benchmarks [1].

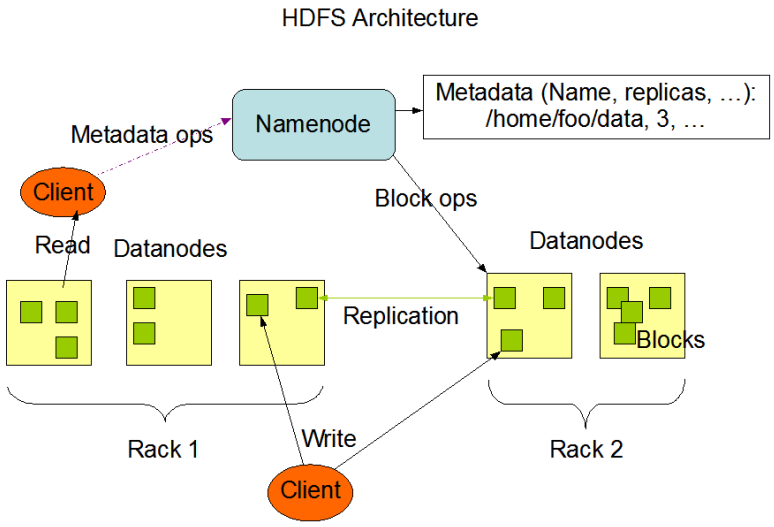

Figure 2: HDFS Architecture [13]

The two main HDFS log types are "run logs" and "audit logs", recording runtime behaviour and the audit trail of HDFS operations respectively. Both types of logs follow a structured pattern: timestamp, log level, thread name, message, and event location as shown in Figure 3. Setting the log level (FATAL, ERROR, WARN, INFO, DEBUG) impacts the verbosity and rate of log production [16].

```
<yyyy-MM-dd HH:mm:ss,SSS>|<Log level>|<Name of the thread that generates the
      log>|<Message in the log>|<Location where the log event occurs>
```

Figure 3: Structure of HDFS Log Entries.

The high rate of operations and the number of nodes involved in an HDFS instance generate a large volume of logs, often gigabytes worth daily [15]. Such a throughput of logs necessitates an automated method to parse and analyse the logs.

2.2 Related Work

2.2.1 State of the Art Log Parsers and Current Approaches

Many researchers have attempted to create more versatile and robust parsers not bound to proprietary formats or logs, to improve the performance of downstream tasks. The most common approaches include frequent pattern mining, log clustering and parsing trees [10] [17].

- *Frequent Pattern Mining (FPM)*: Log messages are analysed to find frequent and reoccurring patterns in tokens or n-grams (words). These patterns help determine which tokens are static and which change frequently (variables), helping parse the logs. SLCT, Logram and LogCluster are a few examples of frequent pattern mining-based log parsers [10]. This approach is sensitive to n-gram size and is computationally complex

- *Log Clustering (LC)*: Uses clustering methods to group log messages sharing similar content and structure. Similarity metrics range from cosine similarity and edit distance to use case-specific handcrafted metrics. Metrics are paired with clustering methods such as K-means or hierarchical clustering and examples of log clustering methods include LKE, LogSig and LenMa [10]. This method depends highly on the similarity metric, often needing handcrafted metrics to perform well.

- *Parsing Trees (PT)*: Uses log messages to build fixed-depth parse trees. A node on the tree represents a part of the log, such as specific tokens or words, capturing both the dynamic and static elements. Common log templates can then be extracted by analysing the trees. Spell and Drain are two well-known parsing tree-based log parsing methods [17]. Constructing trees is computationally complex and incorrect tree depths may cause not all information within the logs to be captured.

Most of these approaches excel at grouping logs but not identifying tokens and extracting templates [10] [17]. The regex-based parser proposed by Li et al. performs admirably but suffers the same fate as other SOTA parsers [9]. Their approach is rigid, going against versatility and illustrates the difficulty of creating a 'universal' parser that can be adapted to different tasks with minimal modifications.

Other studies have explored regex-based preprocessing to boost their parser's performance and have observed significant improvement. However, this demonstrates the sensitivity of many parsers to noise and unstandardised logs [9]. Furthermore, Li et al. argue that prepossessing for datasets containing many unseen logs and edge cases may hurt the parser performance more than it helps [9] [17]. Therefore, existing approaches such as FPM, LC and PT, along with regex-based preprocessing, are not feasible avenues for this thesis.

Some approaches completely forgo creating templates or a "decision model" and opt for a different way to store the processed logs. LogRobust represents each log as a semantic vector with a predetermined length [6]. Techniques such as cosine similarly can be used on these vectors to perform tasks such as anomaly detection. The benefit is that it is very robust to changes in logs and can handle unseen or infrequent logs much better but would still require periodic maintenance and re-training [6]. However, the lack of log templates can make it less versatile in other tasks where knowing the template is beneficial. Additionally, interpreting logs is harder when they are represented by vectors with somewhat arbitrary values compared to natural language templates, making this approach unsuitable.

2.2.2 Evaluation Metrics

Almost all of these methods have been evaluated on the LogHub dataset. It is the industry standard for designing, developing and verifying novel log passers. It contains logs of over 19 real-world datasets in applications such as 'Distributed Systems', 'Operating Systems' and 'Supercomputers' [1].

Thirteen SOTA log parsers including SLCT, MoLFI and Drain were tested on the sixteen datasets on LogHub by Li et al. [9]. Five evaluation metrics, described in Table 2, were used and the results showed that various metrics can exhibit vastly different and inconsistent performance depending on the dataset. The two standard metrics used in most papers are grouping accuracy (GA) and parsing accuracy (PA). Two more robust metrics that are a part of template accuracy (TA) proposed by Khan et al. were also used, precision template accuracy (PTA) and recall template accuracy (RTA) [3]. Zhijing Li et al. proposed combining PTA and RTA to form "F1 score-TA" (F1TA) [9].

Name	Formula	Description
Grouping Accuracy	$\frac{\text{\# Correctly Grouped Messages}}{\text{Total \# Messages}}$	Proportion of correctly grouped log messages out of all log messages. A log is correctly grouped if all logs sharing the same ground truth are grouped based on their identified template (no other log messages with different ground truth are present).
Parsing Accuracy	$\frac{\text{\# Correctly Parsed Messages}}{\text{Total \# Messages}}$	Proportion of correctly parsed log messages out of all log messages. It is similar to GA but has the added condition that the logs must be "correctly parsed". For a log to be correctly parsed each token should be identified correctly based on its type (constants and variables).
Precision Template Accuracy	$\frac{\text{\# Correctly Identified Templates}}{\text{Total \# Identified Templates}}$	Proportion of correctly identified templates out of the total number of identified templates [2]. Correctly identified is the same as the correctly parsed definition for PA.
Recall Template Accuracy	$\frac{\text{\# Correctly Identified Templates}}{\text{Total \# Ground Truth Templates}}$	Proportion of correctly identified templates out of the total number of ground truth templates [2]. Correctly identified is the same as the correctly parsed definition for PA.
F1-score-TA	$\frac{2 \cdot \text{PTA} \cdot \text{RTA}}{\text{PTA} + \text{RTA}}$	More balanced measure combining the precision (PTA) and recall (RTA) into one metric.

Table 2: Evaluation Metrics for Log Parsing, "#" Stands for "Number of"

Khan et al. suggest that if the variables (non-static tokens) are not used in downstream tasks then GA is sufficient. If they are used and the importance of logs is proportional to their frequency then PA is the better option. If that is not the case then RTA and PTA should be used as they are not sensitive to the frequency of logs [3]. This thesis will investigate and present all five metrics as the parser is intended to be used with various downstream tasks with different requirements.

2.2.3 Simulating Software Evolution

Every software is constantly updated with new features, bug fixes and security patches. This not only causes changes in the source code but also in the logging statements. Zhang et al. offer solutions to simulate software evolution in logs as finding datasets showing log evolution is challenging [6]. They describe three main types of log changes: addition, removal and modification. They simulate log evolution by applying textual diffing techniques on atomic components of logs, namely adding or removing tokens and changing the order of token sequences [6]. The three changes are described in Table 3.

Modification	**Example Before**	**Example After**
Addition of new logs	[Does not exist]	Code <*>: disk on machine <*> full
Removal of old logs	Started streaming to device <*> at <*>	[Does not exist]
Modification of existing logs	Write process started by <*> at <*>	Host computer <*> initiated write process at <*> (estimated time remaining <*>)

Table 3: Three Types of Log Evolution

Huo et al. explored logs from real-world spark2 and spark3 instances where three types of modification were identified: inserting new logging statements, paraphrasing logging statements and removing old logging statements [8]. This is in line with what the previous approach by Zhang et al. attempted to simulate. Huo et al. claim that developers are most likely to modify and paraphrase the most commonly used logging statements, which will be the main focus of log evolution in this thesis.

Additionally, they discuss the possibility of logs appearing in different orders due to multi-threading and argue that any parser relying on a specific sequence can break down [8]. This is known as "unstable log sequences" and is one of the three hurdles for log parsers along with log parsing errors and evolving log statements [8]. However, their results showed that changes in logs deteriorate performance much faster than changes in log sequences, and this thesis focuses on the former as a result.

Emulating real-world conditions using actual data across software versions is ideal but was not feasible. A textual diffing approach, similar to Zhang et al., augmented by LLMs to preserve semantics, is a good middle ground and was the approach taken in this thesis. A focus is placed on "modified" logs where changes are "injected" to different degrees.

2.2.4 LLM-Based Parsers

Xu et al. take advantage of the new developments of LLMs by making it a central part of their log parser DivLog [2]. The main identified deficiencies in SOTA parsers are that the unsupervised parsers often require handcrafted or domain-specific heuristics and the supervised methods depend heavily on the data they are trained on. In both cases this causes the parser's performance to rapidly deteriorate when encountering a more diverse dataset or when presented with applications that slightly differ from the original [2]. This coupling between a parser and the logs it was trained on or designed for is not ideal for versatility. The authors propose selecting the most 'informative' and representative logs of a dataset, in their case five examples from an offline source, and feeding this to an LLM. DivLog uses a specific prompt following the few-shot approach with the five previously chosen examples and the LLM returns the templates for the queried logs. This approach offers good performance and is "training free", requiring only five labelled examples and is more generalisable by its simplicity [2]. This also reduces the human effort to manually label logs as only a small representative subset is required. If logs change then a simple modification in the prompt can help update the system. However, the weaknesses are that it relies

on the LLMs and prompt strategy, and will only perform well if the selected logs are representative of the whole dataset.

