

# ConfLogger: Enhance Systems' Configuration Diagnosability through Configuration Logging

Shiwen Shan

School of Software Engineering,  
Zhuhai Key Laboratory of Trusted  
Large Language Models, Sun Yat-sen  
University  
Zhuhai City, China  
shanshw@mail2.sysu.edu.cn

Yintong Huo

Singapore Management University  
Singapore  
ythuo@smu.edu.sg

Yuxin Su

School of Software Engineering,  
Zhuhai Key Laboratory of Trusted  
Large Language Models, Sun Yat-sen  
University  
Zhuhai City, China  
suyx35@mail.sysu.edu.cn

Zhining Wang

School of Software Engineering,  
Zhuhai Key Laboratory of Trusted  
Large Language Models, Sun Yat-sen  
University  
Zhuhai City, China  
wangzhn23@mail2.sysu.edu.cn

Dan Li\*

School of Software Engineering,  
Zhuhai Key Laboratory of Trusted  
Large Language Models, Sun Yat-sen  
University  
Zhuhai City, China  
lidan263@mail.sysu.edu.cn

Zibin Zheng

School of Software Engineering,  
Zhuhai Key Laboratory of Trusted  
Large Language Models, Sun Yat-sen  
University  
Zhuhai City, China  
zhzibin@mail.sysu.edu.cn

## Abstract

Modern configurable systems, including distributed systems and recently popular decentralized systems in Web 3.0, offer customization via intricate configuration spaces, yet such flexibility introduces pervasive configuration-related issues such as misconfigurations and latent software bugs. Existing diagnosability supports focus on **post-failure analysis of software behavior to identify configuration issues**, but none of these **approaches look into whether the software clues sufficient failure information for diagnosis**. To fill in the blank, we propose the idea of configuration logging to enhance existing logging practices at the source code level. We develop **ConfLogger**, the **first tool that unifies configuration-aware static taint analysis with LLM-based log generation** to enhance software configuration diagnosability. Specifically, our method 1) **identifies configuration-sensitive code segments** by tracing configuration-related **data flow** in the whole project, and 2) **generates diagnostic logging statements** by analyzing **configuration code contexts**.

Evaluation results on eight popular software systems demonstrate the effectiveness of ConfLogger to enhance configuration diagnosability. Specifically, ConfLogger-enhanced logs successfully aid a log-based misconfiguration diagnosis tool to achieve 100% accuracy on error localization in 30 silent misconfiguration scenarios, with 80% directly resolvable through explicit configuration information exposed. In addition, ConfLogger achieves 74% coverage of existing logging points, outperforming baseline LLM-based loggers by 12% and 30%. It also gains 8.6% higher in precision, 79.3% higher

in recall, and 26.2% higher in F1 compared to the state-of-the-art baseline in terms of variable logging while also augmenting diagnostic value. A controlled user study on 22 cases further validated its utility, speeding up diagnostic time by 1.25× and improving troubleshooting accuracy by 251.4%.

## CCS Concepts

• Software and its engineering → Automatic programming.

## Keywords

configuration diagnosability, program analysis, code generation, large language model

## ACM Reference Format:

Shiwen Shan, Yintong Huo, Yuxin Su, Zhining Wang, Dan Li, and Zibin Zheng. 2026. ConfLogger: Enhance Systems' Configuration Diagnosability through Configuration Logging. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26)*, April 12–18, 2026, Rio de Janeiro, Brazil. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3744916.3764570>

## 1 Introduction

Modern software systems, including distributed systems and decentralized systems (e.g., decentralized applications built upon Web 3.0 infrastructure [42]), offer configuration options to accommodate the diverse needs of users [38, 48, 58], where the size of configuration spaces has been growing as systems evolve [5, 58]. The intricate combinations of configurations, along with insufficient documentation, hinder users from setting up the system and lead to configuration-related issues during implementation, which are known as misconfiguration or configuration bugs [20, 44, 48, 51, 53]. Configuration issues can have a wide range of consequences, such as subtle disruptions to the system's functionality [26, 38, 48]. In more severe cases, these errors can lead to catastrophic system failures [51] and financial losses for big companies [41]. To diagnose misconfiguration, existing studies [38, 45, 62] show superiority in

\*corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ICSE '26, Rio de Janeiro, Brazil

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2025-3/26/04

<https://doi.org/10.1145/3744916.3764570>

utilizing software operation logs over source code due to the availability of run-time information [4, 51]. Taking the following run-time log as an example, its configuration parameter identifier (i.e., name) `mapred.local.dir` directly indicates the misconfiguration trigger.

```
No valid local directories in property: mapred.local.dir
```

Nevertheless, these log-based solutions rely on the system's *configuration diagnosability* [55, 59], i.e., whether the subject system could properly log configuration-related events. Furthermore, existing configuration testing tools [23, 26] are also dependent on the output logs for validity verification. When different tools are developed to inspect logs, they often overlook a crucial question: **Do existing systems meet the diagnostic requirements for configuration management?** Configuration diagnosability serves as a basis for upcoming reliable configuration management tools.

Based on existing studies [40, 48, 56], we identify two configuration deficiencies in contemporary systems, categorized by *silent failures* and *insufficient diagnostic messages*. Table 1 presents two real-world user reports of such configuration limitations and their impacts. Silent failures indicate scenarios when parameter value conflict occurred with undocumented fallback (i.e., no clear anomalous output) [56]. In the first case<sup>1</sup>, Hadoop fails to produce warning logs on replication checking failure, which might obviate users to validate system behavior and cause another failure later. Insufficient diagnostic messages denote scenarios when system outputs do not explicitly locate misconfigured parameters by name or value [38, 51, 59], which hinders users from rectifying misconfiguration. For example, the second report<sup>2</sup> displays the insufficient system feedback on problematic configuration parameters, preventing users from diagnosing namenode errors.

To promote the diagnosability of software systems, **we propose a configuration logging strategy** that **improves configuration-related logging practices within the existing codebase**. The insight is that developers should proactively write down logging statements outlining potential misconfiguration causes based on the surrounding code context. In this way, users can leverage run-time logs to troubleshoot configuration issues without having to inspect the source code [38].

Enhancing configuration diagnosability through configuration logging is challenging in: (1) detecting configuration-sensitive code segments that could have impact on software behavior, and (2) generating readable and meaningful logs for diagnostics. The first challenge stems from the diverse utilization of configuration parameters in the whole software project, spanning across declaration, propagation, and usage phases [4, 43]. Since the declaration and usage phases are decoupled, we need to track parameter propagation to identify scattered usage phases, thereby locating configuration-sensitive code segments that encapsulate critical constraints and validation logic for logging. The second challenge involves optimizing logging contents in describing misconfiguration symptoms and underlying reasons. Diagnostic logs should include essential configuration information [38, 59], including identifiers, runtime values, and their causal relationships with system behavior. This requires mechanisms to dynamically map configuration states to

operational impacts and encode these relationships through structured log formats.

To boost system configuration diagnosability via logging support, we present **ConfLogger**, including a configuration-sensitive code identification component and a logging statement generation component. The two components resolve the abovementioned two challenges, respectively. Specifically, in the first component, **ConfLogger** starts with labeling the configuration classes in the codebase by mapping from configuration documents. Then, it tracks data flow and control flow information of the configuration parameters to localize the logical entry points of configuration usage code by taint analysis. These entry points, usually being configuration parameters checking statements (e.g., branches), provide context on how particular configuration parameters are used and impact system behavior in source code. Finally, **ConfLogger** extracts code segments associated with these tracked entry points. In the second component, we leverage Chain of Thought (CoT) [46]-enhanced LLM to automatically generate logging statement contents for the configuration-sensitive code segments. Firstly, LLM analyzes existing code to determine the necessity to inject new logging statements. Then, it determines the optimal log points based on block features and handcrafted logging instructions, and generates complete logging statement accordingly. In a nutshell, our approach gains the advantages of project-level configuration tracing via code analysis, and context-aware log generation via language models, enabling precise diagnosis without manual instrumentation.

