Studying Duplicate Logging Statements and Their Relationships with Code Clones

Zhenhao Li, *Student Member, IEEE,* Tse-Hsun (Peter) Chen, *Member, IEEE,* Jinqiu Yang, *Member, IEEE,* and Weiyi Shang, *Member, IEEE*

Abstract—Developers rely on software logs for a variety of tasks, such as debugging, testing, program comprehension, verification, and performance analysis. Despite the importance of logs, prior studies show that there is no industrial standard on how to write logging statements. In this paper, we focus on studying duplicate logging statements, which are logging statements that have the same static text message. Such duplications in the text message are potential indications of logging code smells, which may affect developers' understanding of the dynamic view of the system. We manually studied over 4K duplicate logging statements and their surrounding code in five large-scale open source systems: Hadoop, CloudStack, Elasticsearch, Cassandra, and Flink. We uncovered five patterns of duplicate logging code smells. For each instance of the duplicate logging code smell, we further manually identify the potentially problematic (i.e., require fixes) and justifiable (i.e., do not require fixes) cases. Then, we contact developers to verify our manual study result. We integrated our manual study result and developers' feedback into our automated static analysis tool, DLFinder, which automatically detects problematic duplicate logging code smells. We evaluated DLFinder on the five manually studied systems and three additional systems: Camel, Kafka and Wicket. In total, combining the results of DLFinder and our manual analysis, we reported 91 problematic duplicate logging code smell instances to developers and all of them have been fixed. We further study the relationship between duplicate logging statements, including the problematic instances of duplicate logging code smells, and code clones. We find that 83% of the duplicate logging code smell instances reside in cloned code, but 17% of them reside in micro-clones that are difficult to detect using automated clone detection tools. We also find that more than half of the duplicate logging statements reside in cloned code snippets, and a large portion of them reside in very short code blocks which may not be effectively detected by existing code clone detection tools. Our study shows that, in addition to general source code that implements the business logic, code clones may also result in bad logging practices that could increase maintenance difficulties.

Index Terms—log, code smell, duplicate log, code clone, static analysis, empirical study.

1 Introduction

Software logs are widely used in software systems to record system execution behaviors. Developers use the generated logs to assist in various tasks, such as debugging [1]–[3], testing [4]–[6], program comprehension [7], [8], system verification [9], [10], and performance analysis [11], [12]. A logging statement (i.e., code that generates a log) contains a static message, to-be-recorded variables, and log verbosity level. For example, in the logging statement: *logger.error("Interrupted while waiting for fencing command: " + cmd)*, the static text message is *"Interrupted while waiting for fencing command:"*, and the dynamic message is from the variable *cmd*, which records the command that is being executed. The logging statement is at the *error* level, which is the level for recording failed operations [13].

Even though developers have been analyzing logs for decades [14], there exists no industrial standard on how to write logging statements [3], [15]. Prior studies often focus on recommending where logging statements should be added into the code (i.e., *where-to-log*) [16], [17], and what information should be added in logging statements (i.e., *what-to-log*) [1], [8], [18]. A few recent studies [19], [20] aim to detect potential problems in logging statements. However, these studies often only consider the appropriateness of one single logging statement; while logs are typically analyzed in sequences or clusters [1], [11], [21]–[24]. In other words, we consider that the appropriateness of a log is also influenced by other logs that are generated in system execution.

In particular, an intuitive case of such influence is duplicate logs, i.e., multiple logs that have the same text message. Even though each log itself may be impeccable, duplicate logs, in some occasions, may affect developers' understanding of the dynamic view of the system. For example, as shown in Figure 1, there are two logging statements in two different *catch* blocks, which are associated with the same *try* block. These two logging statements have the same static text message and do not include any other error-diagnostic information. Thus, developers cannot easily distinguish what is the occurred exception when analyzing the produced logs. Since developers rely on logs for debugging and program comprehension [8], such duplicate logging statements may negatively affect developers' activities in maintenance and quality assurance.

