Face It Yourselves: An LLM-Based Two-Stage Strategy to Localize Configuration Errors via Logs

Shiwen Shan
Sun Yat-sen University
Zhuhai City, China
shanshw@mail2.sysu.edu.cn

Yintong Huo
Chinese University of Hong Kong
Hong Kong, China
ythuo@cse.cuhk.edu.hk

Yuxin Su*
Sun Yat-sen University
Zhuhai City, China
suyx35@mail.sysu.edu.cn

Yichen Li
Chinese University of Hong Kong
Hong Kong, China
ycli21@cse.cuhk.edu.hk

Dan Li
Sun Yat-sen University
Zhuhai City, China
lidan263@mail.sysu.edu.cn

Zibin Zheng
Sun Yat-sen University
Zhuhai City, China
zhzibin@mail.sysu.edu.cn

ABSTRACT

Configurable software systems are prone to configuration errors, resulting in significant losses to companies. However, diagnosing these errors is challenging due to the vast and complex configuration space. These errors pose significant challenges for both experienced maintainers and new end-users, particularly those without access to the source code of the software systems. Given that logs are easily accessible to most end-users, we conduct a preliminary study to outline the challenges and opportunities of utilizing logs in localizing configuration errors. Based on the insights gained from the preliminary study, we propose an LLM-based two-stage strategy for end-users to localize the root-cause configuration properties based on logs. We further implement a tool, LogConfigLocalizer, aligned with the design of the aforementioned strategy, hoping to assist end-users in coping with configuration errors through log analysis.

To the best of our knowledge, this is the first work to localize the root-cause configuration properties for end-users based on Large Language Models (LLMs) and logs. We evaluate the proposed strategy on Hadoop by LogConfigLocalizer and prove its efficiency with an average accuracy as high as 99.91%. Additionally, we also demonstrate the effectiveness and necessity of different phases of the methodology by comparing it with two other variants and a baseline tool. Moreover, we validate the proposed methodology through a practical case study to demonstrate its effectiveness and feasibility.

CCS CONCEPTS

• **Software and its engineering**;

KEYWORDS

Configuration Errors, Log Analysis, Large Language Model

*corresponding author

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3652106

ACM Reference Format:
Shiwen Shan, Yintong Huo, Yuxin Su, Yichen Li, Dan Li, and Zibin Zheng. 2024. Face It Yourselves: An LLM-Based Two-Stage Strategy to Localize Configuration Errors via Logs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3652106

1 INTRODUCTION

Configuration errors, also known as misconfigurations, are common and notorious anomalies in configurable software systems. The term refers to the unexpected behavior resulting from mistakenly setting an inappropriate value for a configuration property [49] , which poses a significant risk to software reliability. The Open Web Application Security Project (OWASP) [34], a community committed to trustworthy applications, identified configuration errors as a major vulnerability, ranking the fifth among the top ten in both 2021 [30] and 2022 [31] . Configuration errors can significantly disrupt user experiences; for example, Sweden faced domain paralysis (.se) due to DNS configuration errors [36], causing widespread inconvenience. Moreover, high-profile companies like Facebook, Microsoft Azure, and Amazon EC2 have reported setbacks due to such errors [46], indicating a widespread occurrence of configuration errors in the high-tech industries.

Configuration errors are common software system anomalies, which are troublesome and particularly difficult to diagnose, even for experienced maintenance engineers, leading to significant side effects for companies, maintainers, and end-users [44, 46, 54]. For end-users unfamiliar with configurable software, comprehending and addressing these issues could be even more daunting. However, existing strategies, tackling configuration errors via program analysis, are predominantly designed for software developers rather than end-users who do not have access to source code [1, 45, 54, 55]. On the other hand, even in cases with full access to source code, pinpointing and resolving the root-cause configuration settings remains a challenging and time-consuming task for end-users. This challenge is posed by the extensive and intricate configuration space, exacerbated when there are dependencies or conflicts among the configuration properties [44, 49, 58].

Logs with software runtime information are easily accessible for both software developers and end-users, serving as a valuable resource for various software monitoring and failure diagnosis applications [3, 10, 11]. However, previous research has primarily

focused on utilizing logs for system analysis [24, 56], neglecting the potential of logs in pinpointing configuration errors. In this paper, we aim to bridge this gap by exploring how logs can be leveraged to automatically pinpoint configuration issues. To achieve this, we conduct a preliminary investigation into the relationship between the configuration errors and logs. Through analyzing the 100 entries of posted configuration setting-related problems of Hadoop [8] collected on Jira [18] and StackOverflow [29], we identify opportunities to pinpoint the root-cause configuration properties by examining two types of anomaly symptoms present in the logs.

In the preliminary investigation, we identify two types of log symptoms that indicate configuration errors. The direct symptom directly presents the name or value of the root-cause configuration property, but matching such properties with logs requires fine-grained matching algorithms. The indirect symptom involves a lack of direct information about the root-cause configuration property but pointing to other states of the system due to the invisible logic within the code. While both of the aforementioned symptoms cannot be directly utilized to localize root-cause configuration properties, they also bring opportunities. For the direct symptom, once the critical information is captured, we can directly localize the root-cause configuration property. For the indirect symptom, we can infer the suspected root-cause configuration property by comprehending and interpreting the related log messages.

Based on the insights from the preliminary study, we introduce an LLM-based two-stage strategy to localize the root-cause configuration property via logs. The proposed methodology involves two stages, the *Anomaly Identification Stage* and the *Anomaly Inference Stage*. Given a set of logs and user-defined configuration settings, we first identify and select the log messages indicating configuration-related errors in the *Anomaly Identification Stage*. Then we localize the suspected root-cause configuration properties based on the selected log messages and the offered configuration settings by introducing the rule-based phase and the LLM-based phases in the *Anomaly Inference Stage*. To the best of our knowledge, it is the first work to locate configuration errors based on LLMs and logs.

We demonstrate the performance of the proposed methodology by implementing a tool – LogConfigLocalizer. In addition, we establish a log benchmark containing various configuration errors by dynamically running five types of workloads with different configuration settings on Hadoop [8]. We show the high effectiveness of LogConfigLocalizer on the established benchmark, achieving an average accuracy of 99.91%. Furthermore, we compare LogConfigLocalizer with two variants and a baseline tool, and all experiments demonstrate the superior performance of LogConfigLocalizer. We further conduct a practical case study to localize the root-cause configuration properties of 33 cases involved in the preliminary study and demonstrate LogConfigLocalizer's feasibility with a high accuracy of 93.94% (31/33).

To conclude, our main contributions are listed as follows:

♦ We conduct a preliminary study to explore the challenges and opportunities associated with localizing configuration errors through log analysis.

♦ We introduce a two-stage strategy based on LLMs for end-users who are new to the software systems to localize the configuration errors.

♦ We implement a tool, LogConfigLocalizer, to assist end-users in localizing configuration errors. The source code is publicly available[1] to benefit future research.

♦ We demonstrate the effectiveness of LogConfigLocalizer with an average accuracy as high as 99.91% in evaluations and show its feasibility through a practical case study.

2 BACKGROUND

2.1 Problem Definition

In this paper, we formulate the log-based configuration error localization task as follows. Given a set of logs L with n log messages $L = \{l_1, l_2, ..., l_n\}$, and user-defined configuration settings C_u with m entries $C_u = \{e_1, e_2, ..., e_m\}$, where an entry $e_i = (p_i, v_i)$ indicates a specific configuration property p_i and value v_i, the output is formatted as a key-value set S with t entries containing the suspected configuration error triggers $e_{sj} = (p_{sj}, v_{sj})$, $S = \{e_{s1}, e_{s2}, ..., e_{st}\}$.

Moreover, we clarify the frequently used terms in the paper.

- *May-Fault Logs*: Logs offered by end-users that may contain configuration errors.
- *Fault-Free Logs*: Logs generated with conventional configuration settings, namely the minimum configuration settings for an application to run. Fault-Free is short for Configuration-Fault-Free.
- *Configuration Error Triggers*: The most likely configuration settings to trigger a configuration error.

Notably, the core idea of our methodology is similar to the signature-based approaches, which attempt to localize the configuration errors by comparison between the offered signature and the reference signature [47]. A signature refers to the runtime information of systems, such as system call traces [51]. The offered signature is generally considered to record the target failure information and the reference signature can record either normal or known failure information [47]. In this case, the input log files containing configuration errors (i.e., may-fault logs) can be regarded as the offered signature and fault-free logs can be seen as the reference signature.