To evaluate the effectiveness of **ConfLogger** in improving system diagnosability, we investigate whether the injected logs enhance the capability of the state-of-the-art log-based misconfiguration diagnosis tool across eight representative systems. Experimental results show a 100% diagnosis accuracy when using **ConfLogger**-enhanced logs upon 30 silent misconfiguration cases, compared to 0% original accuracy due to silent failures. 80% of cases are directly localized via configuration parameter names/values exposed in logs (e.g., "Please set 'mapreduce.framework.name' as 'yarn'"). Moreover, compared with other LLM-based logging baselines, **ConfLogger** outperforms state-of-the-art solutions by +8% (74% vs. 66%) and +17% (74% vs. 57%) in the coverage of existing log points and +8.6%/+51.5% higher in precision, +79.3%/+138.3% higher in recall and +26.2%/+42.3% higher in F1 score in variable logging with enriched semantics of logging statements. Regarding efficiency, **ConfLogger** achieves automated identification of configuration entry points, outperforming manual approaches that require 154.696 seconds of human effort to analyze source code statements and extract validation rules, with a 39.36× speedup and a 66.5% reduced invalid rate in our experiments. Last but not least, user studies of **ConfLogger** demonstrate its practicality by speeding up 1.25× diagnosis time and increasing diagnosis accuracy by 251.4% for configuration failures.

In summary, our main contributions are listed as follows:

- ◆ To the best of our knowledge, we are the first to introduce configuration logging as a proactive strategy for enhancing configuration diagnosability.

<sup>1</sup><https://issues.apache.org/jira/browse/HADOOP-14604>

<sup>2</sup><https://issues.apache.org/jira/browse/HDFS-2820>

**Table 1: Motivating examples of deficiencies in configuration diagnosability.**

Report ID	Impact	Error-Triggered Conf Param	# Conf Params	Explanation
HCommon-14604	Silent failure	dfs.replication	398	Silent failure causes the system to ignore dfs.replication in hdfs-site.xml, forcing users to manually check replication counts due to no warnings/errors.
HDFS-2820	Insufficient diagnostic messages	dfs.namenode.shared.edits.dir dfs.namenode.rpc-address	93	Insufficient diagnostic messages omit the misconfigured namenode address when configuring the shared edits directory, causing user confusion and prolonged debugging.

◆ We design and implement ConfLogger<sup>3</sup>, to enhance existing logging practice by two components, i.e., configuration-sensitive code identification and configuration logging statement generation.

◆ We demonstrate the logging quality of ConfLogger on eight software systems, and further show its practicality through user study.

## 2 Problem Definition

In this paper, we formulate the *configuration logging* problem as follows. Given a target system's code  $c_t$ , a logging tool should locate its configuration-sensitive code  $c_c$ , and then outputs its enhanced version  $c_e$  with  $q$  newly injected configuration-related logging statements  $s_{log}$ . The relation between  $c_c$  and  $c_e$  is:  $c_e = c_c \cup \{s_{log1}, \dots, s_{logq}\}$ .

In particular, we decouple the task into two sub-tasks. The first sub-task, is to locate the configuration-sensitive code segments  $c_c$ . With target system's code  $c_t$  containing  $n$  statements  $s$ ,  $c_t = \{s_1, \dots, s_n\}$ , we locate configuration-related code segments  $c_c$  with  $m$  configuration-related statements  $s_c$ ,  $c_c \in c_t$ ,  $\{s_{c1}, \dots, s_{cm}\} \in c_c$ . Taking  $c_c$  as input, the second sub-task targets generating configuration-informative logging statements  $s_{log}$  and inserting them into  $c_c$ , and finally outputs the enhanced code  $c_e$ . To guarantee  $s_{log}$  is configuration-informative,  $s_{log}$  should contain following information of configuration parameters: (1) Configuration parameter identifiers (i.e., names) or configuration parameter values, referring as configuration variables  $var_c$ , (2) Configuration constraints (e.g., configuration dependencies, numeric ranges, etc.) [4, 51]  $text_{cc}$ , and (3) Configuration setting guide  $text_{cg}$  for potential misconfiguration resolution. Therefore, we get:  $\{var_{c1}, \dots, var_{cx}\} \cup \{text_{cc1}, \dots, text_{ccy}\} \cup \{text_{cg1}, \dots, text_{cgz}\} \in s_{log}$ .

## 3 Related Work

**Configuration Practices.** Misconfiguration occur when configuration setting violate the constraints (e.g., invalid value ranges, unsatisfied parameter dependencies, etc.) [32, 53, 56, 58]. To extract configuration constraints, SPEX [51] applies static analysis with predefined rules to track configuration parameter data flow. CDep [4] empirically identifies configuration dependencies through static analysis, while ConfInLog [62] infers constraints directly from log messages. For misconfiguration diagnosis, ConfDiagnoser [57] combines static/dynamic analysis with statistical components. ConfigX [56] localizes silent misconfigurations via static program analysis and deep relation inference. Ciri [31] leverages few-shot LLM technology to locate misconfigurations using

historical data, while LogConfigLocalizer [38] employs rule-based log analysis augmented with LLMs. MisConfDoctor [45] proactively injects misconfigurations to derive diagnostic signatures. Prior work assesses misconfiguration diagnosability [48, 51, 53] through configuration error injection testing (CEIT) [20, 23–26], which injects errors into systems and analyzes their responses. CEIT tools include CeitInspector (a systematic evaluation framework) [25], ConfErr [20], ConfVD [24] and ConfTest [26] which define injection rules and verify validity via system outputs (e.g., logs). ConfDiagDetector [59] enhances diagnosability assessment by combining injection with NLP-based log analysis. In contrast, PCHECK [49] proactively improves system reliability by injecting preemptive parameter checks.

Existing tools address misconfigurations effectively but rely on high system diagnosability, while CEIT tools lack systematic strategies to enhance diagnosability, failing to meet user and tooling requirements [48, 51, 59]. In contrast, ConfLogger enhances configuration logging by explicitly exposing parameter identifiers and other critical information, thereby fundamentally improving diagnosability.

**Logging Practices.** Logging research spans upstream works (logging statement generation [18, 27, 47, 54, 55], logging bug detection [2, 3, 61]), midstream works (log parsing [13, 16, 19]), and downstream works (anomaly detection [6, 15], root cause analysis [38, 45]). Logging statement practice can be further divided into two categories based on their objectives: 1) enhancing the quality of existing logging code and 2) providing automatic suggestions on logging-free code. The first category focuses on improving existing logging's capability in error-handling scenarios, thereby promoting systems' reliability [18, 54, 55]. The second class automatically generates logging statements learning from developers' logging history in general-purpose scenarios [14, 27, 28, 33, 63]. These studies contain decisions-making in logging points [29, 52, 60] (e.g., log points at the line level when given target methods), logging levels [22, 30, 33] (e.g., verbosity level prediction), and constructing complete logging statements [27, 34, 47]. While these studies provide automated logging suggestions, the learning-based nature blocks them from overcoming current drawbacks on logging quality [21, 30, 63].

ConfLogger belongs to the first category but uniquely locates configuration-sensitive code and enhances log instrumentation with configuration-specific diagnostic data. Its diagnosability-centric approach and logging quality improvement further distinguish it from second-category works.

<sup>3</sup><https://github.com/shanshw/ConfLogger>

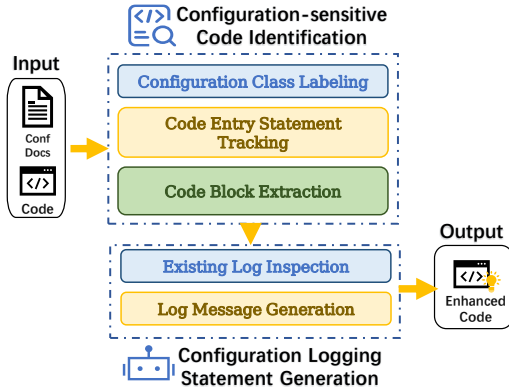


Figure 1: Overview of ConfLogger.

## 4 ConfLogger

### 4.1 Overview

Fig. 1 depicts the design of ConfLogger. Given the system’s code and configuration documentation as input, ConfLogger outputs enhanced code with injected logging statements. Specifically, our model is composed of two components, i.e., configuration-sensitive code identification and configuration logging statement generation. The first identifies configuration-sensitive code blocks as logging target code, and the second injects informative logging statements on the identified code. In the first component, ConfLogger starts with labeling all configuration engine classes in the source code based on the official documentation. Then, it tracks the statements with configuration parameters along the whole software system based on the labeled classes. Based on these statements, ConfLogger further extracts the associated code contexts, denoted as configuration-sensitive code segments. The second component, empowered by LLMs, takes these configuration-sensitive code segments for logging. In particular, ConfLogger firstly determines the necessity of inserting new logging statements. Then, we ask LLMs to generate complete logging statements based on instructions and a set of configuration-tailored logging rules. The output of ConfLogger is the enhanced code equipped with configuration logging.

### 4.2 Configuration-sensitive Code Identification

Considering the large size of the project-level codebase, logging every variable at runtime is impractical and could significantly impact software performance. Therefore, this component begins with identifying scattered code blocks that contain configuration usages.