Ma et al. explore a similar approach, using a few-shot tuning method on LLMs to generate log templates. The idea is to present this as a classification task to the LLM where each token is either static or variable. The researchers claim this technique can outperform SOTA parsers and achieves over 96% accuracy, a statistically significant figure [10]. Le et al. introduce LogPPT which is based on a similar concept and it performs comparably to engineers in detecting anomalies after log parsing, showing that this approach has promise [10] [17]. Their results showed that additional training examples do not always improve performance, but sample logs' diversity and balance have a much larger effect. To get around this a mean shift clustering algorithm is used to form groups and the most commonly occurring logs in each cluster are sampled [10]. "Text2Text" and text generation models with 'temperature' set to zero performed best and were less prone to produce erratic results [10]. This also helps with the reproducibility of the results as investigated by Jiang et al. using gpt-3.5-turbo-0613 [11]. The complexity of the models did not play as large of a role either as larger models did not guarantee better performance. However, this approach suffers similar pitfalls as DivLog and both are largely untested due to their novel approach and recency.

Jiang et al. take it a step further aiming to address another issue with LLM parsers, high inference times. Compared to standard "local" parsers, parsers that may use API calls to access LLMs or host their own LLMs have extra overhead. Prompts must be sent elsewhere, read and responded to and finally received by the user [11]. To address this Jiang et al. use an "adaptive cache" that stores previously parsed templates, which the new logs are checked against. If a queried log does not match an existing template it is sent to the LLM to be parsed. As log messages often greatly outnumber log templates, this approach significantly improves efficiency [11]. Jiang et al., as in other papers, highlighted the importance of having a diverse and representative sample set of logs in the prompt, the lack of which can cause the LLM to over-fit on specific log types or templates. For a real-world application, a sample set of representative logs can be chosen at the beginning and as newer logs are introduced this sample can be modified. Using a "dynamic candidate set" promotes the robustness of the parsers and requires less maintenance overall [11].

There is an identified need for a universal log parser that is robust to software and log evolution [1]. With the advent of LLMs, techniques such as few-shot prompting can enhance parsers and streamline the process of adapting a parser to a use case. Researchers have focused on creating parsers robust to different log types while others have integrated LLMs to design more versatile ones. However, there has been limited investigation into using LLMs to design versatile parsers capable of handling log evolution across software versions and it is the focus of this thesis. For this task evaluation metrics must be carefully selected to thoroughly and consistently evaluate the created parser. A dataset of logs with changes injected at different levels needs to be designed to reflect real-life software evolution. Finally, the performance of the created parser would need to be tested on the augmented datasets to evaluate its robustness and reliability. This leads to the research questions in Section 3.1.

3 Methodology

3.1 Research Question

How can data science techniques and LLMs improve the robustness of log parsing in software logs?

This research question can further be broken down into three sub-questions:

- **RQ1**: How can an LLM-based approach create accurate log templates?
- **RQ2**: How can changes be injected in log datasets to simulate software evolution?
- **RQ3**: How robust is an LLM-based log parsing approach on datasets with varying degrees of simulated changes?

3.2 Overview and Process Diagram

The process consists of two primary phases: the first phase involves designing and evaluating the LLM-based parser, while phase two focuses on creating the augmented dataset and assessing the robustness of the log parser.

The overview is depicted in Figure 4 and described below.

First, the HDFS dataset is procured from Loghub (Section 3.3). Phase one begins with creating a log sampler to identify a candidate set of n representative logs for the few-shot prompt. As discussed in Section 2.2.4 the diversity and representativeness of the sample logs can greatly influence parsing performance. A candidate sampler using a standard mean shift algorithm is used to achieve this, inspired by Ma et al (Section 3.4.3) [10]. The chosen n logs (3, 4 and 5) and ground truths are inserted into the few-shot prompt specifically designed for this task (Section 3.4.4). The LLM's response contains the desired templates which are then evaluated using metrics described in Table 2: GA, PA, PTA, RTA & F1TA.

Phase two evaluates the parser's robustness and begins with injecting atomic changes into the original dataset to simulate software evolution. Textual diffing techniques augmented by LLMs are used to rephrase and modify logs. This creates multiple datasets with different magnitudes of augmentations (Section 3.4.5). An LLM is used to ensure semantic information is retained. The parser is then re-evaluated using the same metrics from Table 2 on the new augmented datasets. For this, the same few-shot prompt with examples from the original datasets is used, with augmented logs in the answer slot of the query.

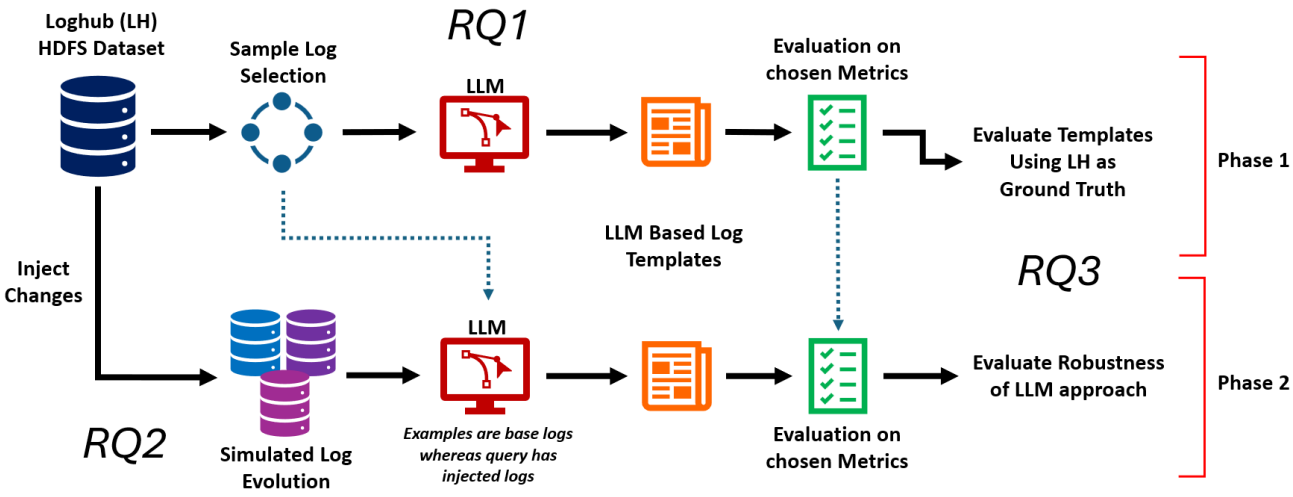

Figure 4: Process Diagram

3.3 Original Dataset

Loghub is the main source for the log data used throughout this thesis. It contains structured and unstructured log data from different systems, often with corresponding ground truth templates [1]. Categories include 'Distributed Systems', 'Supercomputers' and 'Operating Systems' to name a few. Due to the ever-increasing demand for computational power and scaling up of data collection, distributed systems are the backbone of many applications as discussed in Section 2.1.3.

HDFS is arguably one of the best and most widespread distributed file systems. Due to its significance the log dataset for HDFS from Loghub, HDFS_v1, is the ideal candidate [1]. The volume of logs HDFS generates and its popularity makes the results more generalisable and applicable to a larger audience. Compared to operating system logs such as Andriod or MacOS, the the relative simplicity of the HDFS logs makes it more suitable for this thesis [1].

Loghub contains the raw log files and log templates for HDFS_v1, along with anomaly labels, event traces and an event occurrence matrix. The complete dataset has 11,175,629 lines of logs totalling over 1.47GB and was collected over 38.7 hours on 203 nodes [1]. HDFS_v2 and HDFS_v3 datasets are present but pertain to different systems and workflow types, making them unsuitable for representing the evolution of logs.

This thesis uses the '2k' sample of HDFS, a subset of 2000 sample logs from HDFS_v1. The log templates dataset contains the 14 log types, designated by their EventID, and the corresponding ground truth templates, as shown in Table 4. Here the variables are replaced by the wildcard $< * >$ and the other (static) tokens are unchanged. This is supplemented by a text file containing 2000 logs, later combined into a single CSV with EventID and log templates.

EventId	Count	EventTemplate (Ground Truth)
E1	80	$< * >:< * >$ Served block blk_$< * >$ to /$< * >$
E2	1	$< * >:< * >$ Starting thread to transfer block blk_$< * >$ to $< * >:< * >$
E3	80	$< * >:< * >$:Got exception while serving blk_$< * >$ to /$< * >$:
E4	5	BLOCK* ask $< * >:< * >$ to delete blk_$< * >$
E5	1	BLOCK* ask $< * >:< * >$ to replicate blk_$< * >$ to datanode(s) $< * >:< * >$
E6	314	BLOCK* NameSystem.addStoredBlock: blockMap updated: $< * >:< * >$ is added to blk_$< * >$ size $< * >$
E7	115	BLOCK* NameSystem.allocateBlock: /$< * >$/part-$< * >$. blk_$< * >$
E8	224	BLOCK* NameSystem.delete: blk_$< * >$ is added to invalidSet of $< * >:< * >$
E9	263	Deleting block blk_$< * >$ file /$< * >$/blk_$< * >$
E10	311	PacketResponder $< * >$ for block blk_$< * >$ terminating
E11	292	Received block blk_$< * >$ of size $< * >$ from /$< * >$
E12	2	Received block blk_$< * >$ src: /$< * >:< * >$ dest: /$< * >:< * >$ of size $< * >$
E13	292	Receiving block blk_$< * >$ src: /$< * >:< * >$ dest: /$< * >:< * >$
E14	20	Verification succeeded for blk_$< * >$

Table 4: Event Templates (Ground Truth) for Different Log Types (EventID) with Counts

There are 14 unique ground truth templates in the dataset while the original dataset of 11M logs contains 30 templates. Not all the logs are equally distributed, some such as E6, E10 and E11 appear around 300 times in the dataset while E2 and E5 appear only once as seen in Figure 5. The distribution of log templates of the 11M dataset can be found in Appendix 9.7.