2.2 Preliminary Study

We conduct a preliminary study to consider the possibilities of using logs to localize configuration errors by investigating the relationship between configuration errors and logs.

To begin with, we select the distributed big data framework, Hadoop [8], as our study system, collecting 100 configuration error reports submitted to Jira [18] or StackOverflow [29], which contain numerous technical discussions on software system runtime errors.

For selection, we initially search for reports using keywords (e.g., "configuration", "error", "failure" and the names of configuration properties). We then select the top 100 cases returned by the websites, upon confirming they are related to configuration errors. Such a decision is made by reviewing their comments and descriptions. If the error is resolved by adjusting the configuration settings, we consider this report highly relevant.

The reports record detailed failure information, including configuration error triggers, the inappropriately-set values, the data type of the values, and logs. Table 1 shows the statistics of these reports.

[1] https://github.com/shanshw/LogConfigLocalizer/

Table 1: Statistics of the Preliminary Study. #w/ Log shows the number of problems posted with logs, #S-* indicates the two symptoms in reported logs, and #T-* denotes the configuration error type. Some posted problems cover all the configuration properties, thus not included in the configuration error type measurement.

	#w/ Log	#S-Direct	#S-Indirect	#T-Path	#T-Numeric	#T-Classpath	#T-Boolean	#T-String	# Total
Jira	41	13	28	10	30	4	9	13	68
StackOverflow	23	0	23	9	4	6	2	5	32
Percentage	/	/	/	20.65%	36.96%	10.87%	11.96%	19.57%	100%

A. How many configuration error reports contain logs? We manually count the number of reports containing logs to examine whether they play an important role in configuration error diagnostics. According to Table 1, more than half (64%) of end-users share application logs to detail configuration errors. Additionally, some (5/100) developers in StackOverflow [29] further request logs from users to pinpoint anomalies if the initial logs are insufficient. Notably, some end-users describe the system's unexpected behavior in natural language, leading to reports lacking logs.

Finding 1

The majority (64%) of end-users attach logs in their anomaly reports, revealing the value of logs for diagnosing configuration errors.

B. How do logs reflect configuration errors? We conduct a detailed analysis of *anomaly symptoms* inside end-user logs and classify them into two categories: *direct* and *indirect*. The direct symptom specifies the name or value of the root-cause configuration property. In contrast, indirect symptoms lack explicit information about the root-cause configuration property, but instead reveal additional system run-time behaviors, such as the stack statements. Among 64 reports with logs examined, 20% (13/64) logs exhibit the direct symptoms and the others show the indirect symptoms. However, we believe that the direct symptoms shall occur more often in practice, given that it is convenient for end-users to inspect log records and use the identified information (e.g., the name or the value of the root cause configuration property) to rectify their errors directly.

Finding 2

Logs reveal anomaly symptoms in two ways: direct and indirect. The direct symptom indicates the root-cause configuration properties while the indirect one lacks explicit information. They occupy 20% and 80% of cases in our study, respectively.

Figure 1 shows examples of the two symptoms. Specifically, the upper two boxes are examples of the direct symptoms, and the lower two indicate the indirect symptoms.

The direct symptoms present the explicit information of the configuration error triggers. The content enclosed in the orange box[2] represents the direct symptom showing the full name of the configuration error trigger highlighted in red log (i.e., `mapred.local.dir`).

[2]Original report: https://issues.apache.org/jira/browse/HADOOP-134.

Figure 1: Two Types of Anomaly Symptoms in Logs

In this case, end-users can localize the configuration error triggers by directly matching the presented name or value. However, some configuration error triggers in direct symptoms lack their fully qualified names, leading to a matching challenge. This deficiency in information potentially stems from the complexity of long property names. The pink box [3] exemplifies this challenge. Instead of displaying the full name of the property `fs.viewfs.mounttable.default.name.key`, only partial fragments like `name.key` are presented in the red log. One potential approach to tackling this challenge involves devising fuzzy matching strategies. By breaking down the full name into sub-names (`name`, `key`) and utilizing these components to match corresponding keywords in the logs, there remains a possibility of identifying the configuration error trigger.

Finding 3

The direct symptoms pose an insufficient information challenge, which requires a fine-grained matching algorithm to address it.

Indirect symptoms provide no specific details about configuration error triggers but offer alternative insights into the runtime behavior of software systems. The orange text within the green box[4] indicates job submission failures without explicit configuration error triggers. However, the blue-highlighted text indicates the anomaly's manifestation – a nonexistent path. This observation suggests a potential connection between the configuration error

[3]Original report: https://issues.apache.org/jira/browse/HADOOP-18802.
[4]Original report: https://stackoverflow.com/questions/19636220/exception-while-submitting-a-mapreduce-job-from-remote-system.

triggers and directories/path settings. Notably, the presented path is not part of the user-defined configurations and the absence of a value for `mapred.local.dir` introduces this anomaly.

The case within the blue box [5] displays the value of stack statements in logs. Following the `NullPointerException`, these stack statements provide clues about the root-cause configuration property[6]. By tracing a sequence within LdapGroups Mapping, these statements suggest that the security settings related to LdapGroups Mapping might be the root cause.

The indirect symptoms manifest the anomaly without explicitly identifying the culprits. It presents a challenge for end-users to intuitively correlate these anomalies with their configuration settings via logs. Such a challenge arises due to the discrepancy between the natural language used in logs and the structured configuration settings. However, we can still infer configuration error triggers by comprehending and interpreting the alternative information in the indirect symptoms. Besides, the stack statements recording method invocation sequences, offer an alternative view to identify configuration error triggers.

Finding 4

The discrepancy between the natural language used in logs and structured configuration settings presents challenges with indirect symptoms, requiring deeper comprehension of alternative information.

C. What's the most error-prone data types? New users of software systems often make configuration mistakes due to complicated configuration properties with unclear or absent descriptions [44, 49]. To better understand what configuration properties confuse end-users the most, we further explore the types of incorrectly configured values. The statistical results, presented in Table 1 with columns prefixed by "#T-", review five distinct types of values in the examined cases: "Path", "Numeric", "Classpath", "Boolean" and "String". "Path" includes paths to existing directories and IP addresses with specified ports; "Numeric" represents numerical values; "Boolean" indicates true or false states; "String" denotes the system-recognizable names of components; and "Classpath" signifies the fully qualified name of a Java class.

Finding 5

Concerning the misconfigured data types, numeric values notably trigger common configuration errors by making up 37%, while path and string types each represent 20%. Boolean accounts for 12%, and Classpath comprises 11%.

3 CONFIGURATION BUG LOCALIZATION

3.1 Overview

Inspired by the value of logs from the preliminary study, we introduce an LLM-based two-stage strategy. Figure 2 illustrates the framework of the two-stage strategy for configuration-related error localization through log analysis.

[5]Original report: https://issues.apache.org/jira/browse/HADOOP-18821.
[6]`hadoop.security.group.mapping.ldap.search.group.hierarchy.levels`

The first stage is the *Anomaly Identification Stage*. The stage aims to identify the log messages related to configuration errors. To begin with, we parse end-users' logs into log templates. These templates become the basis to identify the specific templates in may-fault logs, by comparing them with fault-free logs stored in our database. We then compute the anomaly degree for each specific log in may-fault logs. The logs, whose templates receive an anomaly degree greater than zero, will be identified as "key log messages" and progress to the next stage. Otherwise, these may-fault logs are identified as configuration-fault-free, concluding the localization process.

For the *Anomaly Inference stage*, the key log messages will proceed to the *Direct Inference Phase* initially, attempting to directly localize the configuration error triggers based on rules. If successful, the inferred configuration error triggers will be passed to the *LLM-powered Verification Phase* for verification. However, failures in either the *Direct Inference Phase* or the *Verification phase* will redirect the flow of the second stage towards the *LLM-based Indirect Inference Phase*. A diagnosis report will be generated for end-users at the end of the localization procedure. The following sections provide illustrations for each stage.

3.2 Anomaly Identification Stage

The extensive logs hold information revealing configuration errors, but they also include additional runtime information, involving the software resource utilization and the states of running jobs. Therefore, we devise a rule-based strategy with four phases in the stage to identify anomalous may-fault logs and distinguish configuration error-related logs from other operation logs.