**4.2.1 Code Unit for Configuration Logging.** Configuration checking and handling are widely used in existing configuration practices [4, 49, 56]. In particular, the checking part validates configuration constraints (e.g., ensuring parameter values fall within predefined ranges, verifying relationships between configuration parameters, and detecting any missing required parameters [4, 32, 49]), and the handling part processes these validation results and mitigates them accordingly. Following these practices, we choose code segments

that involve the configuration checking-handling mechanism as **configuration-sensitive code unit**. Logging over these configuration-sensitive units further offers two advantages: 1) capture system behavior: varying configuration settings can lead to different system behaviors within these code blocks, and 2) provide configuration contexts: the checking-handling process provides essential context for capturing configuration intentions.

An example of configuration-sensitive code is illustrated in the lower corners of Fig. 2, where the code block marked with a green background indicates the checking logic and the handling logic. In this case, the checking part checks if the configuration variable `avoidStaleDataNodeForWrite` is set to “true” and ensures the other two configuration variables (i.e., `staleInterval` and `recheckInterval`) satisfy their dependency relationships. The handling part then assigns this `heartbeatRecheckInterval` the value of either `staleInterval` or `recheckInterval` based on the checking results.

This example illustrates that such code contains rich contextual configuration information, including the values of configuration variables and their interdependencies. Such code context can hint at configuration errors, thereby providing feasible insights for troubleshooting. In contrast, the left panel shows another example that will not be seen as configuration-sensitive code. Here, the green-highlighted branch implements the checking logic, but the condition variable `d.isAlive()` is configuration-independent, thus failing to clue configuration information.

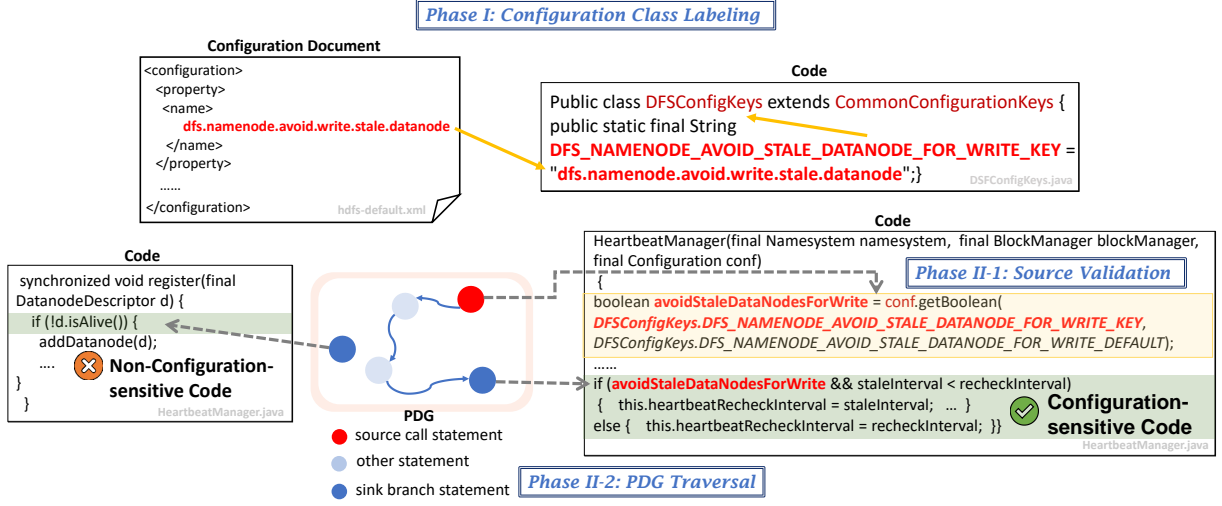
无效代码

**4.2.2 Configuration Class Labeling.** Identifying configuration-sensitive code is challenging for two reasons: first, a software system may have large codebases with multiple files, where configurations might spread across various locations. These files may exceed LLMs’ context length to analyze code and locate certain blocks. Second, the presence of multiple intricate components across various classes, along with complex inheritance hierarchies and composition relationships, hinders comprehensive coverage of configuration engine classes. As a result, manual identification on configuration engines fails to meet practical requirements.

To overcome these challenges, we begin by automatically labeling configuration engine classes from configuration documents. The intuition here is to color every object and class with configuration parameters. In particular, ConfLogger extracts parameter keys from documentation to map their identifiers in source code. These mapped identifiers are then colored by the extracted parameters. ConfLogger identifies seed configuration classes containing these colored identifier variables as filed members. Then, it expands these classes through inheritance hierarchies and composition relationships (e.g., inner and anonymous classes) to label the final configuration engine classes. To ensure comprehensiveness, we incorporate both Java built-in structures and user-specified configuration engines.

The upper side of Fig. 2 displays an example, where ConfLogger parses the configuration document and extracts the configuration parameter name, `dfs.namenode.avoid.write.stale.datanode` as a unified identifier. Then, it maps the configuration variable<sup>4</sup> to the unified qualifier indicated by the yellow arrow. Since the configuration variable is defined as filed members in the java class

<sup>4</sup>DFS\_NAMENODE\_AVOID\_STALE\_DATANODE\_FOR\_WRITE\_KEY



**Figure 2: An example of configuration-sensitive code identification.** The red text indicates objects colored by the configuration parameter, thus being configuration-related.

DFSConfigKeys, we label the class as a seed configuration engine. Besides, ConfLogger includes CommonConfigurationKeys as an expanded configuration engine due to the hierarchy relations.

**4.2.3 Code Entry Statement Tracking.** To locate configuration-sensitive code, we trace data and control flows from configuration parameter definitions to their usage. The logical entry statement in each usage instance would be considered the starting point of configuration-sensitive code. To this end, we conduct inter-procedural analysis on the Program Dependence Graph (PDG) to track all method call edges and path-sensitive data flow based on the idea of taint analysis. PDG is a directed graph representation that encodes data/control dependencies (edges) between program statements (nodes) [7].

To track along PDG, we specify all call statements that are associated with the getter methods in the labeled configuration engines, referred to as candidate source statements for tracing. These getter methods conventionally manage hierarchical key-value mappings. Existing techniques [4, 48] rely on manual annotation of configuration engines to derive source statements, resulting in limited coverage and requiring expertise. To address this, we propose an automated approach to identify valid source statements out of the candidates by a type-specific validation mechanism to getter methods. Specifically, we tailor rules for different types of configuration engines (Table 2): (1) We exclude call statements to the getter methods of Key-Holder (storing parameter identifiers only) as it doesn't store the parameter values that can taint variables; (2) We enable untyped parameter (no constraints) on Both-Holder since it combines key-based and dictionary-based mechanisms, allowing parameter access via direct keys and built-in getter methods for specific parameters (e.g., getResilient()). Doing so offers balanced flexibility. (3) We propose constrained parameter types (Key-Holder or Both-Holder only) on the getters of Dict-Holder (managing configuration dictionaries), reducing false positives through type-restricted value access. This strategy ensures valid source statement

**Table 2: Parameter type requirements on different configuration engine types.**

Parameter Type	Configuration Engine Type
Unified Identifiers	/
/	Both-Holder
Key-Holder & Both-Holder	Built-in Dict-Holder

identification through qualified getter calls while preserving semantic alignment with configuration management patterns.

Sink statements are the destinations of the tracking phase, namely the usage points of configuration parameters. Therefore, we leverage configuration taint analysis to achieve tracking from the identified source statements to these sink statements. Configuration taint analysis is a program analysis approach that tracks the propagation of configuration parameters through execution paths [10, 43], thereby helping tracking configuration usages (i.e., tainted sink statements in technical concepts) in a complicated codebase along the propagation of configuration-related variables. In particular, ConfLogger establishes a PDG for the codebase. The PDG is constructed based on Static Single Assignment Intermediate Representation (SSA IR), enabling precise tracking of both data and control dependencies across procedures. Then it employs the Breadth-First Search Algorithm with limited path lengths to traverse the PDG, tracking tainted sink statements. The path length constraint ensures LLMs' stable capability.