To help developers improve logging practices, in this paper, we focus on studying duplicate logging statements in the source code. We conducted a manual study on five large-scale open source systems, namely Hadoop, CloudStack, ElasticSearch, Cassandra and Flink. We first used static analysis to identify all duplicate logging statements, which are defined as two or more logging statements that have the same static text message. We then manually study all the (over 4K) identified duplicate logging statements and

• *Z. Li, T. Chen J. Yang and W. Shang are with the Department of Computer Science and Software Engineering, Concordia University, Montreal, Quebec, Canada.*
E-mail: l_zhenha,peterc,jinqiuy,shang@encs.concordia.ca

```
...
} catch (AlreadyClosedException closedException) {
      s_logger.warn("Connection to AMQP service is lost.");
} catch (ConnectException connectException) {
      s_logger.warn("Connection to AMQP service is lost.");
}
...
```

Fig. 1. An example of duplicate logging code smell that we detected in CloudStack. The duplicate logging statements in the two *catch* blocks contain insufficient information (e.g., no exception type or stack trace) to distinguish the occurred exception.

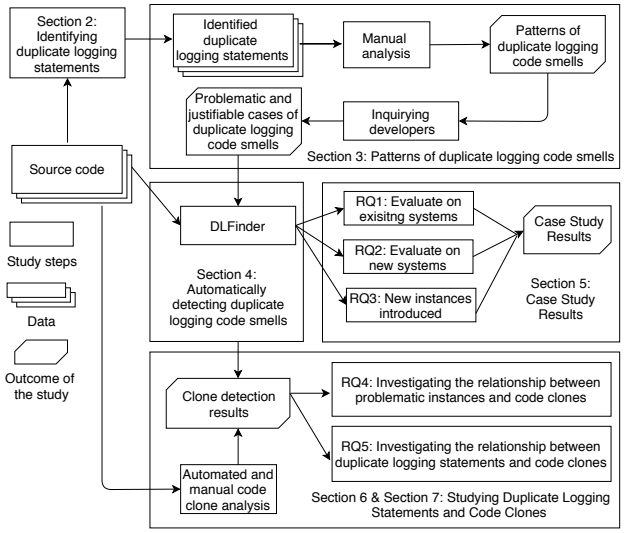


Fig. 2. The overall process of our study.

uncovered five patterns of *duplicate logging code smells*. We follow prior code smell studies [25], [26], and consider duplicate logging code smell as a "surface indication that usually corresponds to a deeper problem in the system". Thus, *not* all of the duplicate logging code smell are problematic and require fixes (i.e., *problematic duplicate logging code smells*). In particular, context (e.g., surrounding code and usage scenario of logging) may play an important role in identifying fixing opportunities. We further categorized duplicate logging code smells into potentially *problematic* or *justifiable* cases. In addition to our manual analysis, we sought confirmation from developers on the manual analysis result. For some of the potentially problematic duplicate logging code smells, developers considered them as technical debt and were reluctant to fix. For the rest of the potentially problematic instances, developers agreed that they are problematic and fixed them. For the justifiable ones, we communicated with developers for discussion (e.g., emails or posts on developers' forums).

We implemented a static analysis tool, DLFinder, to automatically detect *problematic* duplicate logging code smells. DLFinder leverages the findings from our manual study, including the uncovered patterns of duplicate logging code smells and the categorization on problematic and justifiable cases. We evaluated DLFinder on eight systems: five are from the manual study and three are additional systems (Kafka, Camel and Wicket). We also applied DLFinder on the updated versions of the five manually studied systems. The evaluation shows that the uncovered patterns of the duplicate logging code smells also exist in the additional systems, and duplicate logging code smells may be introduced over time. An automated approach such as DLFinder can help developers avoid duplicate logging code smells as systems evolve. In total, we reported 91 instances of duplicate logging code smell to developers and all the reported instances are fixed. Figure 2 shows the overall process of finding and detecting duplicate logging code smells.

We further investigate the relationship between the problematic instances of duplicate logging code smells and code clones. Intuitively, duplicate logging statements could be related to, or are even a consequence of code clones (e.g., logging statements can be copied along with other code since cloning is often performed hastily without much attention on the context [27]). The findings of our study may show other negative effect of code clones on logging statements and inspire future code clone and logging research. We combine both an automated code clone detection tool (i.e., NiCad [28]) and manual study on the eight studied systems to examine if the duplicate logging code smell instances reside in cloned code snippets. We find that 83% of the problematic duplicate logging code smell instances reside in cloned code snippets; however, 17% of the instances reside in very short code blocks that are difficult to detect using automated code detection tools.