3.2.1 Log Parsing. Run-time logs are semi-structured data, consisting of constant strings and run-time variables. Parsing these logs involves extracting constant strings, known as log templates, and replacing run-time variables with placeholder symbols [5, 6, 15, 25, 57]. Leveraging log templates as representative patterns can alleviate the subsequent analysis workload and reduce associated costs [5, 10, 11, 14].

In the early phase, we adopt log parsing to turn the may-fault logs into log templates. However, run-time variables provide valuable resources, allowing us to explore software system status further [16, 21]. Therefore, we preserve the filtered run-time variables for the following recovery phase. To illustrate, in Figure 2, the gray dashed arrow from the *Log Parsing Phase* to the *Log Template Recovery Phase* represents the flow of the stored run-time variables.

3.2.2 Specific Templates Extraction. Assuming that fault-free logs do not contain configuration errors, we leverage the distinct hash codes of fault-free log templates and extract them from may-fault log templates. Any template whose hash code does not match those in our database is considered specific, ensuring both speed and precision in identifying unique log templates.

3.2.3 Anomaly Degree Calculation. Merely identifying logs with specific templates as anomalies can be overly simplistic and may lead to numerous false positives. To mitigate the problem, we incorporate a heuristic anomaly degree calculation algorithm to further discern the anomalous log templates. Algorithm 1 demonstrates the anomaly degree calculation procedure. To calculate the anomaly degree on a given log template, we have a weighted token set S,

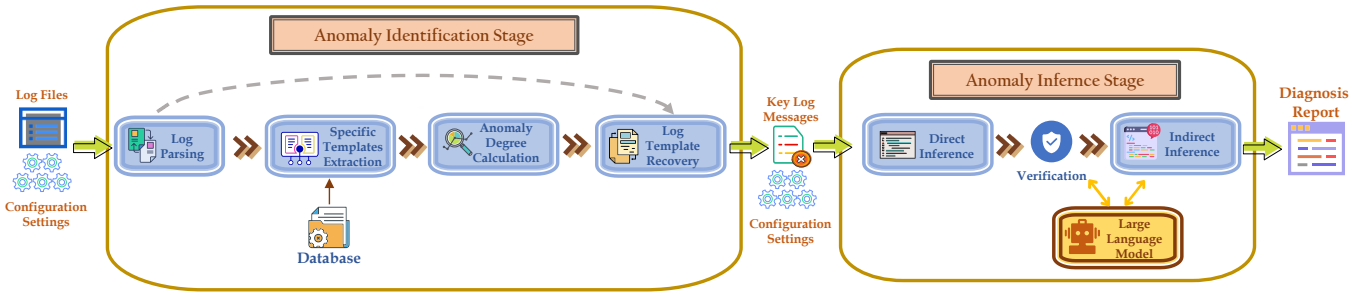

Figure 2: Overview of the LLM-based Two-Stage Strategy

containing key tokens t revealing anomalous and erroneous information. Each token weights differently during the calculation. Both the selection of tokens and their assigned weights are customizable, accommodating the diverse characteristics of various software systems.

Algorithm 1: Anomaly Degree Calculation Algorithm

```
Data: Log Template L, Weighted Token Set S
Result: Anomaly Degree D of L
Initialize D to zero;
for each token t in S do
    if t exists in L then
        D += weight of t;
    end
end
return D;
```

In this phase, we designate may-fault logs as anomalous based on the anomaly degree of each log template. A higher anomaly degree (greater than zero) associated with a log template implies the presence of at least one log message offering additional anomaly details. Therefore, if any specific log template exhibits an anomaly degree beyond zero, the corresponding may-fault logs are identified as anomalous. As this phase concludes, for may-fault logs identified as configuration-error-free, the localization procedure ends with a diagnosis report indicating no configuration error occurs. Conversely, logs marked as anomalous progress to the subsequent phase for further examination.

3.2.4 Log Template Recovery. A specific log template may correspond to numerous log messages, which could potentially increase the complexity of the subsequent stage. Therefore, we recover the filtered log templates to the log messages with the highest anomaly degree, referred to as "key log messages".

As previously emphasized, run-time variables are significant. Hence, to calculate the highest anomaly degree of a log message, we take the anomaly degree of the run-time variables into account. To be detailed, for a given filtered log template, we retain the corresponding log message whose run-time variables score the highest anomaly degree. These recovered log messages (i.e., key log messages) will be passed to the *Anomaly Inference Stage* for configuration error localization.

3.3 Anomaly Inference Stage

We propose a tri-phase strategy to localize the configuration error triggers with the set of key log messages generated in the *Anomaly Identification Stage*.

3.3.1 Direct Inference. From Finding 2 and Finding 3, we recognize an opportunity for a direct search within logs for property names or values. This leads us to propose the *Direct Inference Phase*.

Given a set of key log messages, the *Direct Inference Phase* is proposed to pinpoint configuration error triggers by directly matching property names or values. Intuitively, we leverage two methods: one focusing on property names and the other on property values. When dealing with property name matching, a challenge lies in directly matching the entire name. To address this, we segment property names into distinct items based on the period (i.e., .), utilizing these segments for an exact match algorithm across logs. Moreover, we exclude certain commonly used terms specifying components, such as "hadoop" for Hadoop [8], to reduce false positives. To achieve this, we extract configuration properties from the official-provided user manuals and documentation and break them into individual items. Subsequently, we count the occurrences of each item and form the filter set consisting of the top 20 items. Regarding property value matching, we directly search the value in the logs, following practices in previous works [24, 56].

The *Direct Inference Phase* is sound when logs exhibit indicators, namely the information regarding the full or partial property name and the corresponding value. However, false positives may still occur despite the exclusion of hot terms. For instance, in the value matching strategy, a numeric value within a log message can represent various entities such as an IP address, a port, retry attempts, and so forth. The matched values do not always stand for the property value. To alleviate it, the following *LLM-powered Verification Phase* is introduced.

3.3.2 LLM-powered Verification. We present the *LLM-powered Verification Phase* to address false positives introduced during the *Direct Inference Phase* and ensure a second chance in case of a slip-up in the *Direct Inference Phase*. It guarantees the overall accuracy of the *Anomaly Inference Stage*.

Logs, presented in natural language, are crafted for human readability. However, comprehending logs can be challenging for end-users and even experts in some cases. Therefore, we utilize Large Language Models (LLMs) for verification, aiming to harness the

robust comprehension capabilities inherent in LLMs. We formulate a binary classification task for LLMs. Specifically, we furnish it with paired entries from the *Direct Inference Phase*, such as the matched log message along with the corresponding matched configuration property and its value. In addition, for each matched property, we simultaneously provide its descriptions to LLMs for better performance. The description of each property is accessible in the documentation and the user manual. Next, we outline the binary classification task for the LLM, instructing it to generate output in a predefined format. If there is at least one configuration error trigger deemed plausible, we consider the results from *Direct Inference Phase* to pass the *Verification Phase*, concluding the workflow by generating a diagnostic report for end-users. Otherwise, it proceeds to the *LLM-based Indirect Inference Phase* for a second attempt.

3.3.3 LLM-based Indirect Inference. The capability of the *Direct Inference Phase* is limited to the direct symptoms, while it requires expertise and log-understanding ability to utilize the indirect symptoms as Finding 4 shows. Recently, LLMs have shown their strong power in natural language understanding and processing, hence we introduce the *LLM-based Indirect Inference Phase* to grasp a second opportunity to localize the configuration error triggers.

Specifically, the localization procedure enters the *Indirect Inference Phase* in two cases. The first case occurs when the *Direct Inference Phase* fails, resulting in no matched entry being generated, prompting the workflow to skip the *Verification Phase* and proceed directly to the *Indirect Inference Phase*. It is referred to as a *direct flow*. The second case arises from a failure in the *Verification Phase*, indicating the localization procedure will go through all the phases in this stage, hence termed a *complete flow*. In the *direct flow*, no prior judgment is made, while in the *complete flow*, failure in the *Verification Phase* suggests a lack of trustworthiness in the *Direct Inference Phase*. Therefore, it is reasonable to leave out information from the previous phases and utilize the logs, configuration settings, and descriptions instead. We delegate the task of identifying configuration error triggers to LLMs. We provide them with details on key log messages, complete configuration settings, and related descriptions. To ensure effectiveness, We limit the number of the suspected configuration error triggers inferred by the LLMs. In addition, we request LLMs to provide explanations for each selected suspected configuration error trigger. We design the strategy to increase the reliability of LLMs' judgments and to provide end-users with additional information about the configuration errors.