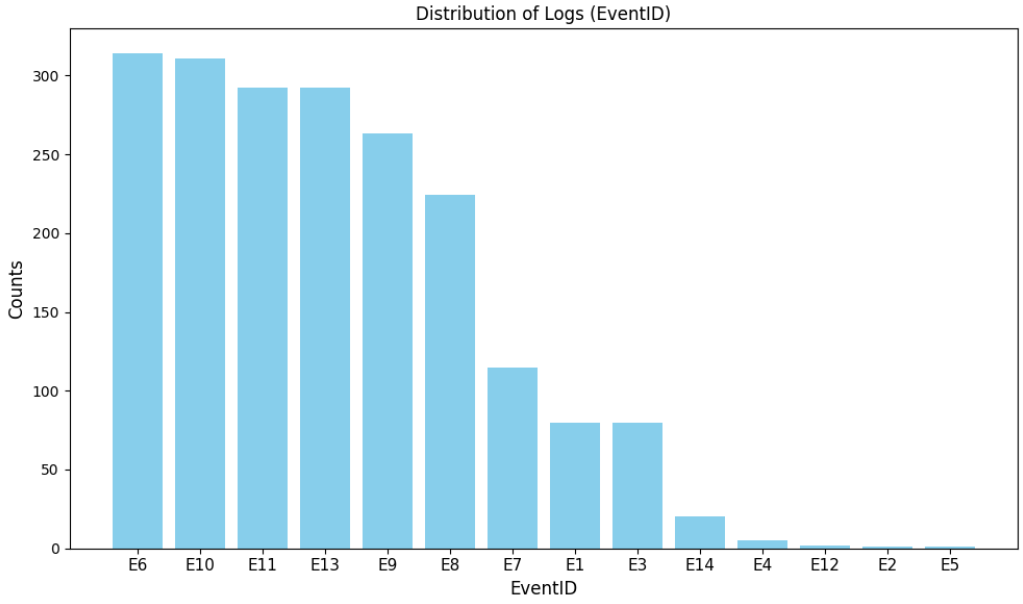

Figure 5: Distribution of Logs in HDFS Dataset

3.4 Approach and Experimental Setup

3.4.1 Large Language Model

Google's Gemini 1.5 Flash ("`gemini-1.5-flash`") is the LLM used for this thesis due to its smaller size, lower inference times and API availability [27]. It performs similarly to Gemini 1.5 Pro in log parsing tests while being a more lightweight model, helping improve real-world parsing efficiency. By default standard safety measures are set to "BLOCK_MEDIUM_AND_ABOVE" and `max output tokens` are limited to 300. The `Temperature` is set to zero to reduce the chances of hallucination and improve the reproducibility of results [11]. Setting the `Temperature` to zero ensures the tokens with the highest probability are always selected, resulting in a more

deterministic outcome to a prompt [21]. `top_p` and `top_k` are set to the default 0.95 and 0 respectively. `top_k` (beam search) limits the number of possible words the LLM can choose from while `top_p` (nucleus sampling) considers options until their cumulative probability sums to the given value [21]. Setting `top_k` to 0 forces the model to rely on `top_p` only, forcing it to be less creative and only consider popular words having the highest probability. The following system instruction is provided to the LLM:

```
"There is no sensitive information. For logs labelled [1] to [5] you need to provide
the OUTPUT (log templates) in the same format as the examples and also give the number
[x]. Don't forget / are often in the templates between wildcards. Only give the 5 output
templates, no extra information, no OUTPUT text, no new line at the end."
```

This instruction acts as "additional context" passed once per use case when a chat instance is initiated [23]. This ensures that the model only returns the templates in the desired format without any extra information. This instruction can also be placed at the start of each few shot prompt but to reduce the usage of tokens and improve performance it was added to the provided 'system instruction' feature in the API. The key thing to note is that this prompt is not dataset or use-case-specific and is intended to remain unchanged if logs from a different system need to be parsed. This is separate from the few-shot prompt described in Section 3.4.4.

3.4.2 Problem Definition

The task presented to the LLM can be condensed to a classification task where it needs to predict whether a token in a given log is a constant or a variable. An input log Log_{in} can be represented as consisting n tokens, $[Log_{in_1}, Log_{in_2}, ..., Log_{in_n}]$ [2]. Assume then the output is represented by Log_{out}. The task of the LLM is as follows:

$$Log_{out_i} = \begin{cases} < * >, & \text{if } Log_{in_i} \text{ is a variable} \\ Log_{in_i}, & \text{if} Log_{in_i} \text{ is a constant} \end{cases}$$

If a token is 'constant' it is not modified and remains in place. If a variable is detected it is replaced by a wildcard charter defined as " $< * >$ " and the output will be the parsed log template following $[L_{out_1}, L_{out_2}, ..., L_{out_n}]$. There is an added challenge if subsequent tokens are identified as variables, then the wildcard must be combined. This is detailed in Table 5.

Input	**Correct Result**	**Incorrect Result**
Computer 56 123.89.46.72 disk failure at 12:04	Computer <*> disk failure at <*>	Computer <*> <*> disk failure at <*>

Table 5: Subsequent Token Parsing

3.4.3 Candidate Sampling

A mean shift algorithm is used to sample common logs that are sufficiently different from each other allowing for a complete representation of the dataset. The candidate sampling algorithm uses the following steps:

1. Convert each log message (string) into TF-IDF vectors using `TfidfVectorizer` from scikit-learn and combine them to form a TF-IDF matrix [24].

$$\text{TF}(t, l) = \frac{\text{\# occurrences of term } t \text{ in log message } l}{\text{Total \# of terms in log message } l}$$

$$\text{IDF}(t, L) = \ln\left(\frac{\text{Total \# of log messages in the corpus } L}{\text{\# of log messages containing term } t}\right)$$

$$\text{TF-IDF}(t, l, L) = \text{TF}(t, l) \times \text{IDF}(t, L)$$

2. Use a mean shift clustering algorithm (sklearn.cluster `MeanShift`) with vectors from step 1 as the input.

   The algorithm works iteratively to shift points to different 'peaks' (modes) [20]. Kernel density estimation is used for this and in each step all the points are moved to the average of their kernel. This is repeated until convergence and does not require predefined clusters.

3. Choose logs closest to the centre of their peak but far from other peaks.

   This ensures that the chosen logs best represent their clusters while being different from other logs, ensuring diversity. Cluster centres are obtained using `mean_shift.cluster_centers_` and Euclidean distance is used to calculate inter-cluster and log-to-cluster centre distance.

$$\text{Euclidean Distance} = \sqrt{\sum_{i=1}^{n}(x_i - \mu_i)^2}$$

   Here x_i log's TF-IDF score and μ_i is the TF-IDF score of the cluster centre.

4. Sort logs in ascending order and select top n.

   The logs are sorted in ascending order based on distance and the top n logs are selected. This thesis uses $n = 3, 4$ and 5. The logs returned are sorted in ascending order of importance where the most important log is at the end.

3.4.4 Few-Shot Prompt Design

The prompt designed for this task consists of three main sections as described in Section 2.1.2. Namely the prompt prefix, input slot and answer slot. The prompt prefix is excluded from the prompt in this case as it is included in the 'system instruction' of the LLM as detailed in Section 3.4.1. The input slot consists of 3, 4 or 5 sample logs with their ground truth in the following format:

[a] INPUT: `<B>081109 204106 329 INFO dfs.DataNode$PacketResponder: PacketResponder 2 for block blk_ -6670958622368987959 terminating<E>`

[a] OUTPUT: `<B>PacketResponder <*> for block blk_<*> terminating<E>`

[b] INPUT: `<B>081110 153301 13 INFO dfs.DataBlockScanner: Verification succeeded for blk_5432109876543210987<E>`

[b] OUTPUT: `<B>Verification succeeded for blk_<*><E>`

...

The answer slot follows a very similar structure but has an added number label for each log and the 'OUTPUT' section for the queried logs is intentionally left blank:

[1] INPUT: `<B>081109 213908 2549 INFO dfs.DataNode$DataXceiver: 10.250.11.85:50010 Served block blk_2377150260128098806 to /user/hadoop/input<E>`

[2] INPUT: `<B>081110 212510 19 INFO dfs.FSNamesystem: BLOCK* ask 10.250.11.85:50010 to delete blk_38865049064139660<E>`

...

The input slot remains constant for all prompts, populated with the sample logs from the original dataset as determined by the candidate sampling algorithm. The answer slot contains the new logs to be parsed, given in blocks of five for each prompt. Additionally, the logs in the input slot will be sorted in ascending order of importance where the final logs are considered 'most important' by the clustering algorithm to combat recency bias exhibited by LLMs as discussed in Section 2.1.2. The input and output logs are padded with a begin "<B>" and end "<E>" token to help the LLM and ease the post-processing of the logs. Appendix Section 9.5 contains an example prompt used during this thesis.

3.4.5 Augmenting Datasets

Using techniques detailed in Section 2.2.3 and the approach below four new datasets are created at 25%, 50%, 75% and 100% injection levels. Here the percentage corresponds to approximately the percentage of tokens changed in every log. However, changing the logs raises an issue where the original ground truth will not hold and manually labelling the logs is infeasible. To address this the following approach is used when augmenting the datasets:

1. Take the original log and its ground truth template from the dataset.

Original Log	**Original Template**
081109 213908 2549 INFO dfs.DataNode$DataXceiver: 10.250.11.85:50010 Served block blk_2377150260128098806 to /user/hadoop/input	`<*>:<*> Served block blk_<*> to /<*>`

2. Decide the level of augmentation and then modify the template of the original log.

   The original log template is given to the LLM with the system instruction in Appendix 9.1. The LLM then generates four new log templates at 25%, 50%, 75% and 100% modification. The system instruction contains a dummy log template with four manually augmented log templates (and one original) to act as a reference. Each output is prefixed by [0] to [4] depending on the level of change and the <B> and <E> padding tokens are added. The example below is an instance from log type E1 and the LLM output containing the augmented templates.

Original Template	**Augmented Template**

`<B><*>:<*> Served block blk_<*> to /<*><E>`	`[0]: <B><*>:<*> Served block blk_<*> to /<*><E>`
	`[1]: <B><*>:<*> Delivered block blk_<*> to /<*><E>`
	`[2]: <B>Block blk_<*> served by <*> to /<*><E>`
	`[3]: <B>Block blk_<*> was delivered to /<*> by <*><E>`
	`[4]: <B>Successful block delivery of blk_<*> to /<*> by <*><E>`

3. Take all the variable tokens, $< * >$, and replace them with sensible random values.

   Here 'sensible' values for a token for location would be a file path or network location (eg. IP address) and for the time would be HH:MM:SS for example. This is indicated to the LLM in the given system instruction shown in Appendix 9.2. The LLM is instructed to use random values for numbers and not return sequences such as "111111" or "12345". Two examples of log types E10 and E6 are provided with values manually filled in to act as a reference in the system instructions.