Phase II in Fig. 2 presents the detailed workflow. In Phase II-1, ConfLogger firstly validates the call statement pointed out by the source node (in red circle) in the established PDG since it's a call statement to the public getter method getBoolean of Both-Holder configuration engine, satisfying the validation rules. And then, ConfLogger labels the variable avoidStaleDataNodesForWrite as taint configuration variable because it carries the return value of the identified source statement. In Phase II-2, ConfLogger tracks

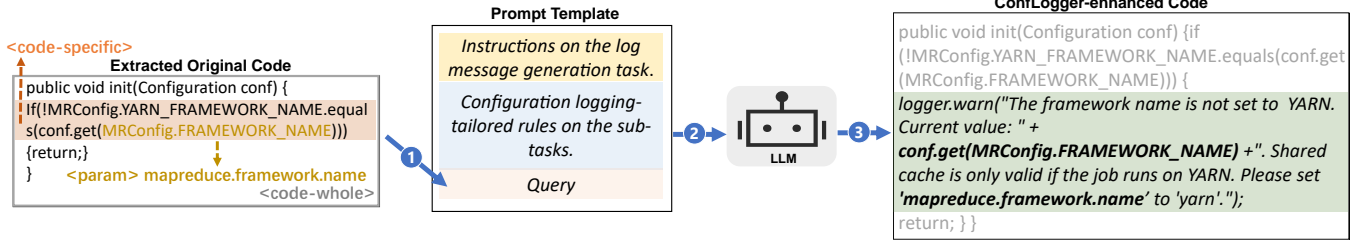


Figure 3: An example case of logging statement generation.

the taint’s data and control flow on the PDG and finds a path reaching a sink branch statement within three hops pointed by the PDG’s sink node (in dark blue). ConfLogger thus detects the tainted sink branch statement, which achieves the checking logic of configuration-sensitive code. These located statements implementing configuration-checking logic will later aid in extracting configuration-sensitive code segments.

**4.2.4 Code Block Extraction.** The located taint sinks are the checking part of configuration-sensitive code. To include their corresponding handling part, we start with grouping the tainted statements by methods. We achieve this by classifying them based on their method signatures. Methods containing these statements are identified as configuration methods and are provided as context to the LLM during subsequent logging statement generation phase. Then, we leverage the WALA framework [9] to obtain the line numbers of these SSA IR statements in the source code, which allows us to determine the starting line numbers of the checking part and extract the handling code accordingly.

### 4.3 Configuration logging statement Generation

As LLMs have demonstrated strong capabilities in code understanding and generation [27], we prompt LLMs with Chain-of-Thought (COT) technologies to generate configuration-related logging statements. Given the configuration-sensitive code segments and the tracked parameters as input, the LLM is required to determine whether to enhance current logging statements or not, followed by identifying optimal log positions and generating the corresponding logging statements. Ultimately, the LLM outputs the enhanced code segments. As illustrated in Fig. 3, in Step ①, ConfLogger marks three key features in the configuration-sensitive code block: <code-specified> (tagging the entry point of the extracted code), <code-whole> (capturing the surrounding code context), and <param> (specifying the configuration parameter `mapreduce.framework.name`). In Step ②, we assemble these code features to construct a prompt for LLM-based logging generation via CoT reasoning. Finally, Step ③ produces enhanced code with a newly-injected logging statement. The generated log message includes the configuration variable<sup>5</sup> and actionable guidance<sup>6</sup> to enable the Shared Cache module in MapReduce.

**4.3.1 Existing Log Inspection.** To mitigate excessive logging, we systematically evaluate existing logging statements on the extracted

Table 3: Target systems. \* The version is 3.0.0-beta-1

Systems	Version	LoC	# Configuration Parameters
Storm	2.6.2	208,677	275
Hbase*	3.0.0	87,438	232
Alluxio	3.1.3	78,541	821
HCommon	3.3.6	180,161	461
Mapreduce	3.3.6	122,822	223
Yarn	3.3.6	323,532	545
HDFS	3.3.6	557,733	643
ZooKeeper	3.9.2	62,681	167

code following established practices [55]. We establish rule-based criteria for LLM decision-making: When the given code contains logging statements with sufficient configuration details (e.g., parameter keys or value ranges), the LLM recommends retaining existing logs; otherwise, it suggests removing redundancies while annotating rationale based on runtime behavior. The LLM skips inspection for code without existing logging statements.

**4.3.2 Log Message Generation.** In this step, LLMs are required to determine the optimal position in code blocks and produce contextual logging statements accordingly. To control logging overhead on selecting essential path constraints, we define rules for different checking scenarios: (1) Insert logs on paths handling invalid/unset values to expose silent default fallbacks as former works points out [51, 56], therefore revealing constraint violation corrections; (2) Instrument service activation/deactivation paths to verify configuration prerequisites as the second motivating example on Table 1 exposes, hence guaranteeing intended behavior, and confirming service-switching outcomes; (3) Insert logs on configuration processing paths to monitor parameter transformations, and validate value-driven system behavior. The observation is that this information uncovers the hidden configuration management in code, therefore delivering troubleshooting guidance for potential misconfigurations. Furthermore, we ask LLMs to generate log contents following SLF4j standards [39] with three mandatory components: (1) severity levels selected from predefined SLF4j options (2) static messages embedding configuration constraints for diagnostics and (3) dynamic variables capturing parameter names and runtime values, aligned with configuration troubleshooting practices [38, 51, 59]. Additionally, the LLM was also instructed to offer configuration guidance to mitigate potential misconfigurations.

<sup>5</sup>`conf.get(MRConfig.FRAMEWORK_NAME)`

<sup>6</sup>Please set ‘mapreduce.framework.name’ to ‘yarn’

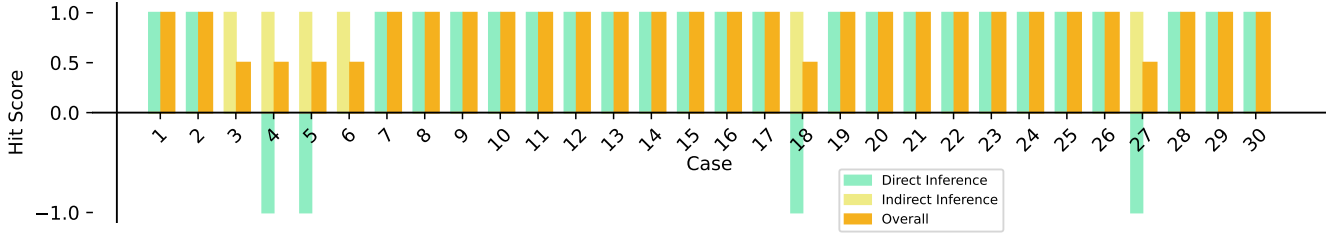


Figure 4: Overall Hit Scores and Hit Scores of different inference phases.

## 5 Evaluation

We conduct experiments for the following research questions (RQs):

**RQ1: How effective of ConfLogger on enhancing systems' configuration diagnosability?**

**RQ2: How does ConfLogger comparing other logging tools on configuration logging?**

**RQ3: How would the source identification strategy affect logging?**

**RQ4: To what extent does ConfLogger help users in mis-configuration diagnosis? (Practical user study)**

### 5.1 Experimental Setup

We implement ConfLogger by a Java-based program analysis module with 4,000 LOC using WALA [9] and ASM [8], and an LLM interface invoking GPT-4o-2024-08-06 via OpenAI API [11] with temperature set to 0 for reproducibility. Following prior studies on configuration practice [38, 40, 48–50], we evaluate our method in eight prominent Java systems: Storm, HBase, Alluxio, Hadoop Common, MapReduce, Yarn, HDFS, and ZooKeeper. Table 3 details these systems, with their code statistics calculated using the cloc tool [1].

We collect JAR/source code files of target systems from the Maven repository [36] and use official configuration documentation as input for ConfLogger. The average execution time is 355.5 minutes. To balance efficiency and accuracy, we constrain tainted path length to 30, mitigating redundant long-path exploration. All experiments ran on an x86\_64 server (80-core Intel Xeon Gold 5218R @2.10GHz, 267GiB RAM, 5.6TB disk) through Docker on Ubuntu 22.04, with Python 3.12.4 and Java 11.

### 5.2 Datasets

Benchmark I includes 30 enhanced code cases whose original implementation suffers from silent failures. These cases' original code fails to log critical configuration-related information, including failures to activate a service due to neglect of certain configuration parameters, out-of-valid-range value settings, thus expose severe potential misconfiguration risks. We identify these case from its original implementation and employ ConfLogger to replicate them. Therefore, they serve as the perfect experiment benchmark to highlight ConfLogger's capability on enhancing configuration diagnosability, with silent failures eliminated.

Benchmark II includes 70 original configuration-sensitive code cases with 90 log points removed. These log points, identified by ConfLogger as diagnostically sufficient and manually validated as

configuration-related, serve as ground truth to evaluate ConfLogger-enhanced logs against two LLM-based baseline loggers in subsequent comparisons.