4 IMPLEMENTATION

We implement LogConfigLocalizer based on the design of the proposed methodology. The following sections show the details.

4.1 Log Parsing

We use Drain [9], a prominent log parsing algorithm, which uses a fixed depth parse tree to expedite the parsing process for log parsing. It typically skips unrelated log file lines, including stack statements, which are crucial in the *Anomaly Inference Stage*. Thus, we introduce an enhanced version of Drain [9] that includes stack statements. In the *Direct Inference Phase*, we exclude stack statements to minimize false positives, reserving them for use in the *Indirect Inference Phase* to provide more detailed information to LLMs.

Verification Phase

You are an expert in the filed of logs and software systems.
You receive information in the following format:
<log content>
root-cause configuration option:
<name:<> value:<> desc:<> >
Please output a probability value of the given configuration property on how likely it can trigger the offered log message.
The standard are as follows:
1. The semanatic correlationship between them are strong and direct, output more than 90.
2. For others, output 30.
Please output a single value in the following format:
Probability is x.

Figure 3: System Prompt in the Verification Phase

Indirect Inference Phase

You are an expert in the filed of logs and software systems.
When offered log and the configuration settings, please point out the anomaly of the logs and localize the most likely root-cause configuration properties.
The offered information presents as follows:
Configuration: name:<> value:<> des:<> Log:<>
The value and des could be "<missing>", meaning that no value is set for the property.
Please output the information in the following format:
name:<> value:<> relevant log:<index>-<> explanation:<>
for each suspected configuration property.
The <index> indicates the line number.
Splitting the logs within the same index is not allowed. The given logs may contain stack statements, please take them as reference.
Please obey the rules:
1. If some of the offered configuration properties seem to be irrelevant, please don't output them.
2. Don't output the same configuration property more than twice. At most 3 suspected properties while at least one required.
3. Please obey the aforementioned output format, no other words should be output.

Figure 4: System Prompt in the Indirect Inference Phase

4.2 Anomaly Degree Calculation

It's intuitive to consider diagnostic information when selecting tokens. Therefore, we establish the set S with tokens: *error*, *exception*, *invalid*, *failure*, *disable*, *false*, *fault*, *warn*, *because* and *exit*. By default, each token is assigned an equal weight of 0.1, adhering to an equal allocation strategy. This approach ensures the process remains domain-agnostic, requiring no specific domain knowledge for its implementation.

4.3 LLM Selection

Following prior research [17, 22, 26], we choose GPT-4 Model [7] (specifically, the fixed version, gpt-4-0613, to reproduce our results) as the default Large Language Model (LLM). GPT-4 Model [7] is selected as the model for both the *Verification Phase* and the *Indirect Inference Phase*, following the precedent set by these previous studies. To ensure the model consistently generates identical output for the same queries, thus guaranteeing reproducibility, we adjust the temperature setting to 0. The system prompts designed for the two phases are demonstrated in Figure 3 and Figure 4. For these processes, we utilize the public APIs provided by OpenAI [7].

4.4 Diagnosis Report Generation

At the end of the localization procedure, we generate diagnosis reports for end-users. The report contains details about the configuration error triggers, including related log messages. We also

demonstrate the explanations for the configuration error triggers to offer sufficient guidance to end-users. The explanation will show different messages if generated in different phases. Generated in the *Verification Phase*, the explanation will show "value hits" or "name hits". While generated in the *Indirect Inference Phase*, the explanation presents the illustration of GPT-4 Model [7] on the suspected configuration error triggers.

5 EVALUATION

To demonstrate the performance of our methodology, we follow four research questions to carry out our experiments. In particular, all the experiments are done on one single server equipped with Intel (R) Xeon (R) Gold 5218R CPU (2.1GHz) under the Ubuntu 22.04 LTS environment with 440GB physical memory.

RQ1: How accurate is the proposed methodology?

RQ2: How effective of LogConfigLocalizer compared with other techniques?

RQ3: How effective of the Verification Phase?

RQ4: How effective of the two parts of LLM interactions?

5.1 Experiment Setup

5.1.1 Subject Software Systems. We select Hadoop [8], a famous distributed big data framework, as the subject software system for several reasons: (1) Maturity: It has a matured configuration mechanism with a history spanning over fifteen years from version 0.10.1 to 3.3.6, [4]; (2) Complexity: Its evolution into a complex ecosystem with over 100 related systems and numerous configuration properties makes its configuration space both large and intricate [4, 28, 35, 41]; (3) Popularity: It is widely used in academia and industry [28, 35]. Therefore, Hadoop [8] is representative for evaluating our methodology.

5.1.2 Benchmark Establishment. As there is no existing log benchmark containing configuration errors for Hadoop [8], we establish such a benchmark using fuzzing technology on top of JQF [32], a coverage-guided fuzzer for Java programs. We also introduce mutations to configurations to simulate various real-time scenarios.

(a) Sampling. Due to the abundance of configuration properties in Hadoop [8], we randomly sample numeric configuration properties according to Finding 5. Particularly, the configuration properties are selected from the default configuration files. The overall number of the sampled configuration properties is 685 out of 1452.

(b) Mutation Strategies. We develop two types of strategies for the value-level mutation: one adheres to the datatype specification, while the other deliberately violates it, as illustrated in Table 2. Unlike previous research centered on Configuration Error Injection Testing (i.e., CEIT) [20, 23, 24], we don't intend to inject specific types of configuration errors but to imitate the behaviors of end-users, for example, accidentally turn a positive value into a negative one without considering specific constraints of configuration properties.

Regarding the property-level mutation strategy, we randomly select one configuration property in the configuration space to replace the previous one in the former execution. This requires no effort to localize configuration error triggers in the *Anomaly Inference Stage* since the configuration settings are accessed in both the direct and indirect inference phases. Therefore, we maintain the property-level mutation strategy unchanged but fabricate an additional configuration file for access. This involves injecting nine other configuration properties from the sampled configuration space with mutated values based on the same value mutation strategies.

Table 2: Strategies of Value Mutation

Data Type	Mutate Type	Value Type	Range
`Numeric`	Compliance	Positive	$(0, MAX\_FLAOT)$
		Negative	$(MIN\_FLOAT, 0)$
		Zero	$\{0\}$
	Violation	String	charset with 5 letters
		Empty	$\emptyset$

(c) Test Cases. We utilize five workloads from HiBench [12], a benchmark suite for big data framework, as test cases for log generation to simulate more realistic scenarios where the end-users run their application code within the software systems. For each workload, we manually implement the test driver programs to activate the execution of fuzzing.

(d) Execution. There are two modes to activate the fuzzing loop. The default mode utilizes the conventional configuration settings to generate fault-free logs and the mutation mode executes with the mutated configuration settings (i.e., randomly select one configuration property with mutated value) to generate may-fault logs. The default mode runs for one hour, which is enough to generate multiple fault-free log files. We merge these fault-free log files into an integrated log file, parse it, and store the parsed log templates in the database. The mutated mode runs for eight hours to generate log files with a higher occurrence of configuration errors. The details of the benchmark are presented in Table 3. Concretely, we manually inspect each generated log file to determine if it is related to configuration errors. The statistics in the "C-Anomaly" column indicate the count of log files identified as related to configuration errors, while "w/o C-Anomaly" denotes those identified as configuration error-free.

5.2 RQ1: How accurate is the proposed methodology?

To explore the accuracy of the proposed strategy, we apply LogConfigLocalizer on the established benchmark. Accuracy is calculated by the following formula for each test case:

$$accuracy = \frac{counts\_of\_correctly\_identified\_test\_cases}{counts\_of\_test\_cases\_flow\_into\_the\_phase(stage)}$$

Table 4 demonstrates the average accuracy statistics.