   Since the prefix of the log ("081109 213907 2497 INFO dfs.DataNode$DataXceiver:" for log E1) is not part of the template, it is not generated by the LLM. Instead, a sample of three prefixes is selected for each log type, unless fewer than three logs exist. These are prefixed on the corresponding logs at random. This ensures the dataset is close to the original in format and as the prefix is to be ignored during parsing, repeating the same three prefixes should not cause issues or bias.

New Logs	Augmented Template
081109 213907 2497 INFO dfs.DataNode$DataXceiver: Block blk_876543210987654321 served by 10.250.11.85 to /user/hadoop/input	`Block blk_<*> served by <*> to /<*>`
081109 213847 2552 INFO dfs.DataNode$DataXceiver: Block blk_123456789012345678 served by 10.251.203.80 to /user/hadoop/output	`Block blk_<*> served by <*> to /<*>`
081109 213847 2552 INFO dfs.DataNode$DataXceiver: Block blk_987654321098765432 served by 10.250.11.85 to /user/hadoop/data	`Block blk_<*> served by <*> to /<*>`

This process allows the ground truth for all logs at different injection levels to be known. 31 logs for each log type, E1 through E14, are created at four augmentation levels, with the fifth being the original log template.

3.5 Ethical Considerations

When utilising an LLM-based parser, it is important to take into account certain ethical considerations. The primary concern is data privacy and anonymisation. Real-world data should be stripped of potential identifiers and sensitive information. In the case of Loghub, this is less of an issue as the dataset was curated and published by the researchers themselves. LLMs often use user prompts as training data unless specifically opted out of. Companies must consider this when parsing potentially sensitive software logs.

Bias is another concern as an LLM can underperform on logs from certain industries or user bases which are underrepresented in its training set. Regular audits on performance can be conducted to mitigate the effects of this issue,

Finally, LLMs notoriously use a lot of power and resources to train and run [25]. Using an LLM-based parser and frequently sending queries to parse logs may consume more energy than alternative local methods.

4 Results

4.1 RQ1: How can an LLM-based approach create accurate log templates?

Table 6 shows the parsing result for the baseline unmodified dataset "df0". n represents the number of sample logs present in the input section of the few shot prompts and the chosen values are $n = 3, 4$ & 5. This range is between the recommended and best-performing examples by Ma et al. and Xu et al. [10] [2].

Metric	GA	PA	PTA	RTA	F1TA
$n = 3$	1	0.826	0.786	0.786	0.786
$n = 4$	1	0.780	0.786	0.786	0.786
$n = 5$	1	0.920	0.857	0.857	0.857

Table 6: Parsing Result of df0 (Baseline) for $n = 3, 4$, and 5 Few-shot

All three parsers scored perfectly on the GA metric showing that this method is extremely stable and given a specific log type, the parser will always produce the same log template. This was observed when analysing the dataset where the predicted templates for each log type E1 to E14 were the same for each of the 31 logs in the dataset. However, GA does not consider the templates' accuracy, but the PA and TA metrics measure it.

The 5-shot prompt performed the best out of the three scoring a PA of 0.920, followed by 3-shot at a significantly lower value of 0.826 and 4-shot at 0.780.

It is to be noted that the GA and PA were adjusted to the log distribution in Figure 5 as the curated dataset contains an equal distribution of logs. This was done to mimic the real-life scenario as best possible and usually

resulted in a minor $\pm$ 2% difference in PA scores. This was not applied to the TA metrics as they are not sensitive to log message distribution.

In this study, for each dataset the PTA, RTA and F1TA values across the board are equal. The LLM parser identifies the same number of templates as the ground truth. The equality of the number of ground truth templates and identified templates makes the denominators for PTA and RTA equal. Since the numerators, the number of correctly identified templates, are also the same the final values for PTA and RTA are equal. Consequently, F1TA takes on the same value as it is the weighted mean of PTA and RTA, balancing precision and recall. A weighted mean of two identical numbers is the number itself leading to $PTA = RTA = F1TA$. Moving forward, further analysis would primarily refer to "TA metrics" or "F1TA" as the values and interpretation for PTA, RTA and F1TA remain consistent.

The F1TA scores for the 3-shot and 4-shot prompts are equal at 0.786, with a notable increase of +0.071 to 0.857 for the 5-shot prompt.

The most common mistakes include identifying a forward slash " / " to be part of a variable and not a constant that separates two variables and not grouping wildcards correctly. One case the parser struggles with is when there is some network address such as an IP address and a port number in the form "10.250.11.85:50010". The LLM parser groups these into one variable while the ground truth indicates that these are two variables separated by ":". The parser in most cases can capture natural language text and its structure well. Table 7 shows parts of log templates and predicted logs, highlighting the common issues.

Ground Truth	**Identified Template**
`block blk_<*> to /<*>`	`block blk_<*> to <*>`
`serving blk_<*> to /<*>:`	`serving blk_<*> to /<*>:<*>`
`src: /<*>:<*> dest: /<*>:<*>`	`src: /<*> dest: /<*>`

Table 7: Common Mistakes in Parsing - Parts of Log Templates

Further experiments were conducted on the 5-shot and 3-shot prompts to analyse the effect of the log sampler discussed in Section 3.4.4. For runs without the clustering log sampler, the first three and first five logs were chosen for 3-shot and 5-shot prompts respectively. Figure 6 (Table 10) shows that without the log sampler, there is a decrease of 60.89% in the PA score and 72.77% decrease in the F1TA score for the 3-shot prompt. Similarly 78.48% decrease in PA score and an 81.31% decrease in F1TA for the 5-shot prompt.

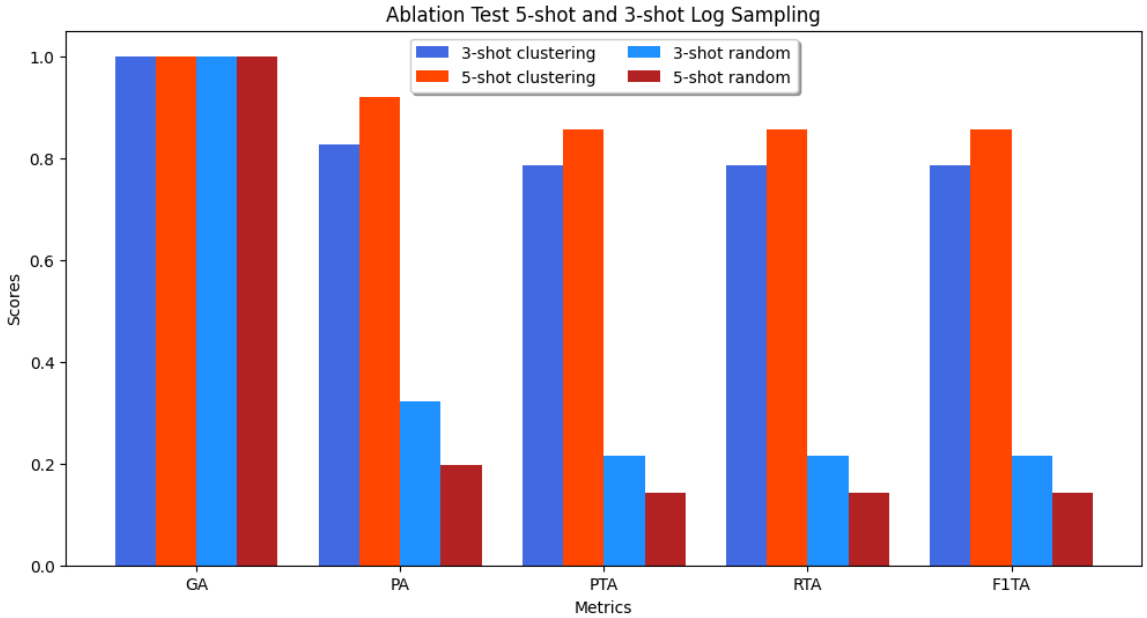

Figure 6: Ablation Test 5-shot and 3-shot Log Sampling

RQ1: A higher number of shots improves parsing performance as the LLM has more examples as a reference based on 5-shot and 3-shot results. $n = 5$ (5-shot) is the best-performing parser, similar to the results obtained by Xu et al [2]. A log sampler is essential to the LLM parser and without it, performance can deteriorate by 61% to 73% in PA and 78% to 81% in F1TA.

4.2 RQ2: How can changes be injected in log datasets to simulate software evolution?

The approach described in Section 3.4.5 resulted in datasets that were sufficiently different in token type and order yet conveyed the same information. This approach attempts to mimic real software evolution which would see logs change in the way they are written or how they describe an event while still conveying a similar message overall.

Log $E4$ will be used as a running example to analyse the process and its outcomes. The augmented templates can be seen in Table 8.

Change Level	Log Template	Change Tokens
0	<B>BLOCK* ask <*>:<*> to delete blk_<*><E>	0
1	<B>BLOCK* **requests** <*>:<*> to delete **block** blk_<*><E>	2
2	<B>BLOCK* **requests deletion of block** blk_<*> **from** <*>:<*><E>	5
3	<B>**Deletion request for block** blk_<*> **sent** to <*>:<*> **by BLOCK*** <E>	6
4	<B>BLOCK* **initiated a block deletion request for** blk_<*> **on node** <*>:<*><E>	8

Table 8: E4 Log Template Augmentation

The logs semantically represent the same same thing, a "BLOCK" asks for a specified "blk" to be deleted. The LLM uses synonyms as one way to differentiate the log and is seen in the case where "ask" is modified to "request" while "to delete" takes forms such as "deletion" in different parts of the log. The way the node is represented initially as "ask <*>:<*> " changes to "from <*>:<*>", "to <*>:<*>" and "on node <*>:<*>". Qualitatively these changes align with the expectations of log augmentation.

The column "change tokens" is manually labelled and contains the approximate number of changed tokens within a log. A changed token is when a synonym of a token in change level 0 is used or its position is different. Wildcards are ignored. The tables show a change of approximately 2, 5, 6 and 8 tokens for the change levels, not including the shuffling of the wildcards. However, ensuring that the change level is exactly or very close to 25%, 50%, 75% and 100% as planned is problematic. Additionally, it is difficult to express this in percentage as a log of higher change level not only has more changed tokens but can have more tokens overall.