### 5.3 Evaluation Results

**5.3.1 RQ1: How effective of ConfLogger on enhancing systems' configuration diagnosability?** We validate the effectiveness of ConfLogger on Benchmark I using LogConfigLocalizer [38] by measuring successful case located by the tool.

**Experiment Setting.** We collect run-time logs by executing test suites on original and ConfLogger-enhanced code in Benchmark I. For untested cases, we manually craft test methods. We employ a log-based misconfiguration diagnosis tool to locate the misconfiguration-trigger parameters. LogConfigLocalizer employs rule-based extraction of explicit parameters in logs during its direct inference phase, validated via LLM verification. If the direct inference fails, it activates the indirect inference, using LLMs to deduce implicit configuration issues from ambiguous logs.

**Metrics.** We propose *Hit Score*, awarding 1 (direct success), 0.5 (indirect success after failure) or 0 (total failure) for overall diagnosis, with phase-specific scores of 1/0/-1. Higher scores indicates stronger diagnosability of ConfLogger logs.

**Results.** Fig. 4 presents the results. All of the 30 cases received an Overall Hit Score between 0.5 and 1, confirming successful localization in at least one phase. This yields 100% localization accuracy, contrasted against original implementations' silent failures. Six cases scored 0.5 due to insufficient explicit information (e.g., missing parameter names/values), with four root causes: (1) Misleading parameter-like tokens (4 cases), (2) Logged variable names instead of documented constants (2 cases), (3) Ambiguous natural language descriptions (2 cases), and (4) Parameter name/documentation inconsistencies (2 cases). While these limitations stem from documentation deficiencies or LLM hallucinations [17, 35], all achieve Indirect Hit Scores of 1, demonstrating latent diagnosability. The majority (24/30) earn direct success (Hit Score 1), indicating enhanced logs' effectiveness in explicit parameter disclosure. Fig. 3 demonstrates Case 21. The extracted original code silently exits without providing feedback when users attempt to enable the Shared Cache module without setting `mapreduce.framework.name` as `yarn`. In contrast, the enhanced version incorporates a logging statement that explicitly links the missing parameter to component initialization failures, enabling immediate diagnosis and guiding users to apply the correct cluster resource management settings.

**Answer to RQ1:** ConfLogger achieves 100% accuracy in diagnosing misconfigurations, eliminating silent failures in original implementations. Among all, 80% cases are resolved via direct parameter extraction, significantly reducing diagnostic overhead.

**5.3.2 RQ2: How does ConfLogger comparing other logging tools on configuration logging?** We compare ConfLogger with UniLog [47] and SCLogger [27] on Benchmark II by measuring how many logging statements added manually can be added automatically by these tools, following previous work [54].

**Experiment Setting.** UniLog employs in-context learning (ICL) to retrieve top-5 similar log method examples via KNN from an external repository, prompting LLMs for target logging, while SCLogger utilizes static program analysis to extract call-chain contexts and applies BM25-based ICL for automated logging. We reproduce a naive version of SCLogger, with the code slice graph construction and ICL strategy, under the agreement of the authors. These tools lack mechanisms on identification of configuration-sensitive code, as they focus on general logging automation rather than configuration-specific requirements. To enable comparative evaluation, we adapted these tools by pre-populating project-specific logging methods and manually designating target methods to meet their input constraints. While this adaptation introduces bias by bypassing ConfLogger’s automated configuration-sensitive code localization, the comparison remains valid to demonstrate its superiority in configuration scenarios over general-purpose loggers, being the first configuration-aware methodology.

**Metrics.** We evaluate logging quality through position accuracy (PA), log-level metrics, variable relevance, and textual similarity. PA quantifies positional validity as 1 only if the predicted log line is within one line of the ground truth and resides in the same code block; otherwise, PA is 0. System coverage is calculated as  $C = \frac{N_{PA=1}}{N_t}$  where  $N_{PA=1}$  counts valid log injections and  $N_t$  represents the total ground truth log points. For cases where PA equals 1, we further evaluate log-level alignment through Level Accuracy (LA), which measures exact matches between predicted and actual log levels, and Average Ordinal Distance (AOD), defined as  $AOD = 1 - \frac{Dist(L_T, L_I)}{maxDist(L_T)}$  where  $L_T$  and  $L_I$  denote the ground truth and injected log levels, respectively. Variable relevance is assessed using Precision  $P = \frac{var_t \cap var_i}{var_i}$ , Recall  $R = \frac{var_t \cap var_i}{var_t}$ ,  $F1 = \frac{2PR}{P+R}$  (omit if no variables) where  $var_t$  represents the ground truth variable and  $var_i$  denotes the injected variable; Textual similarity is measured via BLEU-1, 4 and ROUGE-1, L scores, which compute n-gram overlaps between generated and actual log texts (range: 0–1). Metrics for log levels, variables, and texts are evaluated exclusively for PA=1 cases, as incorrect log positions invalidate the purpose of downstream analysis.

**Results.** Results are summarized in Table 4. ConfLogger (CL) outperforms UniLog (UL) and SCLogger (SL) in diagnostic effectiveness, achieving higher coverage (74% vs. UL:66%, SL:57%), log-level precision (AOD 0.853 vs. UL:0.845, SL:0.747) and optimal logging variable performance (F1 0.501 vs. UL:0.397, SL: 0.352). For coverage, CL’s lower coverage in certain cases, particularly within catch blocks, stems from its intentional insensitivity to generic exception-handling paths that lack explicit validation logic statements (e.g., missing if-else checking statements in such blocks). While UL and

SL suffer from coverage limitations due to their inability to prioritize configuration-sensitive code segments. UL’s ICL strategy struggles to emphasize critical operations in lengthy code snippets, while SL’s inclusion of an extended code slice graph within two hops introduces distracting context, further degrading its performance. Both methods tend to miss log points in configuration validation, followed by specific handling operations (e.g., reverting variables defined/passed outside the branch scope to default values), as they fail to highlight specific logging scopes for LLMs. For log levels, CL tends to prioritize “ideal-state” monitoring (e.g., service availability, parameter compliance), leading to higher severity assignments (e.g., WARN for disabled services). Such a design trade-off reduces its Level Accuracy (LA) compared to UL’s ICL-driven mimicry of historical patterns, however, with higher AOD. SL’s reliance on BM25 sampling for context retrieval underperforms UL’s KNN-based ICL, likely due to suboptimal similarity matching in configuration-specific scenarios. For variable logging, CL’s advantage arises from two mechanisms: its ability to trace of configuration parameters and contextual emphasis on condition variables. UL and SL, lacking such capabilities, fail to retain critical variables in long code contexts. Text similarity metrics reflect CL’s deliberate departure from legacy logs’ poor diagnosability. By injecting configuration constraints and mitigation guidance, CL prioritizes actionable diagnostics over syntactic replication.

Fig. 5 demonstrates a case that exactly explains the reasons for the performance divergence. The original code on the upper side highlights the printed variables marked in *italics*. The lower side shows logging statements injected by the tools. With tracked parameter identifier and specified configuration-sensitive code segments as context, CL highlights the configuration parameter<sup>7</sup> (marked in bold), whose value is carried by the condition variable<sup>8</sup>. Neither UL nor SL is capable of emphasizing such configuration-sensitive context to LLMs, thus they fail to log the corresponding parameter. The original code also fails to indicate the related configuration parameter, demonstrating limited diagnosability. UL achieves higher similarity on the logging variables and the text; however, CL highlights the crucial configuration parameter leading to misconfiguration and includes more instructive text, showing higher diagnostic value.

**Answer to RQ2:** ConfLogger achieves 74% log coverage (vs. 66% and 57%) and 0.853 AOD (vs. 0.845 and 0.747) with superior variable logging (F1=0.541 vs. 0.397 and 0.352), demonstrating optimal configuration-aware diagnosability. Its intentionally low text similarity prioritizes diagnostic capabilities over syntactic replication.

**5.3.3 RQ3: How would the source identification strategy affect logging?** To assess the impact of the source identification strategy, we compare the original and an experimental variant by assessing the efficiency of labeling the configuration engines and the quality of the extracted configuration-sensitive code block.