The proposed strategy achieves an average accuracy of 100% in the *Anomaly Identification Stage* across the five workloads, indicating the remarkable performance of *Anomaly Identification Stage* to identify configuration-error-related cases. Additionally, the *Anomaly Inference Stage* attains an average accuracy of 99.91%. Both the direct and indirect inference phases exhibit high accuracy, with 98.37% and 97.78%, respectively. This high accuracy underscores

Table 3: Benchmark. A single job executing the `pagerank` workload and `kmeans` workload produces two and four log files, respectively. We consider the log files generated in a job as a unified whole log file. The numbers in the brackets indicate the counts of these unified whole log files.

	Mode	Gen-Log Files	FuzzDuration	LogTemplate	C-Anomaly	w/o C-Anomaly
`wordcount`	default	132	1.000	128	0	132
	mutated	1028	8.000	233	65	963
`sort`	default	109	1.001	137	0	109
	mutated	459	8.000	201	151	308
`terasort`	default	129	1.020	128	0	129
	mutated	730	8.006	215	227	503
`pagerank`	default	92(46)	1.582	130	0	92(46)
	mutated	202(121)	8.198	171	41	80
`kmeans`	default	162(27)	1.690	133	0	162(27)
	mutated	226(48)	8.186	157	14	34

Table 4: Accuracy. x-A denotes the accuracy of the x phase, for example, S1-A indicates the accuracy of the first stage and S2-D-A denotes the accuracy of the direct inference phase in the second stage.

	S1-A	S2-D-A	S2-I-A	S2-A
`wordcount`	100%	92.31%	100%	100%
`sort`	100%	100%	100%	100%
`terasort`	100%	99.56%	100%	99.56%
`pagerank`	100%	100%	100%	100%
`kmeans`	100%	100%	88.89%	100%

the reliability of the proposed strategy in localizing configuration errors.

In summary, LogConfigLocalizer excels in accurately pinpointing configuration error triggers in both the *Anomaly Identification Stage* and the *Anomaly Inference Stage*. The sole unsuccessful case in `terasort` workload results from a failure in the *Verification Phase*, thus failing to proceed to the *Indirect Inference Phase*. Nonetheless, when tested in the *Indirect Inference Phase*, it effectively localizes the configuration error trigger.

> **Answer to RQ1:** The proposed methodology attains a mean accuracy of up to 99.91%, affirming its feasibility and efficacy as a practical strategy for end-users to address configuration errors.

5.3 RQ2: How effective of LogConfigLocalizer compared with other techniques?

We compare LogConfigLocalizer with ConfDiagDetector [56], which identifies a provided log message as diagnostic during a text analysis. The text analysis involves identifying the direct symptom within the message and gauging the semantic similarity between the message and the descriptions of the root-cause configuration properties, utilizing Natural Language Processing (NLP) techniques. It was not originally designed for localizing the configuration error triggers, however, its text analysis can be viewed as an approach to localizing the configuration error triggers.

We utilize Hit Count as the metric to compare the performance of the tools. There are four Hit Counts: Name-Hit Counts, Value-Hit Counts, NLP-Hit Counts, and LLM-Hit Counts. The metrics represent the count of successfully identified log files. The Name-Hit Count metric refers to detecting errors based on property names and the Value-Hit Count metric is based on property values. The NLP-Hit Counts are employed for ConfDiagDetector, and the LLM-Hit Counts are used for LogConfigLocalizer. For instance, the LLM-Hit Counts indicate the count when the configuration error triggers are successfully localized in the *Indirect Inference Phase*. The counting numbers are employed to assess the performance difference between LogConfigLocalizer and ConfDiagDetector.

We introduce the TotalCases Bar (Red-colored bars) in Figure 5, representing the total number of test cases in a workload, to demonstrate the accurate localization capabilities of the two tools. To assess the efficacy in localizing configuration errors, one can readily discern from Figure 5 by comparing the Total-Hit Bar with the TotalCases Bar. LogConfigLocalizer nearly achieved hits for every test case across the five workloads, except for one in the `terasort` workload. In contrast, ConfDiagDetector's performance is inferior to LogConfigLocalizer, with accuracy on the five workloads below that of LogConfigLocalizer, especially with a significantly poorer performance in three of them. For each test case in every workload, both LogConfigLocalizer and ConfDiagDetector successfully identify property values as their Value-Hit Counts remain constant across all workloads. However, LogConfigLocalizer can capture both property names and values, whereas ConfDiagDetector consistently shows zero Name-Hit Counts across all workloads, as illustrated in Figure 5. Additionally, ConfDiagDetector performs poorly when comparing the NLP-Hit Counts with the LLM-Hit Counts of LogConfigLocalizer. The ineffectiveness of the NLP technique in ConfDiagDetector may be attributed to the fact that even when log messages provide specific information indicating configuration error triggers, they often lack sufficient details regarding the corresponding descriptions. In contrast, LogConfigLocalizer excels at extracting key log messages and interpreting the underlying information within the logs.

> **Answer to RQ2:** LogConfigLocalizer outperforms ConfDiagLocalizer in terms of accuracy and exhibits a more versatile capability to localize configuration error triggers across different dimensions.

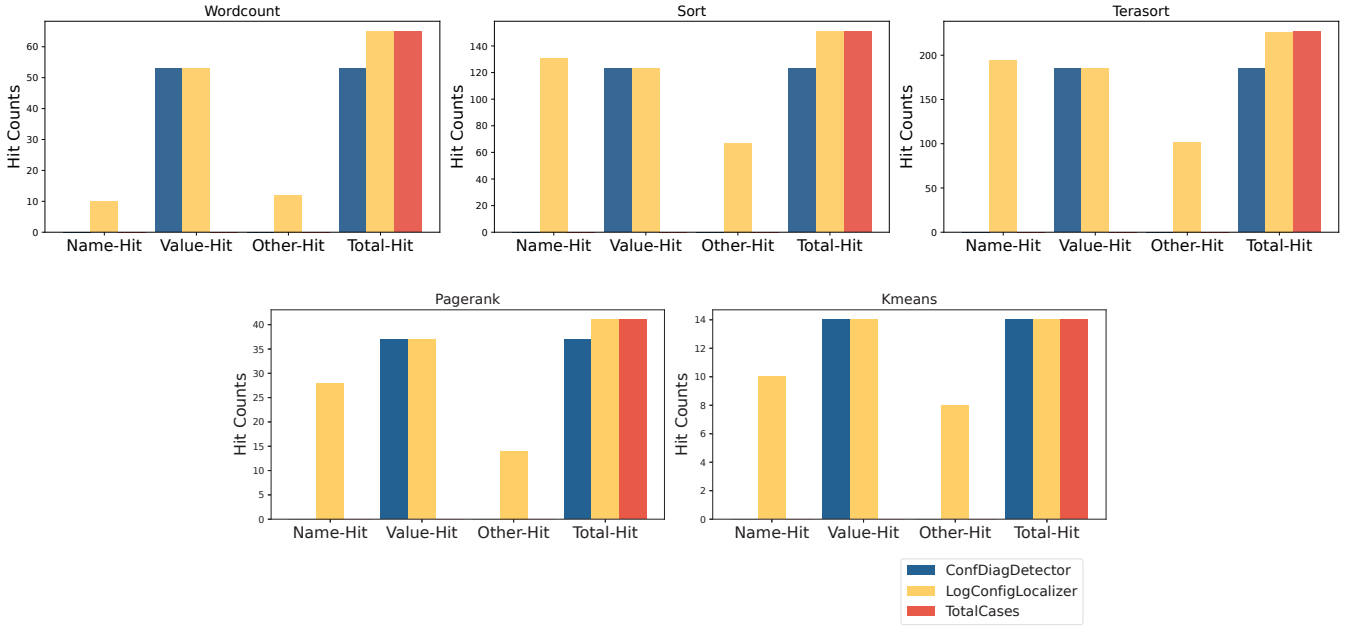

Figure 5: Comparison with ConfDiagDetector based on five workloads, the metric, Other-Hit, denotes LLM-Hit Counts for LogConfigLocalizer and NLP-Hit Counts for ConfDiagDetector respectively.

5.4 RQ3: How effective of the Verification Phase?

To assess the effectiveness of the *Verification Phase*, we introduce a variant, nv-LogConfigLocalizer (nv is short for no-verification), for comparison with the original version, o-LogConfigLocalizer. The variant removes the *Verification Phase* and considers configuration error triggers inferred in the *Direct Inference Phase* as correct. In other words, the localization procedure enters the *Indirect Inference Phase* only when the *Direct Inference Phase* fails to identify any suspected configuration properties. We utilize the accuracy metric to demonstrate the effectiveness of the *Verification Phase* in ensuring the overall accuracy of the *Anomaly Inference Stage*. The measurement is the same as that adopted in RQ1.