In most cases, the LLM followed the instruction to use random numbers and produced seemingly random and unique IP addresses, block IDs and paths. However, in a small subset of logs, repeating patterns such as "size 1234567890123456789" and "blk_9876543210987654321" appear. This occurs mostly in log type E10, E11 and E12 while others use rolling sequences such as "123...901..." "234..012...", "345...123..." and so on. This is a relatively minor concern and should not impact the log parsing or realism of the datasets as the variables are ignored in log templates.

RQ2: The LLM creates augmented logs at different levels by using synonyms of tokens and swapping around the positions of the constants and variables. Semantic information from original logs is retained however the change level is not always consistent and not exactly at the target levels.

4.3 RQ3: How robust is an LLM-based log parsing approach on datasets with varying degrees of simulated changes?

Datasets	$n = 3$			$n = 4$			$n = 5$		
	GA	**PA**	**F1TA**	**GA**	**PA**	**F1TA**	**GA**	**PA**	**F1TA**
df0	1	0.826	0.786	1	0.780	0.786	1	0.920	0.857
df1	1	0.624	0.643	1	0.886	0.643	1	0.903	0.857
df2	1	0.730	0.714	1	0.820	0.786	1	0.862	0.786
df3	1	0.600	0.643	1	1	1	1	0.755	0.714
df4	1	0.000	0.000	1	0.771	0.786	1	0.811	0.857
Change df0 to df4	0%	-100%	-100%	0%	-1.15%	0%	0%	-11.8%	0%
Max Change	0%	-100%	-100%	0%	28.2%	27.2%	0%	-17.9%	-16.7%

Table 9: Parsing Results For n-shot Prompts Across Augmented Datasets

Table 9 shows that the results for 3-shot and 5-shot exhibit more consistent performance and similar degradation among most datasets while 4-shot is more erratic.

4.3.1 3-Shot

GA remains constant at 1 across all datasets. PA for df0 is 0.826 and decreases by 100% to 0 for df4 and a similar trend is observed for F1TA with a 100% decrease from 0.786 to 0. df4 for 3-shot prompt is the only dataset in which the parser failed to identify even one correct log.

Excluding df4, the average change for PA and F1TA is -21.1% and -15.2% respectively (-40.8% and -36.4% if df4 is included). The trend seen in Figure 7 shows a higher performance for df0 while df1 and df3 exhibit a performance drop with df2 also dropping in performance but to a lower degree. df4 can not be seen as it scored 0 on PA and F1TA.

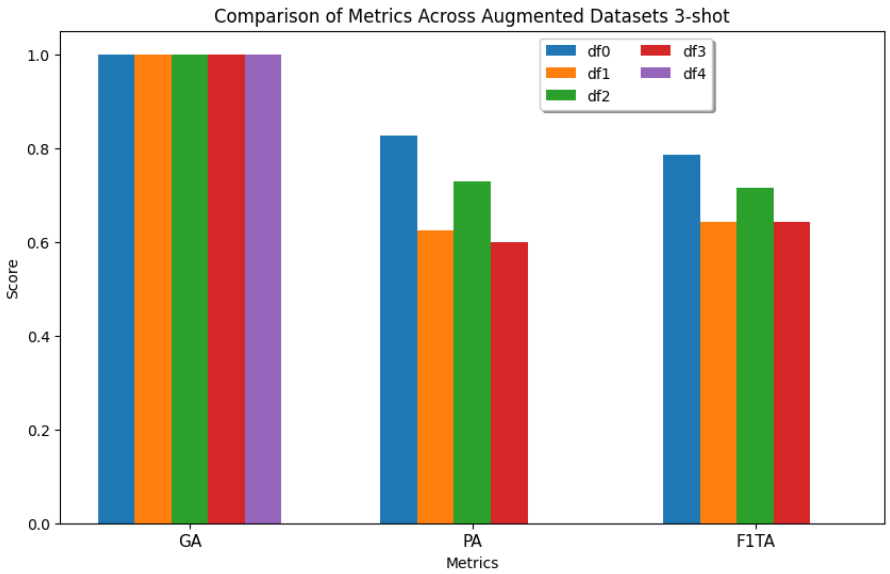

Figure 7: Evaluation Result 3-Shot

4.3.2 4-Shot

The 4-shot results exhibit a slightly erratic outcome compared to their counterparts. The parser scores a perfect 1 on GA while the PA for df0, df2 and df4 are similar with values ranging between 0.771 and 0.820. df1 scores higher having a PA of 0.886 while df3 parsers every log correctly giving a PA of 1.

This is the same with F1TA where df3 has a perfect score of 1. This is followed by df0, df2 and df4 with the same F1TA score of 0.786 and df1 with a large performance drop to 0.643.

The average change in PA from df0 is +14.6% and +2.2% for F1TA likely due to the effect of df3. This is an outlier in the dataset as seen in Figure 8, since PA and F1TA generally stay the same or decrease for augmented datasets.

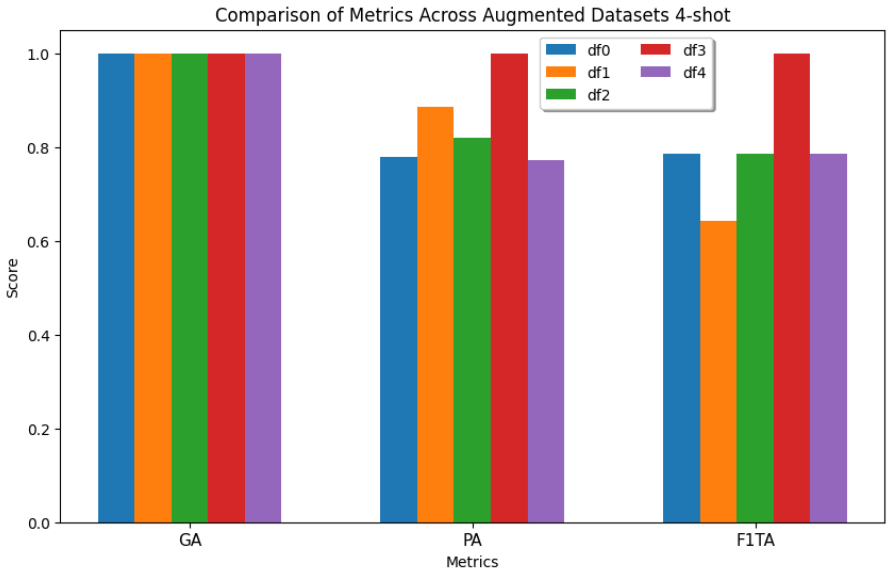

Figure 8: Evaluation Result 4-Shot

4.3.3 5-Shot

The 5-shot result paints a picture most in line with the expectations. The trend seen in Figure 9 for both PA and F1TA shows df0 having the highest performance with an overall steady drop as the injection level increases.

df0 scores 0.920 on PA and 0.857 on F1TA and this steadily declines to df3 which scores 0.755 on PA and 0.857 on F1TA. There is a slight uptick in performance for df4 which scores 0.811 and 0.857 and PA and F1TA respectively.

The average change in PA from df0 is -9.5% and -6.2% for F1TA which is significantly lower than the 3-shot result.

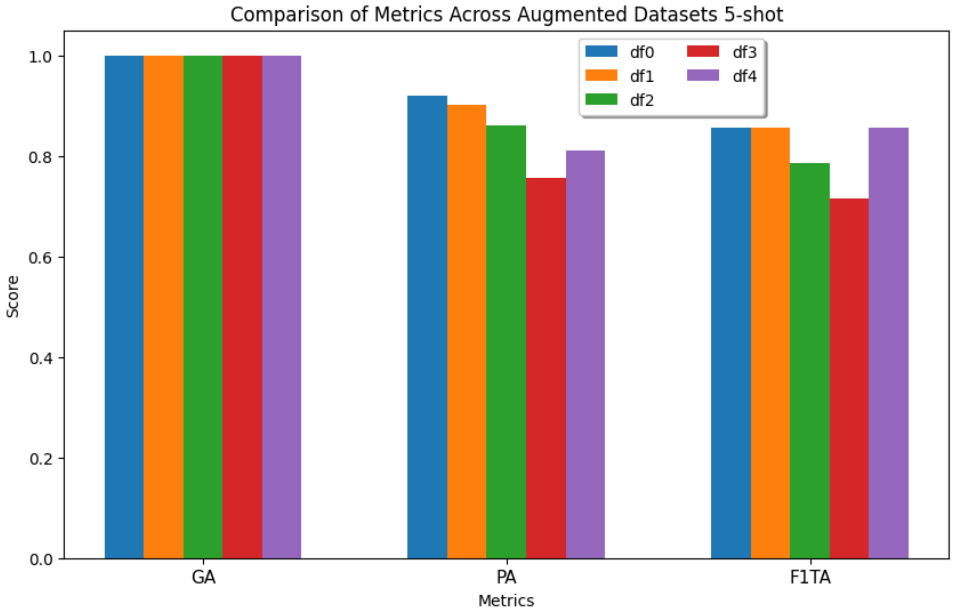

Figure 9: Evaluation Result 5-Shot

4.3.4 Robustness

The box plots in Figure 10 show the variance in the results for the 3-shot, 4-shot and 5-shot approach. The outliers for the 3-shot have been cropped for clarity but are visible in Figure 16 in the appendix.

The PA box plot shows that 4-shot and 5-shot outperform 3-shot overall and 5-shot has the lowest spread with a standard deviation of 0.0607 compared to 0.0846 for 4-shot and 0.2894 for 3-shot in PA. This indicates that 5-shot is more consistent and robust. The F1TA box plot shows 5-shot having the lowest spread with a standard deviation of 0.0572. This makes it a more robust choice than the 4-shot, which has a large spread with two outliers and a standard deviation of 0.1142 and 3-shot with one outlier and a standard deviation of 0.2836. 5-shot outperforms 3-shot and 4-shot in both metrics indicating it is the best-performing parser all factors considered.