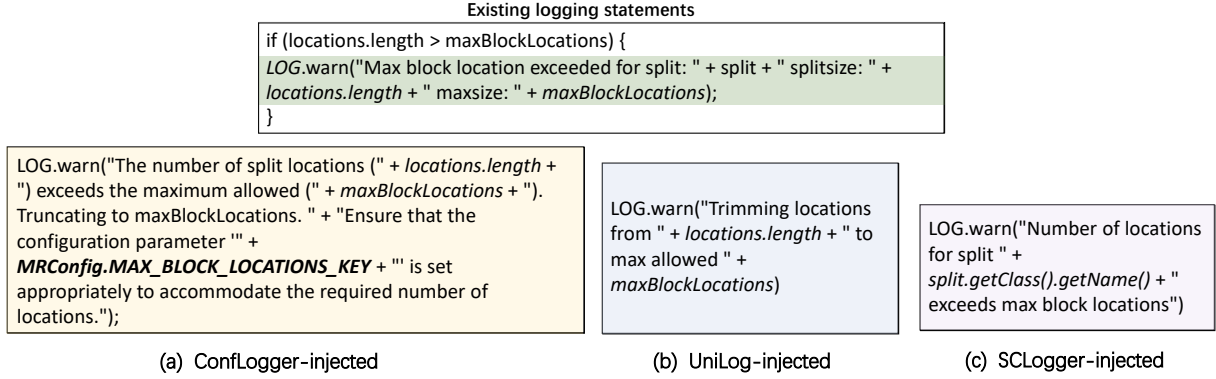
**Experiment Setting.** The RN (Random-NoType) variant removes the automated strategy, instead randomly selecting one seed configuration engine from ConfLogger’s configuration engine list grasped by the automated source identification strategy. Engineers then

<sup>7</sup>MRCConfig.MAX\_BLOCK\_LOCATIONS\_KEY

<sup>8</sup>maxBlockLocations

**Table 4: Results against baseline tools on existing logging practice.\* Despite methodological concerns from existing logs' poor diagnosability, we include text similarity analysis for evaluation completeness.**

Systems	Coverage	Log Level		Logging Variable			Logging Text*			
		LA	AOD	PRECISION	RECALL	F1	BLEU-1	BLEU-4	ROUGE-1	ROUGE-L
CL-Storm	<b>80% (8/10)</b>	<b>0.875</b>	<b>0.938</b>	0.630	<b>0.343</b>	0.533	<b>0.173</b>	<b>0.025</b>	0.344	0.267
UL-Storm	40% (4/10)	0.750	0.875	/	0	/	0.126	<b>0.025</b>	<b>0.358</b>	<b>0.323</b>
SL-Storm	70% (7/10)	0	0.429	<b>0.750</b>	0.171	<b>0.643</b>	0.084	<b>0.025</b>	0.196	0.179
CL-Hbase	<b>100% (2/2)</b>	0	0.667	/	0	/	<b>0.282</b>	<b>0.022</b>	<b>0.342</b>	<b>0.299</b>
UL-Hbase	<b>100% (2/2)</b>	<b>1.000</b>	<b>1.000</b>	/	0	/	0.133	0.020	0.307	0.227
SL-Hbase	<b>100% (2/2)</b>	<b>1.000</b>	<b>1.000</b>	/	0	/	0.154	<b>0.023</b>	0.217	0.130
CL-Alluxio	<b>100% (2/2)</b>	<b>0.500</b>	<b>0.833</b>	<b>1.000</b>	<b>0.583</b>	<b>0.733</b>	<b>0.265</b>	<b>0.032</b>	<b>0.296</b>	<b>0.296</b>
UL-Alluxio	50% (1/2)	0	0.333	<b>1.000</b>	0.333	0.500	0.070	0.016	0.000	0.000
SL-Alluxio	<b>100% (2/2)</b>	<b>0.500</b>	0.667	/	0	/	0.035	0.008	0.105	0.105
CL-HCommon	58% (7/12)	0.429	0.738	<b>0.694</b>	<b>0.833</b>	<b>0.744</b>	0.128	0.018	0.293	0.242
UL-HCommon	<b>75% (9/12)</b>	<b>0.778</b>	<b>0.889</b>	0.417	0.312	0.389	<b>0.210</b>	<b>0.107</b>	<b>0.466</b>	<b>0.426</b>
SL-HCommon	50% (6/12)	0.333	0.694	0.250	0.250	0.292	0.144	0.024	0.288	0.272
CL-Mapreduce	<b>81% (17/21)</b>	<b>0.824</b>	0.882	<b>0.590</b>	<b>0.333</b>	<b>0.437</b>	<b>0.105</b>	0.012	0.200	0.175
UL-Mapreduce	71% (15/21)	0.733	<b>0.889</b>	0.538	0.250	0.397	0.095	0.019	0.277	0.236
SL-Mapreduce	67% (14/21)	0.571	0.762	0.312	0.150	0.367	0.102	<b>0.022</b>	<b>0.316</b>	<b>0.297</b>
CL-Yarn	<b>67% (6/9)</b>	<b>0.500</b>	<b>0.833</b>	<b>0.667</b>	<b>0.611</b>	<b>0.660</b>	0.080	0.015	0.185	0.185
UL-Yarn	56% (5/9)	0.400	0.767	<b>0.667</b>	0.267	0.500	0.057	0.010	<b>0.239</b>	<b>0.226</b>
SL-Yarn	44% (4/9)	0.250	0.750	0.500	0.250	0.500	<b>0.098</b>	<b>0.018</b>	0.179	0.179
CL-HDFS	<b>71% (22/31)</b>	0.591	0.871	0.352	<b>0.433</b>	<b>0.365</b>	0.147	0.020	0.246	0.194
UL-HDFS	<b>71% (22/31)</b>	0.591	0.841	<b>0.400</b>	0.258	0.333	0.121	0.028	0.168	0.146
SL-HDFS	48% (15/31)	<b>0.733</b>	<b>0.883</b>	0.333	0.250	0.292	<b>0.198</b>	<b>0.046</b>	<b>0.276</b>	<b>0.276</b>
CL-ZooKeeper	<b>100% (3/3)</b>	<b>0.667</b>	<b>0.778</b>	0.750	0.500	0.750	0.072	0.011	0.174	0.174
UL-ZooKeeper	33% (1/3)	0	0.333	<b>1.000</b>	<b>1.000</b>	<b>1.000</b>	<b>0.833</b>	<b>0.760</b>	<b>0.909</b>	<b>0.909</b>
SL-ZooKeeper	33% (1/3)	0	0.667	/	0	/	0.077	0.016	0.222	0.222
CL-Average	<b>74% (67/90)</b>	0.642	<b>0.853</b>	<b>0.541</b>	<b>0.459</b>	<b>0.501</b>	<b>0.135</b>	0.018	0.247	0.207
UL-Average	66% (59/90)	<b>0.644</b>	0.845	0.498	0.256	0.397	0.134	<b>0.048</b>	<b>0.274</b>	0.244
SL-Average	57% (51/90)	0.490	0.747	0.357	0.193	0.352	0.131	0.029	0.260	<b>0.247</b>

**Figure 5: A case example on Benchmark II.**

manually trace general configuration engines, identify getter functions for taint analysis rules. Conventional approaches only require locating configuration engine types regardless of specific parameters as RN implements. We deem the original version as Auto, and compare it with the RN version.

**Metrics.** We introduce metrics: (1) configuration class localization time, (2) total labeled configuration engines, (3) extracted configuration-sensitive code, and (4) valid/invalid/version-specific

cases (where version-specific cases are those extracted exclusively by a single version). The specific rate is computed by  $r = \frac{N_s}{N_o}$ , where  $N_s$  is the count of version-specific cases and  $N_o$  is the number of valid cases.

**Results.** Table 5 shows the details. The experimental results demonstrate that ConfLogger (Auto version) significantly outperforms the RN version across multiple metrics. Strategy Time is