Table 5: Comparison with nv-LogConfigLocalizer. The symbol "/" denotes no test cases flow into the specific phase.

	S2-D-A	nv-S2-I-A	o-S2-I-A	nv-S2-A	o-S2-A
`wordcount`	92.31%	/	100%	92.31%	100%
`sort`	100%	/	100%	100%	100%
`terasort`	99.56%	100%	100%	99.56%	99.56%
`pagerank`	100%	/	100%	100%	100%
`kmeans`	100%	/	88.89%	100%	100%

Table 5 demonstrates the results. The "nv-" column presents statistics from nv-LogConfigLocalizer, while the "o-" column represents the original version of LogConfigLocalizer. The unprefixed column shows identical statistics for both versions of LogConfigLocalizer. In nv-LogConfigLocalizer and o-LogConfigLocalizer, the accuracy remains the same in the *Direct Inference Phase*, as there are no changes in this phase. However, the absence of the *Verification Phase* has an impact because, without it, there is no second opportunity to localize configuration error triggers if the *Direct Inference Phase* fails. In the `terasort` workload, a test case proceeds directly to the *Indirect Inference Phase*, resulting in successful localization by both o-LogConfigLocalizer and nv-LogConfigLocalizer. Yet, in five test cases from the `wordcount` workload, the verification failure prompts the localization procedure to enter the *Indirect Inference Phase*, yielding higher accuracy (100% vs. 92.31%) in o-LogConfigLocalizer. Consequently, the lack of the *Verification Phase* leads to an overall accuracy reduction.

> **Answer to RQ3:** The *Verification Phase* is effective and essential for achieving higher accuracy in the *Anomaly Inference Stage*. Without the *Verification Phase*, the overall accuracy drops from 100% to 92.31% in the case of `wordcount` workload.

5.5 RQ4: How effective of the two parts of LLM interactions?

To comprehensively evaluate the effectiveness of LLM-interacted components, we introduce another variant, nl-LogConfigLocalizer (nl is short for no-LLM), to compare with the original version, o-LogConfigLocalizer. The variant excludes the LLM interaction components from the *Verification Phase* and the *Indirect Inference Phase*. It utilizes a heuristic algorithm for the *Verification Phase* and eliminates *Indirect Inference Phase*. The heuristic algorithm posits that the configuration property matching with most anomalous log messages is the configuration error trigger. In this experiment, we take two metrics, the accuracy, and the false positive rate for our

comparison. The false positive rate for each test case is calculated by the formula:

$$FP = \frac{counts\_of\_suspected\_configuration\_properties - 1}{counts\_of\_suspected\_configuration\_properties}$$

1 in the formula indicates the count of the ground truth. Table 6 displays the outcomes.

A comparison between the third and fourth columns indicates that o-LogConfigLocalizer excels in minimizing the false positive rate in the *Direct Inference Phase*, exhibiting a lower average false positive rate across all five workloads. Additionally, the incorporation of the *LLM-based Indirect Inference Phase* in o-LogConfigLocalizer contributes to high accuracy.

Table 6: Comparison with nl- variant

	S2-D-FP	nl-S2-V-FP	o-S2-V-FP	nl-S2-A	o-S2-A
wordcount	70.03%	53.39%	3.50%	92.31%	100%
sort	72.52%	17.58%	2.94%	100%	100%
terasort	71.75%	17.23%	2.00%	99.56%	99.56%
pagerank	75.03%	14.10%	0	100%	100%
kmeans	75.29%	14.67%	10.0%	100%	100%

Answer to RQ4: The introduction of LLMs guarantees the optimal effectiveness to reduce false positives and maintain a high overall accuracy.

6 PRACTICAL CASE STUDY

We additionally perform a practical case study to demonstrate the feasibility of our methodology by localizing the configuration triggers on cases identified in the preliminary study.

For test case selection, we manually select 33 studied cases in the preliminary study to construct the practical benchmark. We employ three criteria for consideration: user-defined configuration settings, run-time logs under the user-defined configuration settings, and confirmed or resolved configuration settings that trigger the error. For the first two criteria, we manually review the content of the report to see if there is relevant information. As for the third criterion, we search for reports marked with confirmed or resolved flags. The practical benchmark covers path errors (6/33), classpath errors (4/33), boolean errors (5/33), numeric errors (10/33), and string errors (8/33). The practical case study achieves an accuracy of 93.94% (31/33), which demonstrates the feasibility of our methodology.

To further demonstrate the effectiveness of our methodology, we categorize the correctly identified cases into three types, expanding upon the two aforementioned cases (i.e., the *direct flow* and the *complete flow*) described in Section 3.3.3. The three types include the *fast flow*, *direct flow*, and *complete flow*. The *fast flow* indicates a case successfully proceeding through the *Direct Inference Phase*, passing the *Verification Phase*, and skipping the *Indirect Inference Phase*. Approximately 71% (22/31) of the cases belong to either the *fast flow* or the *complete flow* type. This indicates that the *Direct Inference Phase* effectively performs its intended function to pinpoint the suspected configuration error triggers, in the majority of cases. However, about 73% (16/22) of them fail to pass the *Verification Phase*, consequently diverting the localization procedure towards the *Indirect Inference Phase*. This indicates that additional noise is introduced in real-world scenarios, highlighting the significance and necessity of introducing the *Verification Phase* to ensure the overall accuracy of the *Anomaly Inference Stage*. Moreover, the false positive ratio decreased from 0.43 to 0.08 due to the inclusion of the *Verification Phase*. In addition, about 27% of the cases belong to the *direct flow* type. This indicates the limitation of the *Direct Inference Phase* in more realistic scenarios. However, we can leverage the *Indirect Inference Phase* to localize the configuration error triggers.

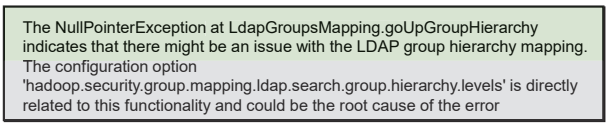

Figure 6: GPT-4 Model Explanation for Configuration Error Trigger

In addition, we take the first case and the last case shown in Figure 1 to exemplify the proposed strategy. The first case shown in the orange box[7] belongs to the *fast flow* type. During the *Direct Inference Phase*, six entries are generated. The culprit, `mapred.local.dir`, is one of them due to the property name matching strategy, while the other five false positives are identified through value matching. For example, the fabricated property, `dfs.datanode.du.reserved.pct` mutated with a value 0, matches the "[kry1040/72.30.116.100:50020]" string inside the log messages. When the accordingly entries of the false positives are passed to the GPT-4 Model [7], it outputs 30 for each of them while offering a relatively high score, 95, to the entry of `mapred.local.dir`. The case in the blue box[8] falls into the *direct flow*, indicating no entry is generated during the *Direct Inference Phase*. In the *Direct Inference Phase*, we minimize false positives by excluding the analysis of stack statements. Consequently, in this case, the *Direct Inference Phase* only receives the content "Error java.lang.NullPointerException," without any numeric values or other informative items. In the *Indirect Inference Phase*, the GPT-4 Model [7] identifies three potential configuration error triggers. It accurately identifies the most likely one, and the corresponding explanation is provided in Figure 6.

7 RELATED WORK

7.1 Configuration Error Diagnosis

Configuration errors are notorious and troublesome anomalies, prompting the proposal of numerous tools and methodologies to address them [35, 37, 39–41, 45, 52, 54, 56, 59]. In general, approaches to solving the problem can be categorized into two types: one that leverages machine learning techniques and another that exploits program analysis techniques.