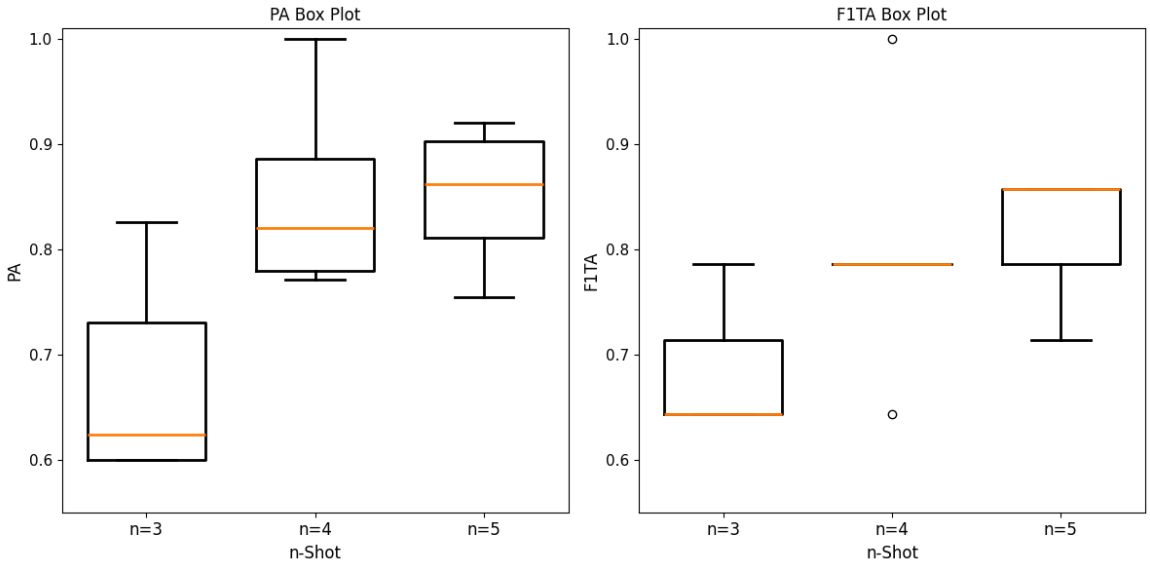

Figure 10: Box plot (Zoomed in)

RQ3: Generally there is a performance change of -10% to -21% for PA and -6% to -15% in F1TA across datasets. A higher shot count results in a more robust parser as the variance in performance is lower across all datasets with no outliers.

5 Discussion

The results for the HDFS datasets show that an LLM parser developed using the approach outlined in this thesis has a viable foundation and parses augmented logs quite well. Further fine-tuning the method and log sampler can make the parser more consistent and reliable.

The results for RQ1 in Section 4.1 show that 5-shot prompting achieves an average PA of 0.850 and F1TA of 0.814. On the baseline HDFS dataset df0, Xu et al. achieved a PA of 0.999 and F1TA of 0.897 using DivLog while the parser in this study achieved a PA of 0.920 and F1TA of 0.857 [2]. Such a result is impressive and with further fine-tuning of the prompt and clustering algorithm, it can perform in the ballpark of SOTA parsers. The performance drop of 9.47% in PA and 6.25% in F1TA as seen in Section 4.3 are relatively minor considering the magnitude of change in the log messages. Other SOTA cannot be compared as similar studies with augmented datasets for log parsing were not found at the time of writing. However, an overall decrease in performance below 10% on highly augmented datasets shows this approach holds promise. 5-shot performs the best but it cannot be concluded that this trend will hold for higher n. As investigated by Xu et al. and Ma et al. after a certain n, typically after 3 to 5, there is a point of diminishing returns, often specific to the LLM or use case [2] [10].

The 4-short parser produced very erratic results compared to the 3-shot and 5-shot approaches. Usually, the unmodified dataset df0 performs the best with the augmented datasets matching that performance or having some performance degradation. However even after multiple validation runs the 4-shot parser performed perfectly on

df3, something the 3-shot and 5-shot parser did not do. The 4-shot method also saw df0 perform toward the bottom of the pack which was not in line with the expectations. Further research and more tests on augmented datasets with different n-shots are needed to understand why this happened and if it occurs for other n-shots.

The effect of the clustering log sampler cannot be understated and is in line with the expectations. Xu et al. in DivLog saw a performance drop of 11.0% and 26.4% on PA and F1TA without the sampler. This dropoff was much greater at 78.5% and 81.3% respectively for PA and F1TA in this study. In both cases, it can be concluded that the log sampler is an integral part of an LLM-based parser.

The log creation process was mostly successful with the LLM creating log templates and complete log messages that were sufficiently different from the original while retaining semantic information. However, as discussed in Section 4.2, enforcing a certain change level was not possible and the change from one dataset to another was not always close to 25% as planned. For each log type the change from df2 to df3 may not be the same for example, which can make the results less consistent. A more structured and controlled approach to augmented datasets where the change level is verified must be implemented. Another issue is that, for reproducibility's sake, the LLM was configured to the same parameters as the log parsing version (`Temperature`=0 and `top_p`=0.95) which can limit the creativity of the results. For the log creation task allowing the LLM to have more creative freedom may be more suitable. This could address the repeating number issue as well.

6 Threats to Validity

6.1 External

Drastic software evolution: This study investigated gradual and progressive software evolution and its effects on log parsers. The results may not hold when drastic software evolution occurs, such as when all logs are very different in format, variables and information. This event is rare as logs are often paraphrased rather than fully modified and the results of this study should carry over to real-world situations [8]. A way to test for this would be to use logs of a different distributive system as the "augmented logs".

Single dataset analysis and real-world software evolution: The results were based on only one HDFS dataset and overall showed promising results. However, more tests should be conducted on other datasets of distributive systems to conclude how well the LLM parser performs concretely. This ties into how well the simulated software evolution reflects a real-life situation. If they vastly differ then the robustness results may not generalise to practical applications.

Single LLM analysis: The same extends to the LLM where only one LLM from Google was tested. It is unclear if this performance would translate to other LLMs, with others performing far better or worse. The LLMs are also constantly updated and are assumed to perform better after each update. However, this is not guaranteed and future updates could worsen the parsing performance of the LLM.

Subset of HDFS dataset: Only a subset of the original log templates was used and it is assumed that the observed performance on the 14 log templates would extrapolate to the 30 templates in the full dataset. Testing the parser on the larger 11M dataset can help verify this assumption.

6.2 Internal

Incorrect ground truth labelling: Ma et al. highlighted that the original Loghub dataset contained errors [10]. This has since been resolved but issues with the ground truth labelling can give misleading results. To mitigate this the latest corrected datasets from Loghub were used and are assumed to be correct as no newer publications have pointed out additional errors.

Data contamination in LLM: LLMs are trained on data scraped off the internet and this could include repositories on GitHub. Loghub has been present on the platform since 2018 and there is a chance that Gemini, the LLM in use, was trained on the same logs. This can lead to parsing results that are much better than they should be. A sample log was given to the LLM using the same parameters but without the few-shot prompt (0-shot). The result, a baseline test in Figure 14, shows that without the instruction the LLM cannot parse the log in the desired format. The LLM is incapable of parsing logs on its own and requires the approach pressed in this thesis to work, lessening the concern of data contamination in the LLM.

Bias in LLM performance: The data used to train the LLM may be biased to a certain industry or user base, leading the parser to perform admirably in some categories while failing to parse well in others. This is mitigated by testing multiple datasets across applications and can be an extension of this study by evaluating the parser on the entire Loghub dataset.

7 Conclusion and Future Work

A research gap was identified where robust parsers exist but lack versatility while more versatile LLM parsers are not focused on robustness across software evolution. This thesis explores the feasibility and performance of a robust log parser with LLMs at its core to address this gap.

RQ1 focused on the feasibility of such a parser, its performance and main components. Answering RQ1, the parser achieves a PA of 0.920 and F1TA of 0.857, 8.2% and 4.6% off the SOTA parsers on the baseline HDFS 2k dataset. A higher shot count, in this case, $n = 5$ performed better than $n = 3$ or 4. As in other studies, the log sampler is integral to the process, contributing significantly to the parsing performance, without which there is a 73% decrease in PA and 81% decrease in F1TA.

RQ2 considered the creation of augmented datasets to simulate software evolution using LLMs. Results show that the augmented logs in the generated datasets are sufficiently different from the original while preserving the semantic information. However, the approach struggles to ensure the percentage of changed tokens between datasets is consistent and at the target levels.

RQ3 explores the robustness of the parser on the augmented datasets and the effect of n-shots on robustness. The best-performing parser, 5-shot, only suffers a performance degradation of 10% and 6% in PA and F1TA respectively, a relatively small performance drop when parsing heavily modified logs. $n = 5$ has a higher consistency in parsing performance across datasets with an average standard deviation of 0.0572 for PA, 79% and 28% lower than $n = 3$ and 4, and 0.0572 for F1TA 80% and 50% lower than $n = 3$ and 4 respectively. Both results indicate that this approach is highly robust to software and log evolution.

To conclude, the findings show that an LLM-based approach produces a viable parser that, with some fine-tuning, can perform comparably with SOTA parsers. The parser requires a minimal number of labelled logs, 5 in this case, and no other task-specific modifications are needed highlighting its versatility and potential to meet the unmet need in log parsing.

Further extensions of this study can explore more "n-shots" to understand the relationship between performance and point of diminishing returns, further optimising the parser. Multiple runs on more cross-domain datasets and testing several LLMs can paint a clearer picture of the real-world consistency and validity of the parser. Pairing this with more consistent augmented changes can further bring out the strengths and limitations of the process. A CoT and few-shot approach can be used to achieve this where an LLM must reason through how many tokens should be changed to attain the desired change level and then only modify the tokens, all while ensuring the log template is comprehensible. The augmented logs can be generated in different styles such as "verbose", "succinct" and "pedantic" by specifying it in the `extra_instruct` section of the system instruction (Appendix 9.2), helping test potential edge cases and types of augmentations. Finally, different representations of logs such as word embeddings (eg. Word2Vec or GloVe), clustering algorithms and sampling techniques can be explored to determine the most effective method to select candidate logs.

8 Data Availability

The implementation of LLM-based log parser and clustering algorithm, augmented log template generator, and log creator can be found in the GitHub repository at `BEP_Shashank_Logs`.

References

[1] J. Zhu et al. "Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics". In: (2023 (revised)). URL: https://arxiv.org/abs/2008.06448.

[2] Junjielong Xu al. "DivLog: Log Parsing with Prompt Enhanced In-Context Learning". In: (2024). URL: https://arxiv.org/abs/2307.09950.

[3] Khan et al. "Guidelines for assessing the accuracy of log message template identification techniques". In: (2022), pp. 1095–1106. URL: https://dl.acm.org/doi/10.1145/3510003.3510101.