In summary, this paper makes the following contributions:

- We uncovered five patterns of duplicate logging code smells through an extensive manual study on over 4K duplicate logging statements.
- We presented a categorization of duplicate logging code smells (i.e., problematic or justifiable), based on both our manual analysis and developers' feedback.
- We proposed DLFinder, a static analysis tool that integrates our manual study result and developers' feedback to detect problematic duplicate logging code smells.
- We reported 91 instances of problematic duplicate logging code smells to developers (DLFinder is able to detect 81 of them), and all of them are fixed.
- We found that most of the problematic instances of duplicate logging code smells (83.0%) reside in cloned code snippets, which indicates that code clones may also result in bad logging practices that increase maintenance difficulties.
- We found that more than half of the duplicate logging statements reside in cloned code snippets, and a large portion of them reside in short code blocks (i.e., micro-clones) which are difficult to detect using existing code clone detection tools.
- We found that duplicate logging statements have a non-negligible impact on helping the detection of code clones. After removing them, from 25.0% to 47.1% of the cloned code snippets with duplicate logging statements can not be detected as cloned code snippets.
- We provided a replication package of our paper for future studies on logging code and code clones[1].

1. We share the data of this paper at: https://github.com/SPEAR-SE/Duplicate_Logs_Data

Our study provides an initial step on creating a logging guideline for developers to improve the quality of logging code. DLFinder is also able to detect duplicate logging code smells with high precision and recall. Future code clone studies should also consider other possible side effects of code clones (e.g., understanding system runtime behaviour), and may consider including information from other software artifacts (e.g., duplicate logging statements) to further improve clone detection results.

This work extends our previous work [29]. First, we add one more system to our manual study and extend our evaluation to include an additional system and compare our text-analysis-based algorithm on detecting inconsistently updated log messages with two baselines. We also add discussions on duplicate logging statements that do not belong to one of the uncovered smells. Second, we study the relationship between duplicate logging statements, including the problematic instances of duplicate logging code smells, and code clones Finally, we investigate the potential impact between duplicate logging statements and code clones.

Paper organization. Section 2 describes data preparation and the studied systems. Section 3 discusses the process and the results of our manual study. Section 4 discusses the implementation details of DLFinder. Section 5 presents the case study results. Section 6 investigates the relationship between problematic instances of duplicate logging code smells and code clones. Section 7 investigates the relationship between duplicate logging statements and code clones, as well as the potential impact of duplicate logging statements on detecting code clones. Section 8 discusses the threats to validity of our study. Section 9 surveys related work. Section 10 concludes the paper. Appendix A discusses the false positive rate of the automated clone detection tool.

2 Identifying Duplicate Logging Statements for Manual Study

Definition and how to identify duplicate logging statements. We define duplicate logging statements as logging statements that have identical static text messages. We focus on studying the log message because such semantic information is crucial for log understanding and system maintenance [8], [30]. As an example, the two following logging statements are considered duplicate: "*Unable to create a new ApplicationId in SubCluster*" + ***subClusterId.getId()***, and "*Unable to create a new ApplicationId in SubCluster*" + ***id***.

To prepare for a manual study, we identify duplicate logging statements by analyzing the source code using static analysis. In particular, the static text message of each logging statement is built by concatenating all the strings (i.e., constants and values of string variables) and abstractions of the non-string variables. We also extract information to support the manual analysis, such as the types of variables that are logged, and the log level (i.e., *fatal*, *error*, *warn*, *info*, *debug*, or *trace*). Log levels can be used to reduce logging overheads in production (e.g., only record *info* and more severe levels) and may target different phases of software maintenance (e.g., *debug* logs may be used for debugging and *info* logs may provide information for general audience) [30], [31]. If two or more logging statements have the same static text message, they are identified as duplicate logging statements.

Studied systems. Table 1 shows the statistics of the studied systems. We identify duplicate logging statements from the top five large-scale open source Java systems in the table for our manual analysis: Hadoop, CloudStack, Elasticsearch, Cassandra and Flink which are commonly used in prior studies for log-related research [19], [20], [32]. The studied systems also use popular Java logging libraries (e.g., Log4j [13] and SLF4J [33]). Hadoop is a distributed computing framework, CloudStack is a cloud computing platform, Elasticsearch is a distributed search engine, Cassandra is a NoSQL database system, and Flink is a stream-processing framework. These systems belong to different domains and are well maintained. We study all Java source code files in the main branch of each system and exclude test files, since we are more interested in studying duplicate logging statements that may affect log understanding in production. In general, we find that there is a non-negligible number of duplicate logging statements in the studied systems (6% to 20%). The median number of words in the duplicate logging statements are similar to that of non-duplicate logging statements (i.e., both range from 6 to 8 words).