**Table 5: Comparison results on source statement identification strategy versus the RN variant.**

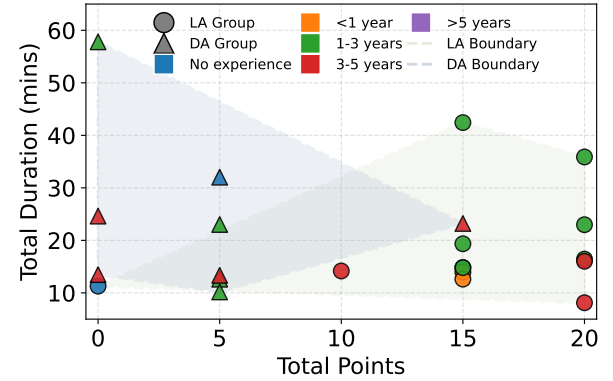
Systems	Version	Strategy Time(s)	# Engines	Overall Configuration Method				Valid Configuration Method	
				# Valid	# Invalid	# Total	% Invalid Rate	# Specific Case	% Specific Rate
Storm	RN	50.338	2	55	68	123	55.30	22	40.000
	Auto	<b>1.465</b>	<b>4</b>	<b>65</b>	<b>0</b>	65	<b>0</b>	<b>33</b>	<b>50.769</b>
Hbase	RN	197.209	2	17	0	17	<b>0</b>	0	0
	Auto	<b>1.810</b>	<b>8</b>	<b>19</b>	<b>0</b>	<b>19</b>	<b>0</b>	<b>2</b>	<b>10.526</b>
Alluxio	RN	298.108	2	3	0	3	<b>0</b>	1	33.333
	Auto	<b>0.414</b>	<b>10</b>	<b>24</b>	<b>0</b>	<b>24</b>	<b>0</b>	<b>23</b>	<b>95.833</b>
HCommon	RN	189.52	1	77	6	83	7.20	19	24.675
	Auto	<b>10.353</b>	<b>89</b>	71	5	76	<b>6.60</b>	13	18.310
Mapreduce	RN	71.109	1	95	8	103	7.80	3	3.158
	Auto	<b>0.616</b>	<b>2</b>	<b>101</b>	<b>8</b>	<b>109</b>	<b>7.30</b>	<b>10</b>	<b>9.901</b>
Yarn	RN	162.198	1	38	1	39	<b>2.60</b>	5	13.158
	Auto	<b>0.846</b>	<b>5</b>	37	2	39	5.10	4	10.811
HDFS	RN	102.409	1	68	3	71	4.20	4	5.882
	Auto	<b>11.754</b>	<b>20</b>	<b>102</b>	<b>1</b>	<b>103</b>	<b>1.00</b>	<b>40</b>	<b>39.216</b>
ZooKeeper	RN	166.677	2	9	45	54	83.30	0	0
	Auto	<b>4.172</b>	<b>81</b>	<b>21</b>	<b>27</b>	<b>48</b>	<b>56.30</b>	<b>16</b>	<b>76.190</b>
Average	RN	154.696	1.5	45.250	16.375	61.625	26.60	6.750	14.917
	Auto	<b>3.929</b>	<b>27.375</b>	<b>55</b>	<b>5.375</b>	<b>60.375</b>	<b>8.90</b>	<b>17.625</b>	<b>32.045</b>

drastically reduced in all systems with Auto, averaging 3.929s compared to RN's 154.696s, reflecting a 97.5% improvement in efficiency. For example, Storm's Auto version completes in 1.465s versus RN's 50.338s, while Alluxio's Auto version achieves 0.414s compared to RN's 298.108s. Auto also exhibits superior configuration validity, with an average 8.90% invalid rate versus RN's 26.60%. Systems like Storm and HBase using Auto achieve 0% invalid configurations, eliminating errors entirely. Additionally, Auto supports more engines (average 27.375 vs. RN's 1.5), enhancing scalability. Notably, Auto excels in specific case tracking, achieving a 32.045% average rate (vs. RN's 14.917%). For instance, Alluxio's Auto version tracks 95.833% of specific cases (vs. 33.333% for RN), and Zookeeper's Auto version achieves 76.190% (vs. 0% for RN). RN's ability to locate more configuration-sensitive code segments because it has no restrictions on parameter types of getter methods, thus enabling more identified source statements. However, such lack of restrictions results in more invalid code segments. In summary, ConfLogger (Auto) delivers faster execution, higher accuracy, and better scalability while maintaining robustness across diverse systems.

**Answer to RQ3:** ConfLogger achieves a 39.36× speedup in configuration engine identification, with an 8.9% invalid detection rate and 32% specific success rate. These quantitative outcomes validate its scalability and precision, substantially reducing manual operational overhead in configuration automation.

**5.3.4 RQ4: To what extent does ConfLogger help users in misconfiguration diagnosis? (Practical user study).** To evaluate ConfLogger's practicality in human-centered misconfiguration diagnosis, we conduct a controlled user study.

**Experiment Setting.** The user study employs a between-subjects experimental design [12, 55] with five representative misconfiguration scenarios sampled from Benchmark I. Participants are randomly allocated to two experimental conditions: the Documentation-Assisted (DA) group using official configuration Documents, and

**Figure 6: Results of Group LA and DA on total points and duration.**

the Log-Assisted (LA) group utilizing ConfLogger's enhanced logs, with equivalent task complexity across both groups. The timed experiment restricted task completion to 60 minutes, with quality control measures excluding responses completed in under 8 minutes from the final dataset.

**Metrics.** To assess diagnostic performance, we implement a scoring system awarding 5 points for correct parameter identification and 0 point for incorrect responses. Participants' development experience is categorized into five proficiency levels (No experience, <1, 1-3, 3-5, >5 years) to contextualize performance analysis.

**Results.** Fig. 6 compares diagnostic performance between LA and DA groups. LA Group data points (circles) cluster in the lower-right quadrant (green dashed boundary), demonstrating a 1.25× speedup in diagnostic time (18.68 vs. 23.36 minutes) and 251.4% accuracy improvement (15.38 vs. 4.38 scores) over DA Group, demonstrating efficiency and accuracy improvements. In contrast, DA

Group data points (triangles) concentrate in the upper-left quadrant (blue dashed boundary), reflecting prolonged efforts for lower scores, likely due to inefficient parameter searches in documentation. Notably, DA Group exhibits sparse high-score outliers requiring disproportionate time. Participant experience analysis reveals LA Group's robustness: even inexperienced users (0–1 year) achieved scores of 15–20 with minimal time, except one novice who scored zero after early abandonment. Conversely, DA Group yielded zero scores even among experienced developers (3–5 years), underscoring enhanced logging's efficacy in reducing diagnostic effort. Taking the demonstrated case in Fig. 3 for example, we simulate a failure scenario where the Shared Cache component in a MapReduce system fails to activate, prompting participants to identify the correct parameter among ten options. LA Group receives enhanced logs, while DA Group relies on ambiguous documentation for `mapreduce.framework.name`, which lacks explicit parameter-component correlation. Results show 85% accuracy in LA Group (including 27% of inexperienced users) versus 11% in DA Group, where the sole correct response resulted from a guess. Enhanced logs enabled diagnostic success even among inexperienced users (0–1 year), while traditional documentation led to inefficiencies (e.g., failures by experienced developers due to ambiguous parameter

**Answer to RQ4:** ConfLogger substantially reduces diagnostic effort with enhanced logs, achieving a 251.4% accuracy improvement and resolving issues 1.25× faster than documentation-based approaches. These enhancements also enable successful diagnosis even for inexperienced users.

## 6 Threats to Validity

This section systematically examines the internal and external validity of ConfLogger, addressing potential threats and corresponding mitigation strategies.

**Internal Threats.** The primary internal validity concern stems from potential false positives in our program analysis. The PDG-based taint analysis, which employs control dependence for inter-procedural analysis, may lead to over-tainting phenomena [4, 37]. To mitigate this, we implemented two safeguards: (1) restricting tainted path lengths to a maximum of 30 edges in our experimental settings, and (2) providing an optional control dependence deactivation mechanism during PDG construction. An additional validity consideration arises from our validation strategy for getter method calls, which lacks parameter type constraints. While this optimization enhances efficiency, it may permit false positive confirmations. A proposed mitigation involves disabling confirmation procedures for such method calls. A third internal threat is a potential for data leakage from the LLM's pre-training data. The LLM was trained on data up to October 2023, introducing a residual risk that some benchmark code, particularly from Benchmark II, may have been seen. While this could potentially inflate absolute performance, the comparative performance of our study remains valid, as all evaluated tools use the same base LLM. For Benchmark I, this risk is negligible as the log-enhanced code versions were generated by ConfLogger.

**External Threats.** Two primary factors may affect generalizability. First, our analysis framework assumes standard configuration practices and is implemented in Java. While our core methodology

is language-agnostic, adapting it to other languages would require modular adjustments. Second, our bytecode-to-source code mapping mechanism necessitates the concurrent availability of both source files and compiled JARs. This dependency could be alleviated by embedding source code within JAR packages during deployment. Another external threat arises from the impact of our approach on other diagnostic problems. While ConfLogger enhances logs for configuration issues, the added information could potentially introduce noise for diagnostic modules designed for other problem types. However, as the enriched logs help quickly exclude common configuration issues, the benefits outweigh this drawback. Furthermore, since ConfLogger operates primarily within configuration-sensitive code blocks, its impact on diagnosing non-configuration issues is inherently constrained. Future work could explore strategies to balance this trade-off by selectively including other relevant variables.