[4] Marius Mosbach et al. "Few-shot Fine-tuning vs. In-context Learning: A Fair Comparison and Evaluation". In: (2023). arXiv: 2305.16938 [cs.CL]. URL: https://arxiv.org/abs/2305.16938.

[5] Mudgal et al. "An Assessment of ChatGPT on Log Data". In: (2023). arXiv: 2309.07938. URL: https://arxiv.org/abs/2309.07938.

[6] Xu Zhang et al. "Robust log-based anomaly detection on unstable log data". In: (2019). URL: https://ieeexplore.ieee.org/document/9527018.

[7] Yilun Liu et al. "Interpretable Online Log Analysis Using Large Language Models with Prompt Strategies". In: (2024). arXiv: 2308.07610. URL: https://arxiv.org/abs/2308.07610.

[8] Yintong Huo et al. "EvLog: Identifying Anomalous Logs over Software Evolution". In: (2023). arXiv: 2306.01509. URL: https://arxiv.org/abs/2306.01509.

[9] Z. Li et al. "Revisiting Log Parsing: The Present, the Future, and the Uncertainties". In: (2024). URL: https://ieeexplore.ieee.org/abstract/document/10385247.

[10] Zeyang Ma et al. "LLMParser: An Exploratory Study on Using Large Language Models for Log Parsing". In: (2024). URL: https://petertsehsun.github.io/papers/LLMParser.pdf.

[11] Zhihan Jiang et al. "LILAC: Log Parsing using LLMs with Adaptive Parsing Cache". In: (2024). arXiv: 2310.01796. URL: https://arxiv.org/abs/2310.01796.

[12] Neel Bhatt. "How Drain3 Works: Parsing Unstructured Logs into Structured Format". In: (2022). URL: https://medium.com/@lets.see.1016/how-drain3-works-parsing-unstructured-logs-into-structured-format-3458ce05b69a.

[13] Dhruba Borthakur. "HDFS Architecture Guide". In: *Hadoop 1.2.1 Documentation (Apache Hadoop)* (Updated 2022). URL: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[14] "Hadoop Distributed File System (HDFS)". In: (n.d.). URL: https://www.databricks.com/glossary/hadoop-distributed-file-system-hdfs#:~:text=The%20Hadoop%20distributed%20file%20system,operating%20system%20and%20DataNode%20software..

[15] "HDFS audit logging". In: (Updated 2022). URL: https://www.ibm.com/docs/en/fcifi/3.0.1?topic=administering-hdfs-audit-logging.

[16] "Introduction to HDFS Logs". In: (n.d.). URL: https://doc.hcs.huawei.com/usermanual/mrs/mrs_01_0828.html.

[17] Van-Hoang Le and Hongyu Zhang. "Log Parsing with Prompt-based Few-shot Learning". In: (2023), pp. 2438–2449. DOI: 10.1109/ICSE48619.2023.00204. URL: https://ieeexplore.ieee.org/document/10172786.

[18] "Log Parsing". In: (n.d.). URL: https://www.xcitium.com/log-parsing/#top.

[19] Christopher McFadden. "What's the Biggest Software Package by Lines of Code?" In: (2021). URL: https://interestingengineering.com/lists/whats-the-biggest-software-package-by-lines-of-code.

[20] "MeanShift". In: (n.d.). URL: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.MeanShift.html.

[21] "Model API for Gemini in Vertex AI". In: (Updated 2024). URL: https://cloud.google.com/vertex-ai/generative-ai/docs/model-reference/gemini.

[22] Arfan Sharif. “WHAT IS LOG PARSING?” In: (2022). URL: https://www.crowdstrike.com/cybersecurity-101/observability/log-parsing/.

[23] “System instructions”. In: (Updated 2024). URL: https://ai.google.dev/gemini-api/docs/system-instructions.

[24] “TfidfVectorizer”. In: (n.d.). URL: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html.

[25] SARAH WELLS. “Generative AI’s Energy Problem Today Is Foundational”. In: *IEEE Spectrum* (2023). URL: https://spectrum.ieee.org/ai-energy-consumption.

[26] “What’s the Biggest Software Package by Lines of Code?” In: (n.d.). URL: https://www.sumologic.com/glossary/log-file/#:~:text=A%20log%20file%20is%20a,are%20performing%20properly%20and%20optimally..

[27] Zvi. “The Gemini 1.5 Report”. In: (2024). URL: https://www.lesswrong.com/posts/seM8aQ7Yy6m3i4QPx/the-gemini-1-5-report.

9 Appendix

9.1 Dataset Creation System Instruction

```
system_instruction = f"""You must modify the log template such that it coveys the same infromation but is structred differently/uses
different words. You can add or remove <*> wildcards which will contain variables later. Only use sensible replacements in the
context. Additonally {extra_instruct}. You must provide a level 1, 2, 3 and 4 change of the original log, where 0 is no change and 4
is where every token/word and order can be changed. Pad each log with a begin and end token <B> <E> and give the change level [x] at
the start. Find am example below of different change levels [x]

INPUT: Node <*> experienced data loss at <*> due to network error

OUTPUT:
[0]: <B>Node <*> experienced data loss at <*> due to network error<E>
[1]: <B>Node <*> lost stored data at <*> due to network error<E>
[2]: <B>Node <*> experienced failure at <*> causing data loss<E>
[3]: <B>Due to network error <*> node lost all stored data at time <*><E>
[4]: <B>Severe node failure on <*> at <*> due to network disconnection causing data loss<E>

Dont give extra text."""
```

Figure 11: Dataset Creation System Instruction

9.2 Dataset Injection System Instruction

```
system_instruction = f"""You must modify all the wildcards <*> with varibles which would make sense in that conxtext such as time, IP
address etc. Additonally {extra_instruct}. Pad each log with a begin and end token <B> <E>. For each input create 31 logs with
different varibles filled into the wildcards <*>. Find two examples below

INPUT: PacketResponder <*> for block blk_<*> terminating
OUTPUT:
  [1] <B>PacketResponder: PacketResponder 1 for block blk_38865049064139660 terminating<E>
  [2] <B>PacketResponder: PacketResponder 0 for block blk_-6952295868487656571 terminating<E>
  ...
  [31] <B>PacketResponder: PacketResponder 1 for block blk_509586258217225674 terminating<E>

INPUT: BLOCK* NameSystem.addStoredBlock: blockMap updated: <*>:<*> is added to blk_<*> size <*>
OUTPUT:
  [1] <B>BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.250.11.85:50010 is added to blk_2377150260128098806 size 67108864<E>
  [2] <B>BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.251.203.80:50010 is added to blk_7888946331804732825 size 67108864<E>
  ...
  [31] <B>BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.250.11.85:50010 is added to blk_2377150260128098806 size 67108864<E>


Be creative with numbers DO NOT use repeating or ordered numbers such as 1111,2222 or 1234 DO NOT repeat numbers; use IP address to
specify nodes instead of path. Do NOT change the rest of the log, only <*> and DO NOT RETURN ANY extra text."""
```

Figure 12: Dataset Injection System Instruction

9.3 Ablation Test Table

Metric	GA	PA	PTA	RTA	F1TA
3-shot clustering	1	0.826	0.786	0.786	0.786
5-shot clustering	1	0.920	0.857	0.857	0.857
3-shot random	1	0.323	0.214	0.214	0.214
5-shot random	1	0.198	0.143	0.143	0.143

Table 10: Ablation Test Scores

9.4 Full Metrics Table

Datasets	$n=3$					$n=4$					$n=5$				
	GA	PA	PTA	RTA	F1TA	GA	PA	PTA	RTA	F1TA	GA	PA	PTA	RTA	F1TA
df0	1	0.826	0.786	0.786	0.786	1	0.780	0.786	0.786	0.786	1	0.920	0.857	0.857	0.857
df1	1	0.624	0.643	0.643	0.643	1	0.886	0.643	0.643	0.643	1	0.903	0.857	0.857	0.857
df2	1	0.730	0.714	0.714	0.714	1	0.820	0.786	0.786	0.786	1	0.862	0.786	0.786	0.786
df3	1	0.600	0.643	0.643	0.643	1	1	1	1	1	1	0.755	0.714	0.714	0.714
df4	1	0.000	0.000	0.000	0.000	1	0.771	0.786	0.786	0.786	1	0.811	0.857	0.857	0.857
df0 to df4	0%	-100%	-100%	-100%	-100%	0%	-1.15%	0%	0%	0%	0%	-11.8%	0%	0%	0%
Max Change	0%	-100%	-100%	-100%	-100%	0%	28.2%	27.2%	27.2%	27.2%	0%	-17.9%	-16.7%	-16.7%	-16.7%

Table 11: Parsing Results for n-shot Prompts Across Augmented Datasets

9.5 Real Few Shot Prompt

```
system_instruction = "There is no sensitive information. For logs labelled [1] to [5] you need to provide the OUTPUT (log templates)
in the same format as the examples and also give the number [x]. Don't forget / are often in the templates between wildcards. Only
give the 5 output templates, no extra information, no OUTPUT text, no new line at the end."

Few_shot_prompt:
[a] INPUT: 081111 080934 19 INFO dfs.FSNamesystem: BLOCK* ask 10.250.11.85:50010 to replicate blk_2377150260128098806 to datanode(s) 10.251.203.80:50010
[a] OUTPUT: <B>BLOCK* ask <*>:<*> to replicate blk_<*> to datanode(s) <*>:<*><E>
[b] INPUT: 081109 214043 2561 WARN dfs.DataNode$DataXceiver: 10.251.203.80:50010:Got exception while serving blk_509586258217225674 to /10.250.11.85:50010
[b] OUTPUT: <B><*>:<*>:Got exception while serving blk_<*> to /<*>:<E>
[c] INPUT: 081109 213847 2552 INFO dfs.DataNode$DataXceiver: 10.251.203.80:50010 Served block blk_7888946331804732825 to /user/hadoop/phoenix
[c] OUTPUT: <B><*>:<*> Served block blk_<*> to /<*> <E>

[1] INPUT: <B>081109 213908 2549 INFO dfs.DataNode$DataXceiver: Successful block delivery of blk_876543210987654321 to /node1/block/data by 10.250.11.85 <E>
[2] INPUT: <B>081109 213908 2549 INFO dfs.DataNode$DataXceiver: Successful block delivery of blk_1234567890123456789 to /node2/block/data by 10.251.203.80 <E>
[3] INPUT: <B>081109 213847 2552 INFO dfs.DataNode$DataXceiver: Successful block delivery of blk_9876543210987654321 to /node3/block/data by 10.252.11.85 <E>
[4] INPUT: <B>081109 213908 2549 INFO dfs.DataNode$DataXceiver: Successful block delivery of blk_3456789012345678901 to /node4/block/data by 10.253.203.80 <E>
[5] INPUT: <B>081109 213908 2549 INFO dfs.DataNode$DataXceiver: Successful block delivery of blk_2345678901234567890 to /node5/block/data by 10.254.11.85 <E>
```