3 Patterns of Duplicate Logging Code Smells

In this section, we conduct a manual study to investigate duplicate logging statements. Note that duplicate logging statements may not necessarily be a problem. Hence, our goal is to uncover patterns of potential code smells that may be associated with duplicate logging statements (i.e., *duplicate logging code smells*). Similar to prior code smell studies, we consider duplicate logging code smells as a *"surface indication that usually corresponds to a deeper problem in the system"* [25], [26]. Such duplicate logging code smells may be indications of logging problems that require fixes.

We categorize each duplicate logging code smell instance as either problematic (i.e., require fixes) or justifiable (i.e., do not require fixes), by understanding the surrounding code. Not every duplicate logging code smell is problematic. Intuitively, one needs to consider the code context to decide whether a code smell instance is problematic and requires fixes. As shown in prior studies [3], [16], [32], logging decisions, such as log messages and log levels, are often associated with the structure and semantics of the surrounding code. In addition to the manual analysis by the authors, we also ask for developers' feedback regarding both the problematic and justifiable cases. By providing a more detailed understanding of code smells, we may better assist developers to improve logging practices and inspire future research.

Manual study process. We conduct a manual study by analyzing all the duplicate logging statements in the five studied systems. In total, we studied 1,371 sets of duplicate logging statements (more than 4K logging statements in total; each set contains two or more logging statements with the same static message). Specifically, we examine the four following criteria when studying the code snippets: 1) the generated log messages record incorrect information (i.e., the recorded method name is different from the method where the log message is generated), 2) the recorded information cannot be used to distinguish the occurred errors

TABLE 1
An overview of the studied systems. The top five systems are used for manual study in Section 3. The bottom three systems are used for evaluating our duplicate logging code smell detection tool in Section 5.

System	Version	Release date	LOC	Num. of logs	Num. of dup. logs	Num. of dup. log sets	Med. words in dup. logs	Med. words in non-dup. logs
Cassandra	3.11.1	Oct. 2017	358K	1.6K	113 (7%)	46	7	7
CloudStack	4.9.3	Aug. 2017	1.18M	11.7K	2.3K (20%)	865	8	8
Elasticsearch	6.0.0	Nov. 2017	2.12M	1.7K	94 (6%)	40	6	7
Flink	1.7.1	Dec. 2018	177K	2.5K	467 (11%)	203	6	7
Hadoop	3.0.0	Nov. 2017	2.69M	5.3K	496 (9%)	217	6	6
Camel	2.21.1	Apr. 2018	1.68M	7.3K	2.3K (32%)	886	6	6
Kafka	2.1.0	Nov. 2018	542K	1.5K	406 (27%)	104	5	7
Wicket	8.0.0	May. 2018	381K	0.4K	45 (11%)	21	6	8

(e.g., to distinguish different exception types), 3) there are inconsistencies in terms of log levels or the recorded debugging information, and 4) the duplicated log message may need to be updated to ensure consistency (i.e., maintenance of logs).

The process of our manual study involves five phases:

Phase I: The first two authors manually studied 301 randomly sampled (based on 95% confidence level and 5% confidence interval [34]) sets of duplicate logging statements and the surrounding code to derive an initial list of duplicate logging code smell patterns. All disagreements were discussed until a consensus was reached.

Phase II: The first two authors *independently* categorized *all* of the 1,371 sets of duplicate logging statements to the derived patterns in Phase I. We did not find any new patterns in this phase. The results of this phase have a Cohen's kappa of 0.811, which is a substantial-level of agreement [35].

Phase III: The first two authors discussed the categorization results obtained in Phase II. All disagreements were discussed until a consensus was reached.

Phase IV: The first two authors further studied all logging code smell instances that belong to each pattern to identify justifiable cases that may not need fixes. The instances that do not belong to the category of justifiable are considered potentially problematic and may require fixes.

Phase V: We verified both the problematic and justifiable instances of logging code smells with developers by creating pull requests, sending emails, or posting our findings on developers' forums (e.g., Stack Overflow). We reported every instance that we believe to be problematic (i.e., require fixes), and reported a number of instances for each justifiable category.