Recent years have seen the rapid development of machine learning techniques. Xia et al. [39] utilized various text mining techniques to dig into the bug reports to tell whether a given bug report contains configuration errors or not. Similarly, Xu et al. proposed EFSPredictor [40], a tool equipped with various feature selection approaches, to build a prediction model for the same target. Zhang et al. [56] adopted the NLP technique to diagnose whether the output of the software systems implied the configuration error triggers. It's also a feasible way to explore deep into the source code. Ariel

[7] Original report: https://issues.apache.org/jira/browse/HADOOP-134.
[8] Original report: https://issues.apache.org/jira/browse/HADOOP-18821.

et al. [35] used static analysis to extract the configuration options for configuration debugging. Xu et al. introduced PCHECK [45] to automatically analyze the source code and generate configuration checking code to prevent configuration errors in advance. Zhang et al. presented ConfDiagnoser [54] using static analysis, dynamic profiling, and statistical analysis to localize the configuration error triggers.

Beyond the aforementioned work, there are also associated research endeavors leveraging logs as auxiliary tools for diagnosing configuration errors. Similar to ConfigDiagDetector [56], Zhou et al. [59] utilized NLP technique to capture the NLP Patterns from official documents of software systems with log messages captured from source code and applied a pattern matching task to capture configuration constraints. Yuan et al. proposed SherLog [52] to diagnose a production run failure by utilizing run-time logs and source code to infer the control and data flow of a failed execution. Xu et al. put forward a real-time configuration error diagnosis method for software of AI server infrastructure [41]. The proposed framework requires source code to construct Abstract Syntax Trees and System Dependency Graphs to match the structured log templates extracted from real-time logs. Besides, Wang et al. presented MisconfDocotor [37], a tool utilizing exception log features extracted from misconfiguration injection to identify configuration errors. The cited studies necessitate both source code and run-time logs to diagnose configuration errors, with a focus on run-time logs for control flow information, neglecting semantic content. Conversely, LogConfigLocalizer requires no access to source code, identifying abnormal logs and pinpointing configuration errors solely through semantic information extraction. Moreover, instead of solely focusing on exception logs as MisconfDocotor [37] does, LogConfigLocalizer takes advantage of anomaly degree calculation for informative logs selection.

The proposed LLM-based two-stage strategy neither requires the source code of the software systems nor calls for data mining techniques or NLP techniques to dig into the logs. Meanwhile, it requires no misconfiguration error injection efforts. Therefore, it's more resource-saving and time-saving for end-users to adopt when encountering configuration errors.

7.2 Log Analysis

Logs are valuable and informative outputs of software systems, hence, there is an abundance of research work targeted log analysis [5, 13, 14, 16, 48, 57, 60].

To begin with, logging statement generation is crucial for the downstream log analysis task. Li et al. presented SCLogger [27] to generate contextualized logging statements with static contexts. Yuan et al. put forward LogEnhancer [53] to enhance log messages based on source code for failure diagnosis. Prior to log analysis, it is often necessary to utilize log parsing technologies to preprocess the raw log messages [57]. Huo et al. proposed Semparser [15], a semantic-based parser to extract both explicit and implicit semantics of logs. Yu et al. introduced Log3T [50] to support new log types inside new-coming logs based on a transformer encoder-based model. Logs have been utilized for anomaly detection for long. Du et al. proposed DeepLog [5], a deep neural network model exploiting Long Short-Term Memory (LSTM) to detect anomalies based on the recognized log patterns. Zhang et al. introduced LogRobust [57], utilizing an attention-based Bi-LSTM model for unstable log events and sequences to detect anomalies. Yang et al. presented nLSALog [48], an anomaly detection framework taking log sequences as sources to detect anomalies in Intelligent Transportation Systems based on the LSTM model and the self-attention mechanism. Previous research highlights the reliance on anomaly detection in log analysis on deep learning models, which require high-quality pre-collected and labeled datasets. The effectiveness of these methods hinges on dataset quality, a time-consuming process. Thus, a user-friendly and time-efficient anomaly identification stage, integrating rule-based strategies to assist end-users in detecting configuration errors, offers a novel approach compared to traditional methods.

LLMs have demonstrated significant capabilities in various domains, including fuzzing, type inference, and more [2, 19, 27, 33, 38, 43]. The integration of LLMs with log analysis is also a growing area of research. Li et al. [26] explored the performance of LLMs on the automated logging statement generation practice. Xu et al. [42] proposed UniLog to automatically decide where and what to log based on the in-context learning paradigm utilized in LLMs. Le et al. [22] investigated the power of ChatGPT in automated log parsing, and Jiang et al. [17] introduced LLMParser, an LLM-based log parsing framework to achieve better performance on log parsing. However, many existing works primarily concentrate on automated logging practice and log parsing, neglecting the capability of LLMs to enhance log analysis for anomaly identification and inference.

In summary, the LLM-based two-stage strategy for localizing configuration errors through logs is user-friendly compared to existing works using log analysis for anomaly detection. Additionally, we make a significant advancement by employing the robust natural language understanding capabilities of LLMs to interpret logs, thereby facilitating the localization of configuration errors.

8 CONCLUSION

Configuration error remains a challenging problem for both experienced maintainers and novice end-users, especially in the scenario of inaccessible source code. Given that logs are an easily accessible resource for most end-users, we conduct a preliminary study and outline the challenges and opportunities to apply log analysis to localize configuration errors. We present an LLM-based two-stage strategy via log analysis based on the insights from the preliminary study. To our knowledge, this is the first work to localize the root-cause configuration properties for end-users based on LLMs and logs. We implement LogConfigLocalizer based on the design of the strategy and show its effectiveness, accuracy, and feasibility via evaluations and a practical case study. We believe our work can alleviate the burden of end-users and facilitate a more streamlined use of configurable software systems.

ACKNOWLEDGEMENTS

We appreciate all the anonymous reviewers for their valuable and practical comments. We also extend our gratitude to Zhuhan Dai, Weifeng Hu, Chengpeng Hu, Jiahao Cui, and Biao Zhu for their invaluable early feedback on the draft of this work. The work described in this paper was supported by the National Natural Science Foundation of China (No. 62202511) and the Guangdong Basic and Applied Basic Research Foundation (2022A1515011713).

REFERENCES

[1] Mona Attariyan and Jason Flinn. 2010. Automating configuration troubleshooting with dynamic information flow analysis. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*.

[2] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, et al. 2023. Empowering Practical Root Cause Analysis by Large Language Models for Cloud Incidents. *arXiv preprint arXiv:2305.15778* (2023).

[3] Marcello Cinque, Domenico Cotroneo, and Antonio Pecchia. 2012. Event logs for the analysis of software failures: A rule-based approach. *IEEE Transactions on Software Engineering* 39, 6 (2012), 806–821.

[4] Zhen Dong, Artur Andrzejak, David Lo, and Diego Costa. 2016. Orplocator: Identifying read points of configuration options via static analysis. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 185–195.

[5] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 1285–1298.

[6] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis. In *2009 Ninth IEEE International Conference on Data Mining*. 149–158. https://doi.org/10.1109/ICDM.2009.60

[7] GPT-4. 2023. gpt-4. https://openai.com/gpt-4. Accessed: 2023-11-23.

[8] Apache Hadoop. 2023. Apache Hadoop. https://hadoop.apache.org/. Accessed: 2023-11-23.

[9] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. 2017. Drain: An Online Log Parsing Approach with Fixed Depth Tree. In *2017 IEEE International Conference on Web Services (ICWS)*. 33–40. https://doi.org/10.1109/ICWS.2017.13

[10] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R Lyu. 2021. A survey on automated log analysis for reliability engineering. *ACM computing surveys (CSUR)* 54, 6 (2021), 1–37.

[11] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. 2018. Identifying impactful service system problems via log analysis. In *Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 60–70.

[12] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. 2010. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*. 41–51. https://doi.org/10.1109/ICDEW.2010.5452747

[13] Yintong Huo, Cheryl Lee, Yuxin Su, Shiwen Shan, Jinyang Liu, and Michael R Lyu. 2023. EvLog: Identifying Anomalous Logs over Software Evolution. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 391–402.

[14] Yintong Huo, Yichen Li, Yuxin Su, Pinjia He, Zifan Xie, and Michael R Lyu. 2023. Autolog: A log sequence synthesis framework for anomaly detection. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 497–509.

[15] Yintong Huo, Yuxin Su, Cheryl Lee, and Michael R Lyu. 2023. Semparser: A semantic parser for log analytics. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 881–893.

[16] Yintong Huo, Yuxin Su, and Michael Lyu. 2022. LogVm: Variable Semantics Miner for Log Messages. In *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 124–125.