## 7 Conclusion

While modern software systems offer large configuration spaces for customization, they also put a higher requirement on diagnosing configuration errors. This paper introduces the idea of configuration logging to expose run-time configuration details by inserting essential logging statements. Based on this insight, we design ConfLogger to enhance system logging practices specifically for configurations, thereby benefiting in diagnosing relevant errors later. More specifically, ConfLogger consists of two key components: a configuration-sensitive code identification component realized by a taint analysis module to track configuration usage and an LLM-empowered logging statement generation component. Our evaluations on eight mature systems confirm the effectiveness of ConfLogger in identifying and logging critical configuration-related information. Moreover, our user study underscores the practical applicability of ConfLogger in real-world misconfiguration diagnosis scenarios.

## Acknowledgments

We appreciate all the anonymous reviewers for their valuable and practical comments. The work was supported by the National Key Research and Development Program of China (2023YFB2704100), the National Natural Science Foundation of China (No. 62202511), CCF - Sangfor 'Yuanwang' Research Fund and the Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant.

## References

- [1] AlDanial. 2024. cloc. <https://github.com/AlDanial/cloc>. Accessed: 2025-01-20.
- [2] Boyuan Chen and Zhen Ming Jiang. 2017. Characterizing and detecting anti-patterns in the logging code. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 71–81.
- [3] Boyuan Chen and Zhen Ming Jiang. 2019. Extracting and studying the Logging-Code-Issue-Introducing changes in Java-based large-scale open source software systems. *Empirical Software Engineering* 24 (2019), 2285–2322.
- [4] Qingrong Chen, Teng Wang, Owolabi Legunsen, Shanshan Li, and Tianyin Xu. 2020. Understanding and discovering software configuration dependencies in cloud and datacenter systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 362–374.
- [5] 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.
- [6] 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.
- [7] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.
- [8] ASM Framework. 2025. <https://asm.ow2.io/>. Accessed: 2025-01-20.
- [9] WALA Framework. 2025. <https://github.com/wala/WALA>. Accessed: 2025-01-20.
- [10] Ying Fu, Teng Wang, Shanshan Li, Jinyan Ding, Shulin Zhou, Zhouyang Jia, Wang Li, Yu Jiang, and Xiangke Liao. 2024. MissConf: LLM-Enhanced Reproduction of Configuration-Triggered Bugs. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 484–495.
- [11] GPT-4o. [n. d.]. GPT-4o. <https://platform.openai.com/docs/models/gpt-4o>. Accessed: 2025-01-20.
- [12] James Hampton. 2018. The between-subjects experiment. In *Laboratory psychology*. Psychology Press, 15–37.
- [13] 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)*. IEEE, 33–40.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. *Comput. Surveys* 55, 12 (2023), 1–38.
- [18] Zhouyang Jia, Shanshan Li, Xiaodong Liu, Xiangke Liao, and Yunhui Liu. 2018. SMARTLOG: Place error log statement by deep understanding of log intention. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 61–71.
- [19] Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, and Michael R Lyu. 2024. LILAC: Log parsing using LLMs with adaptive parsing cache. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 137–160.
- [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)*. IEEE, 157–166.
- [21] Heng Li, Weiye Shang, Bram Adams, Mohammed Sayagh, and Ahmed E Hassan. 2020. A qualitative study of the benefits and costs of logging from developers' perspectives. *IEEE Transactions on Software Engineering* 47, 12 (2020), 2858–2873.
- [22] Heng Li, Weiye Shang, and Ahmed E Hassan. 2017. Which log level should developers choose for a new logging statement? *Empirical Software Engineering* 22 (2017), 1684–1716.
- [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] Shanshan Li, Wang Li, Xiangke Liao, Shaojiang Peng, Shulin Zhou, Zhouyang Jia, and Teng Wang. 2018. Confvd: System reactions analysis and evaluation through misconfiguration injection. *IEEE Transactions on Reliability* 67, 4 (2018), 1393–1405.
- [25] 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*. 478–490.
- [26] Wang Li, Shanshan Li, Xiangke Liao, Xiangyang Xu, Shulin Zhou, and Zhouyang Jia. 2017. ConfTest: Generating comprehensive misconfiguration for system reaction ability evaluation. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. 88–97.
- [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. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 609–630.
- [28] Zhenhao Li. 2020. Towards providing automated supports to developers on writing logging statements. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. 198–201.
- [29] Zhenhao Li, Tse-Hsun Chen, and Weiye Shang. 2020. Where shall we log? studying and suggesting logging locations in code blocks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 361–372.
- [30] Zhenhao Li, Heng Li, Tse-Hsun Chen, and Weiye Shang. 2021. DeepLv: Suggesting log levels using ordinal based neural networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1461–1472.
- [31] Xinyu Lian, Yinfang Chen, Runxiang Cheng, Jie Huang, Parth Thakkar, Minjia Zhang, and Tianyin Xu. 2024. Large Language Models as Configuration Validators. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 204–216.
- [32] Xiangke Liao, Shulin Zhou, Shanshan Li, Zhouyang Jia, Xiaodong Liu, and Haochen He. 2018. Do you really know how to configure your software? configuration constraints in source code may help. *IEEE Transactions on Reliability* 67, 3 (2018), 832–846.
- [33] Jiahao Liu, Jun Zeng, Xiang Wang, Kaihang Ji, and Zhenkai Liang. 2022. Tell: log level suggestions via modeling multi-level code block information. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 27–38.
- [34] Antonio Mastropaolo, Luca Pascarella, and Gabriele Bavota. 2022. Using deep learning to generate complete log statements. In *Proceedings of the 44th International Conference on Software Engineering*. 2279–2290.
- [35] Mirza Masfiquir Rahman and Ashish Kundu. 2024. Code Hallucination. *arXiv preprint arXiv:2407.04831* (2024).
- [36] Maven Repository. 2024. Maven Repository. <https://mvnrepository.com/>. Accessed: 2025-01-20.
- [37] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*. IEEE, 317–331.
- [38] 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*. 13–25.
- [39] SLF4J. 2024. SLF4J. <https://slf4j.org/>. Accessed: 2025-01-20.
- [40] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. 2020. Testing configuration changes in context to prevent production failures. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 735–751.
- [41] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatchalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic configuration management at facebook. In *Proceedings of the 25th symposium on operating systems principles*. 328–343.
- [42] Adrian-Victor Vevera, Andreea Cătălina CRĂCIUN, Mihail DUMITRACHE, Ionut SANDU, Carmen-Ionela ROTUNĂ, and Radu Alexandru BOSTAN. 2025. Cyber-security Challenges in Managing Domain Names. From DNS to ENS in the Web3 Era. *Romanian Cyber Security Journal* 7, 1 (2025), 97–112.
- [43] Teng Wang, Haochen He, Xiaodong Liu, Shanshan Li, Zhouyang Jia, Yu Jiang, Qing Liao, and Wang Li. 2023. Confainter: Static taint analysis for configuration options. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1640–1651.
- [44] Teng Wang, Zhouyang Jia, Shanshan Li, Si Zheng, Yue Yu, Erci Xu, Shaojiang Peng, and Xiangke Liao. 2023. Understanding and detecting on-the-fly configuration bugs. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 628–639.
- [45] 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.
- [46] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning

- in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.
- [47] Junjielong Xu, Ziang Cui, Yuan Zhao, Xu Zhang, Shilin He, Pinjia He, Liquan Li, Yu Kang, Qingwei Lin, Yingnong Dang, et al. 2024. UniLog: Automatic Logging via LLM and In-Context Learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.
  - [48] 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.
  - [49] 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.
  - [50] Tianyin Xu and Owolabi Legunsen. 2019. Configuration testing: Testing configuration values as code and with code. *arXiv preprint arXiv:1905.12195* (2019).
  - [51] 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.
  - [52] Kundi Yao, Guilherme B. de Pádua, Weiye Shang, Steve Sporea, Andrei Toma, and Sarah Sajedi. 2018. Log4perf: Suggesting logging locations for web-based systems' performance monitoring. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. 127–138.
  - [53] 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.
  - [54] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 293–306. <https://www.usenix.org/conference/osdi12/technical-sessions/presentation/yuan>
  - [55] 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.
  - [56] Jialu Zhang, Ruzica Piskac, Ennan Zhai, and Tianyin Xu. 2021. Static detection of silent misconfigurations with deep interaction analysis. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–30.
  - [57] 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.
  - [58] 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.
  - [59] 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.
  - [60] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 565–581.
  - [61] Renyi Zhong, Yichen Li, Jinxi Kuang, Wenwei Gu, Yintong Huo, and Michael R Lyu. 2024. Automated Defects Detection and Fix in Logging Statement. *arXiv preprint arXiv:2408.03101* (2024).
  - [62] 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.
  - [63] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. 2015. Learning to log: Helping developers make informed logging decisions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 415–425.