Results. In total, we uncovered five patterns of duplicate logging code smells. Table 2 lists the uncovered code smell patterns and the corresponding examples. Table 3 shows the number of problematic code smell instances for each pattern that we manually found. Below, we discuss each pattern according to the following template:

Description: A description of the pattern of duplicate logging code smell.

Example: An example of the pattern.

Code smell instances: Discussions on the manually-uncovered code smell instances. We also discuss the justifiable cases if we found any.

Developers' feedback: A summary of developers' feedback on both the problematic and justifiable cases.

Pattern 1: Inadequate information in catch blocks (IC).

Description. Developers usually rely on logs for error diagnostics when exceptions occur [36]. However, we find that sometimes, duplicate logging statements in different *catch* blocks of the same *try* block may cause debugging difficulties since the logs fail to tell which exception occurred.

Example. As shown in Table 2, in the `ParamProcessWorker` class in CloudStack, the *try* block contains two *catch* blocks; however, the log messages in these two *catch* blocks are identical. Since both the exception message and stack trace are not logged, once one of the two exceptions occurs, developers may encounter difficulties in finding the root causes and determining the occurred exception.

Code smell instances. After examining all the instances of IC, we find that all of them are potentially problematic and require fixes. For all the instances of IC, none of the exception type, exception message, and stack trace are logged.

Developers' feedback. We reported all the problematic instances of IC (15 instances), and all of them are fixed by adding more error diagnostic information (e.g., stack trace) into the logging statements. Developers agree that IC will cause confusion and insufficient information in the logs, which may increase the difficulties of error diagnostics.

Pattern 2: Inconsistent error-diagnostic information (IE).

Description. We find that sometimes duplicate logging statements for recording exceptions may contain inconsistent error-diagnostic information (e.g., one logging statement records the stack trace and the other does not), even though the surrounding code is similar.

Example. As shown in Table 2, the two classes in CloudStack: `Create`**`PortForwarding`**`RuleCmd` and `Create`**`Firewall`**`RuleCmd` have similar functionalities. The two logging statements have the same static text message and are in methods with identical names (i.e., *create()*, not shown due to space restriction). The *create()* method in `Create`**`PortForwarding`**`RuleCmd` is about creating rules for port forwarding, and the method in `Create`**`Firewall`**`RuleCmd` is about creating rules for firewalls. These two methods have very similar code structure and business logic. However, the two logging statements record different information: One records the stack trace information and the other one only records the exception message (i.e., *ex.getMessage()*). Since the two logging statements have similar context, the error-diagnostic information recorded by the logs may need to be consistent for the ease of debugging. We reported this example, which is now fixed to have consistent error-diagnostic information.

Code smell instances. We find 25 instances of IE (Table 3), and six of them are considered problematic in our manual

TABLE 2
Patterns of duplicate logging code smells and corresponding examples.

Name	Example
Inadequate information in catch blocks (IC)	`catch (final IllegalArgumentException e) { s_logger.error("Error initializing command " + cmd.getCommandName() + ", field " + field.getName() + " is not accessible."); ... } catch (final IllegalAccessException e) { s_logger.error("Error initializing command " + cmd.getCommandName() + ", field " + field.getName() + " is not accessible."); ... }` ***Log message cannot be used to distinguish which exception occurred***
Inconsistent error-diagnostic information (IE)	`public class CreatePortForwardingRuleCmd{ ... } catch (NetworkRuleConflictException ex) { s_logger.info("Network rule conflict: " + ex.getMessage()); ... }` ***Same log message and similar surrounding code, but record different error diagnostic information*** `public class CreateFirewallRuleCmd{ ... } catch (NetworkRuleConflictException ex) { s_logger.info("Network rule conflict: ", ex); ... }`
Log message mismatch (LM)	`public void doScaleUp() { ... s_logger.error("Can not find the groupid " + groupId + " for scaling up"); ... }` ***Match*** `public void doScaleDown() { ... s_logger.error("Can not find the groupid " + groupId + " for scaling up"); ... }` ***Mismatch*** ***A copy-and-paste error, scaling up is the behaviour of doScaleUp()***
Inconsistent log level (IL)	`public AllSSTableOpStatus performCleanup(){ ... if (!StorageService.instance.isJoined()){ logger.info("Cleanup cannot run before a node has joined the ring"); return AllSSTableOpStatus.ABORTED; } ... }` ***Log levels are different in two very similar methods*** `public void forceUserDefinedCleanup(){ ... if (!StorageService.instance.isJoined()){ logger.error("Cleanup cannot run before a node has joined the ring"); return; } ... }`
Duplicate log in polymorphism (DP)	`public class PowerShellFencer extends Configured implements FenceMethod { ... } catch (InterruptedException ie) { LOG.warn("Interrupted while waiting for fencing command: " + ps1script); ... }` `public class ShellCommandFencer extends Configured implements FenceMethod { ... } catch (InterruptedException ie) { LOG.warn("Interrupted while waiting for fencing command: " + cmd); ... }` ***Both implementations of FenceMethod have the same log message***