[17] Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, and Michael R Lyu. 2023. LLMParser: A LLM-based Log Parsing Framework. *arXiv preprint arXiv:2310.01796* (2023).

[18] Jira. 2023. Jira. https://www.atlassian.com/software/jira. Accessed: 2023-11-23.

[19] Sungmin Kang, Gabin An, and Shin Yoo. 2023. A Preliminary Evaluation of LLM-Based Fault Localization. *arXiv preprint arXiv:2308.05487* (2023).

[20] Lorenzo Keller, Prasang Upadhyaya, and George Candea. 2008. ConfErr: A tool for assessing resilience to human configuration errors. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. 157–166. https://doi.org/10.1109/DSN.2008.4630084

[21] Van-Hoang Le and Hongyu Zhang. 2021. Log-based Anomaly Detection Without Log Parsing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 492–504. https://doi.org/10.1109/ASE51524.2021.9678773

[22] Van-Hoang Le and Hongyu Zhang. 2023. Log Parsing: How Far Can ChatGPT Go?. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 1699–1704.

[23] Junqiang Li, Senyi Li, Keyao Li, Falin Luo, Hongfang Yu, Shanshan Li, and Xiang Li. 2024. ECFuzz: Effective Configuration Fuzzing for Large-Scale Systems. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[24] Wang Li, Zhouyang Jia, Shanshan Li, Yuanliang Zhang, Teng Wang, Erci Xu, Ji Wang, and Xiangke Liao. 2021. Challenges and Opportunities: An in-Depth Empirical Study on Configuration Error Injection Testing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 478–490. https://doi.org/10.1145/3460319.3464799

[25] Xiaoyun Li, Pengfei Chen, Linxiao Jing, Zilong He, and Guangba Yu. 2020. SwissLog: Robust and Unified Deep Learning Based Log Anomaly Detection for Diverse Faults. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. 92–103. https://doi.org/10.1109/ISSRE5003.2020.00018

[26] Yichen Li, Yintong Huo, Zhihan Jiang, Renyi Zhong, Pinjia He, Yuxin Su, and Michael R Lyu. 2023. Exploring the effectiveness of llms in automated logging generation: An empirical study. *arXiv preprint arXiv:2307.05950* (2023).

[27] Yichen Li, Yintong Huo, Renyi Zhong, Zhihan Jiang, Jinyang Liu, Junjie Huang, Jiazhen Gu, Pinjia He, and Michael R Lyu. 2024. Go Static: Contextualized Logging Statement Generation. *arXiv preprint arXiv:2402.12958* (2024).

[28] Yuhao Liu, Wei Wang, Yan Jia, Sihan Xu, and Zheli Liu. 2023. CRSExtractor: Automated configuration option read sites extraction towards IoT cloud infrastructure. *Heliyon* 9, 4 (2023).

[29] Stack Overflow. 2023. Stack Overflow. https://stackoverflow.com/. Accessed: 2023-11-23.

[30] OWASP. 2021. *A05:2021 – Security Misconfiguration*. https://owasp.org/Top10/A05_2021-Security_Misconfiguration/ Accessed: November 23, 2023.

[31] OWASP. 2022. *OWASP Top 10 Vulnerabilities in 2022*. https://www.spiceworks.com/it-security/vulnerability-management/articles/owasp-top-ten-vulnerabilities/#_001 Accessed: November 23, 2023.

[32] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: Coverage-Guided Property-Based Testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 398–401. https://doi.org/10.1145/3293882.3339002

[33] Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R Lyu. 2023. Generative Type Inference for Python. *arXiv preprint arXiv:2307.09163* (2023).

[34] The Open Web Application Security Project. 2023. The Open Web Application Security Project. https://owasp.org/. Accessed: 2023-11-23.

[35] Ariel Rabkin and Randy Katz. 2011. Static extraction of program configuration options. In *Proceedings of the 33rd International Conference on Software Engineering*. 131–140.

[36] CircleID Reporter. 2009. *Misconfiguration Brings Down Entire .SE Domain in Sweden*. https://circleid.com/posts/misconfiguration_brings_down_entire_se_domain_in_sweden/ Accessed: November 23, 2023.

[37] Teng Wang, Xiaodong Liu, Shanshan Li, Xiangke Liao, Wang Li, and Qing Liao. 2018. MisconfDoctor: diagnosing misconfiguration via log-based configuration testing. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 1–12.

[38] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2023. Universal fuzzing via large language models. *arXiv preprint arXiv:2308.04748* (2023).

[39] Xin Xia, David Lo, Weiwei Qiu, Xingen Wang, and Bo Zhou. 2014. Automated Configuration Bug Report Prediction Using Text Mining. In *2014 IEEE 38th Annual Computer Software and Applications Conference*. 107–116. https://doi.org/10.1109/COMPSAC.2014.17

[40] Bowen Xu, David Lo, Xin Xia, Ashish Sureka, and Shanping Li. 2015. EFSPredictor: Predicting Configuration Bugs with Ensemble Feature Selection. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*. 206–213. https://doi.org/10.1109/APSEC.2015.38

[41] Guangquan Xu, Xinru Ding, Sihan Xu, Yan Jia, Shaoying Liu, Shicheng Feng, and Xi Zheng. 2023. Real-Time Diagnosis of Configuration Errors for Software of AI Server Infrastructure. *IEEE Transactions on Dependable and Secure Computing* (2023).

[42] Junjielong Xu, Ziang Cui, Yuan Zhao, Xu Zhang, Shilin He, Pinjia He, Liqun Li, Yu Kang, Qingwei Lin, Yingnong Dang, et al. 2023. UniLog: Automatic Logging via LLM and In-Context Learning. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 129–140.

[43] Junjielong Xu, Ruichun Yang, Yintong Huo, Chengyu Zhang, and Pinjia He. 2024. DivLog: Log Parsing with Prompt Enhanced In-Context Learning. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 983–983.

[44] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 307–319.

[45] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early detection of configuration errors to reduce failure damage. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 619–634.

[46] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 244–259.

[47] Tianyin Xu and Yuanyuan Zhou. 2015. Systems approaches to tackling configuration errors: A survey. *ACM Computing Surveys (CSUR)* 47, 4 (2015), 1–41.
[48] Ruipeng Yang, Dan Qu, Ying Gao, Yekui Qian, and Yongwang Tang. 2019. nLSALog: An Anomaly Detection Framework for Log Sequence in Security Management. *IEEE Access* 7 (2019), 181152–181164. https://doi.org/10.1109/ACCESS.2019.2953981
[49] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. 2011. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 159–172.
[50] Siyu Yu, Yifan Wu, Zhijing Li, Pinjia He, Ningjiang Chen, and Changjian Liu. 2023. Log Parsing with Generalization Ability under New Log Types. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 425–437.
[51] Chun Yuan, Ni Lao, Ji-Rong Wen, Jiwei Li, Zheng Zhang, Yi-Min Wang, and Wei-Ying Ma. 2006. Automated known problem diagnosis with event traces. *ACM SIGOPS Operating Systems Review* 40, 4 (2006), 375–388.
[52] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. Sherlog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*. 143–154.
[53] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2012. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems (TOCS)* 30, 1 (2012), 1–28.
[54] Sai Zhang and Michael D Ernst. 2013. Automated diagnosis of software configuration errors. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 312–321.
[55] Sai Zhang and Michael D Ernst. 2014. Which configuration option should I change?. In *Proceedings of the 36th international conference on software engineering*. 152–163.
[56] Sai Zhang and Michael D Ernst. 2015. Proactive detection of inadequate diagnostic messages for software configuration errors. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 12–23.
[57] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. 2019. Robust log-based anomaly detection on unstable log data. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 807–817.
[58] Zenong Zhang, George Klees, Eric Wang, Michael Hicks, and Shiyi Wei. 2023. Fuzzing configurations of program options. *ACM Transactions on Software Engineering and Methodology* 32, 2 (2023), 1–21.
[59] Shulin Zhou, Xiaodong Liu, Shanshan Li, Zhouyang Jia, Yuanliang Zhang, Teng Wang, Wang Li, and Xiangke Liao. 2021. Confinlog: Leveraging software logs to infer configuration constraints. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 94–105.
[60] Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, and Michael R Lyu. 2023. Loghub: A large collection of system log datasets for ai-driven log analytics. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 355–366.

Received 16-DEC-2023; accepted 2024-03-02