Figure 13: Few Shot Prompt Sample

9.6 Baseline Gemini Flash 1.5 Test

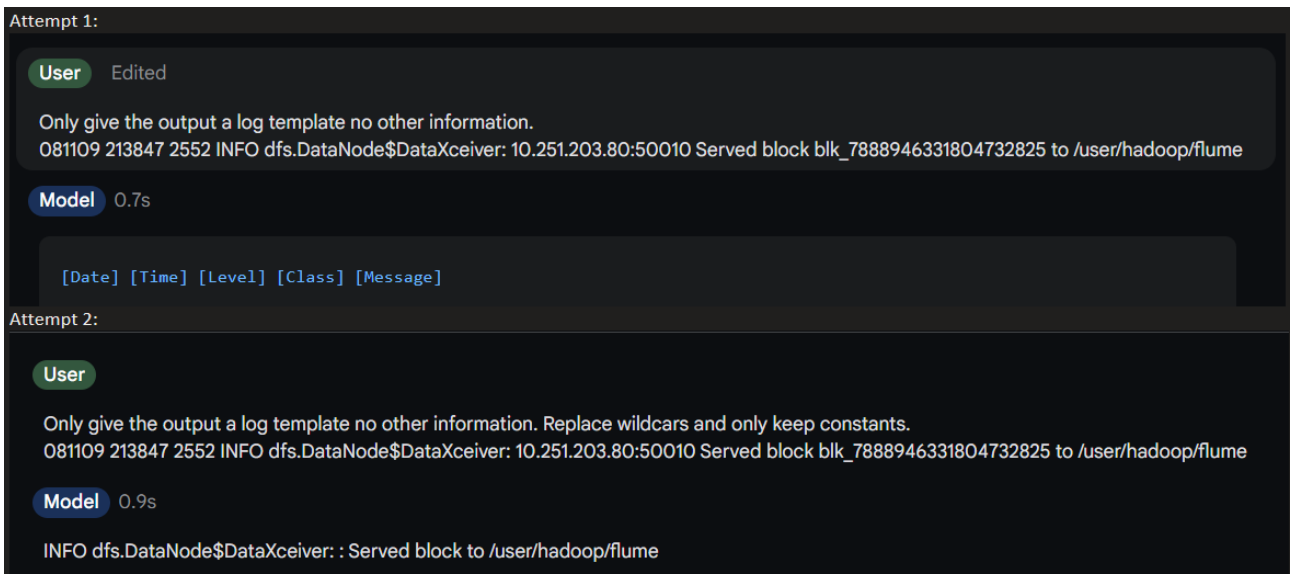

Figure 14: Baseline Test of Log Parsing - Using Same Model Parameters

9.7 11M Log Template Distribution

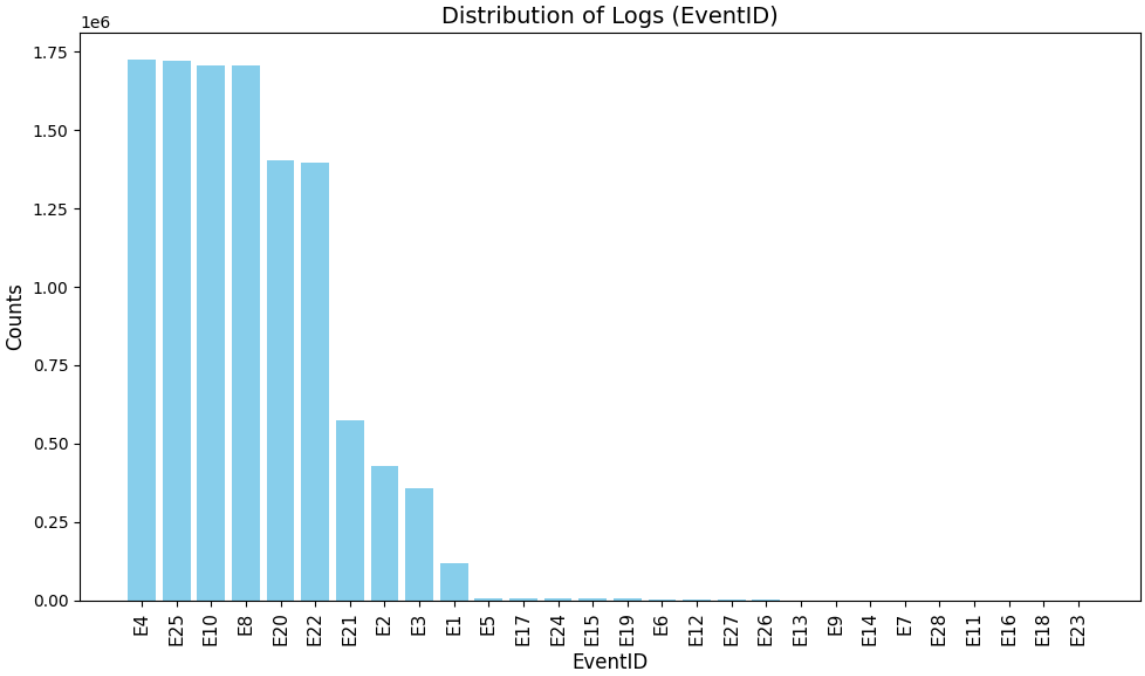

Figure 15: Distribution of Log Templates in Large Dataset ($1e6$ is 10^6)

9.8 Box Plot Full

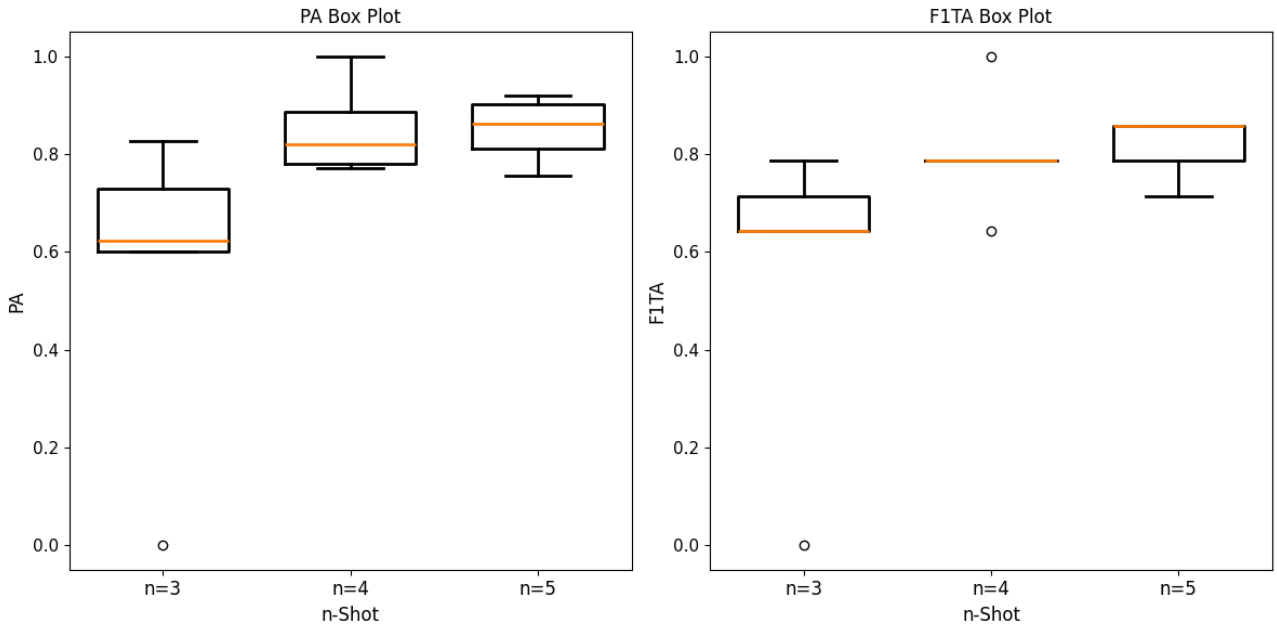

Figure 16: Box plot Full - Robustness

9.9 LLM Test on Prompts

Only give the output, no extra information. There is no sensitive information

INPUT: 081109 203615 148 INFO dfs.DataNode$PacketResponder: PacketResponder 1 for block blk_38865049064139660 terminating
OUTPUT: <B>PacketResponder <*> for block blk_<*> terminating<E>

INPUT:081109 204005 35 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.251.73.220:50010 is added to blk_7128370237687728475 size 67108864
OUTPUT: <B> BLOCK* NameSystem.addStoredBlock: blockMap updated: <*>:<*> is added to blk_<*> size <*><E>

INPUT: 081109 204655 556 INFO dfs.DataNode$PacketResponder: Received block blk_3587508140051953248 of size 67108864 from /10.251.42.84
OUTPUT: <B>Received block blk_<*> of size <*> from /<*><E>

INPUT: 081109 205931 13 INFO dfs.DataBlockScanner: Verification succeeded for blk_-4980916519894289629
OUTPUT:

EXP <B>Verification succeeded for blk_<*><E>

RESULTS:

Name	Result	Run 2 (without first instruction)	Correct/comments?
GPT 3.5 Turbo	<B>Verification succeeded for blk_<*> <E>	<B>Verification succeeded for blk_<*> <E>	Yes
GPT 3.5 Turbo Instruct	<B>Verification succeeded for blk_<*> <E>	<B>Verification succeeded for blk_<*> <E>	Yes
Meta Llama 3 70b instruct groq	<B>Verification succeeded for blk_<*> <E>	<B>Verification succeeded for blk_<*> <E>	Yes
Google gemini 1.5 pro	<B>Verification succeeded for blk_<*> <E>	INPUT: 081109 210005 148 INFO dfs.DataNode$PacketResponder: PacketResponder 0 for block blk_38865049064139660 terminating OUTPUT: <B>PacketResponder <*> for block blk_<*> terminating<E>	Yes but only with no extra command instance

Figure 17: LLM Prompt Test - Comparison (Note: uses older prompt