TABLE 3
Number of problematic instances (Prob.) verified by our manual study and developers' feedback, number of instances of technical debt (Tech.), and total number of instances (Total) including non-problematic instances.

	IC		IE		LM		IL		DP	
	Prob.	Total	Prob.	Total	Prob.	Total	Prob.	Total	Tech.	Total
Cassandra	1	1	0	1	0	0	0	3	2	2
CloudStack	8	8	4	14	27	27	0	47	107	107
Elasticsearch	1	1	0	5	1	1	0	9	3	3
Flink	0	0	2	5	4	4	0	14	24	24
Hadoop	5	5	0	0	9	9	0	17	27	27
Total	15	15	6	25	41	41	0	90	163[1]	163

[1] Developers acknowledged the problem as technical debt, because systematic refactoring of DP would require supports from logging libraries. Therefore, we did not report all the instances of DP.

study. From the remaining instances of IE, we find three justifiable cases that may not require fixes.

Justifiable case IE.1: Duplicate logging statements record general and specific exceptions. For 11/25 instances of IE, we find that the duplicate logging statements are in the *catch* blocks of different types of exception. In particular, one duplicate logging statement is in the *catch* block of a generic exception (i.e., the `Exception` class in Java) and the other one is in the *catch* block of a more specific exception (e.g., application-specific exceptions such as `CloudRuntimeException`). In all of the 11 cases, we find that one log would record the stack trace for `Exception`, and the duplicate log would only record the type of the occurred exception (e.g., by calling *e.getMessage()*) for a more specific exception. The rationale may be that generic exceptions, once occurred, are often not expected by developers [36], so it is important that developers record more error-diagnostic information.

Justifiable case IE.2: Duplicate logging statements are in the same catch block for debugging purposes. For 6/25 instances of IE, the duplicate logging statements are *in the same catch* block and developers' intention is to use a duplicate logging statement at *debug* level to record rich error-diagnostic information such as stack trace (and the log level of the other logging statement could be *error*). The extra logging statements at *debug* level help developers debug the occurred exception and reduce logging overhead in production [31] (i.e., logging statements at debug level are turned off).

Justifiable case IE.3: Having separate error-handling classes. For 2/25 instances, we find that the error-diagnostic information is handled by creating an object of an error-handling class. As an example from CloudStack:

```
public final class LibvirtCreateCommandWrapper {
   ...
      } catch (final CloudRuntimeException e) {
         s_logger.debug("Failed to create volume: " +
              e.toString());
         return new CreateAnswerErrorHandler(command, e);
      }
   ...
}

public class KVMStorageProcessor {
   ...
      } catch (final CloudRuntimeException e) {
         s_logger.debug("Failed to create volume: ", e);
         return new CopyCmdAnswerErrorHandler(e.toString());
      }
   ...

}
```

In this example, extra logging is added by using error-handling classes (i.e., `CreateAnswerErrorHandler` and `CopyCmdAnswerErrorHandler`) to complement the logging statements. As a consequence, the *actual* logged information is consistent in these two methods: One method records *e.toString()* in the logging statement and records the exception variable *e* through an error-handling class; the other method records *e* in the logging statement and records *e.toString()* through an error-handling class.

Developers' feedback. We reported all the six instances of IE that we consider problematic to developers, all of which are fixed. Moreover, we ask developers whether our conjecture was correct for each of the justifiable cases of IE. Developers confirmed our observation on the justifiable cases. They agreed that those cases are not problematic thus do not require fixes.

Pattern 3: Log message mismatch (LM).

Description. Sometimes after developers copy and paste a piece of code to another method or class, they may forget to change the log message. This results in having duplicate logging statements that record inaccurate system behaviors.

Example. As an example, in Table 2, the method *doScaleDown()* is a code clone of *doScaleUp()* with very similar code structure and minor syntactical differences. However, developers forgot to change the log message in *doScaleDown()*, after the code was copied from *doScaleUp()* (i.e., both log messages contain *scaling up*). Such instances of LM may cause confusion when developers analyze the logs.

Code smell instances. We find that there are 41 instances of LM that are caused by copying-and-pasting the logging statement to new locations without proper modifications. For all the 41 instances, the log message contains words of incorrect class or method name that may cause confusion when analyzing logs.

Developers' feedback. Developers agree that the log messages in LM should be changed in order to correctly record the execution behavior (i.e., update the copy-and-pasted log message to contain the correct class/method name). We reported all the 41 instances of LM that we found and all of them are fixed.

Pattern 4: Inconsistent log level (IL).

Description. Log levels (e.g., *fatal*, *error*, *info*, *debug*, or *trace*) allow developers to specify the verbosity of the log message and to reduce logging overhead when needed [31]. A prior study [30] shows developers frequently modify log levels to find the most adequate level. We find that there are duplicate logging statements that, even though the log messages are exactly the same, the log levels are different.

Example. In the IL example shown in Table 2, the two methods, which are from the same class `CompactionManager`, have very similar functionality (i.e., both try to perform cleanup after compaction), but different log levels.

Code smell instances. We find three justifiable cases in IL that may be developers' intended behavior. We do not find problematic instances of IL after communicating with developers – Developers think the problematic instances identified by our manual analysis may not be problems.

Justifiable case IL.1: Duplicate logging statements are in the catch blocks of different types of exception. Similar to what we observed in IE, we find that for 9/90 instances, the log level

for a more generic exception is usually more severe (e.g., *error* level for the generic Java `Exception` and *info* level for an application-specific exception). Generic exceptions might be unexpected to developers [36], so developers may use a higher log level (e.g., *error*) to record exception messages.

Justifiable case IL.2: Duplicate logging statements are in different branches of the same method. There are 42/90 instances belong to this case. Below is an example from Elasticsearch, where a set of duplicate logging statements occur in the same method but in different branches.

```
if (lifecycle.stoppedOrClosed()) {
    logger.trace("failed to send ping transport message",
        e);
} else {
    logger.warn("failed to send ping transport message",
        e);
}
```

In this case, developers already know the desired log level and intend to use different log levels due to the difference in execution (i.e., in the if-else block).

Justifiable case IL.3: Duplicate logging statements are followed by error-handling code. There are 19/90 instances that are observed to have such characteristics: In a set of duplicate logging statements, some statements have log levels of higher verbosity, and others have log levels of lower verbosity. However, the duplicate logging statement with lower verbosity log level is followed by additional error handling code (e.g., *throw a new Exception(e);*). Therefore, the error is handled elsewhere (i.e., the exception is re-thrown), and may be recorded at a higher-verbosity log level.

Developers' feedback. In all the instances of IL that we found, developers think that IL may not be a problem. In particular, developers agreed with our analysis on the justifiable cases. However, developers think the problematic instances of IL from our manual analysis may also not be problems. We concluded the following two types of feedback from developers on the "suspect" instances of IL (i.e., 20 problematic ones from our manual analysis out of the 90 instances of IL). The first type of developers' feedback argues the importance of semantics and usage scenario of logging in deciding the log level. A prior study [30] suggests that logging statements that appear in syntactically similar code, but with inconsistent log levels, are likely problematic. However, based on the developers' feedback that we received, IL still may not be a concern, even if the duplicate logging statements reside in very similar code. A developer indicated that "conditions and messages are important but the *context* is even more important". As an example, both of the two methods may display messages to users. One method may be displaying the message to *local* users with a *debug* logging statement to record failure messages. The other method may be displaying the message to *remote* users with an *error* logging statement to record failure messages (problems related to remote procedure calls may be *more severe* in distributed systems). Hence, even if the code is syntactically similar, the log level has a reason to be different due to the different semantics and purposes of the code (i.e., referred to as different *contexts* in developers' responses). Our findings show that future studies should consider both the syntactic structure and semantics of the code when suggesting log levels.

The second type of developers' feedback acknowledges the inconsistency. However, developers are reluctant to fix such inconsistencies since developers do not view them as concerns. For example, we reported the instance of IL in Table 2 to developers. A developer replied: "I think it should probably be an *ERROR* level, and I missed it in the review (could make an argument either way, I do not feel strongly that it should be *ERROR* level vs *INFO* level." Our opinions (i.e., from us and prior studies [30], [31]) differ from that of developers' regarding whether such inconsistencies are problematic. On one hand, whether an instance of IL is problematic or not can be subjective. This shows the importance of including perspectives from multiple parties (e.g., user studies or interviews) in future studies of software logging practice. On the other hand, the discrepancy also indicates the need of establishing a guidance for logging practice and further even enforcing such standard. In short, none of the IL instances that we manually identified are problematic based on developers' feedback.

Pattern 5: Duplicate logging statements in polymorphism (DP).

Description. Classes in object-oriented languages are expected to share similar functionality if they inherit the same parent class or if they implement the same interface (i.e., polymorphism). Since log messages record a higher level abstraction of the program [8], we find that even though there are no clones among a parent method and its overridden methods, such methods may still contain duplicate logging statements. Such duplicate logging statements may cause maintenance overhead. For example, when developers update one log message, they may forget to update the log message in all the other sibling classes. Inconsistent log messages may cause problems during log analysis [20], [37].

Example. In Table 2, the two classes (`PowerShellFencer` and `ShellCommandFencer`) in Hadoop both extend the same parent class, implement the same interface, and share similar behaviors. The inherited methods in the two classes have identical log message. However, as the system evolves, developers may not always remember to keep the log messages consistent, which may cause problems during system debugging, understanding, and analysis.

Code smell instances. We find that all the 163 instances of DP are potentially problematic that may be fixed by refactoring. In most of the instances, the parent class is an abstract class, and the duplicate logging statements exist in the overridden methods of the subclasses. We also find that in most cases, the overridden methods in the subclasses are very similar with minor differences (e.g., to provide some specialized functionality), which may be the reason that developers use duplicate logging statements.

Developers' feedback. Developers agree that DP is associated with logging code smells and specific refactoring techniques are needed. One developer comments that:

"You want to care about the logging part of your code base in the same way as you do for business-logic code (one can argue it is part of it), so salute DRY (do-not-repeat-yourself)."

Based on developers' feedback, DP is viewed more as technical debts [38], while resolving DP often requires systematic refactoring. However, to the best of our knowledge, current Java logging frameworks, such as SLF4J and Log4j 2, do not support the use of polymorphism in logging

statements. Thus, we find that developers are more reluctant to fix DP. The way to resolve DP is to ensure that the log message of the parent class can be reused by the subclasses, e.g., storing the log message in a static constant variable. We received similar suggestions from developers on how to refactor DP, such as *"adding a method in the parent class that generates the error text for that case: logger.error(notAccessible(field.getName()));"*, or *"creat[ing] your own Exception classes and put message details in them"*. We find that without supports from logging frameworks, even though developers acknowledged the issue of DP, they do not want to *manually* fix the code smells. Similar to some code smells studied in prior research [39], [40], developers may be reluctant to fix DP due to additional maintenance overheads but limited supports (i.e., need to manually fix hundreds of DP instances). Therefore, we refer to the instances of DP as technical debts, instead of problematic instances, in the rest of the paper. In short, logging frameworks should provide better support to developers in creating log "templates" that can be reused in different places in the code.

Discussions on duplicate logging statements that do not belong to one of the uncovered smells. In this paper, we focus on studying the problematic patterns of duplicate logging statements. However, we do not consider all duplicate logging statements as bad logging practice. For other duplicate logging statements that do not belong to the identified smells, we did not find evidence that they may cause confusion when analyzing logs. In most of the cases, the log message may be similar by coincidence (e.g., the log messages are used to record a certain type of exception message and stack trace). In some cases, we found that developers intentionally write duplicate logging statements with comments explaining the reasons. For example, some developers mentioned in the comment that the code snippet is copied from another class, and said the code should be refactored in the future. In some other cases, developers described the intention of the two duplicate logging statements. Although the static messages are identical, the comments are different, which shows that duplicate logging statements could have different intentions in different places. In such cases, duplicate logging statements may assist machine-learning based approaches to suggest where-to-log.

> We manually uncovered five patterns of duplicate logging code smells. In total, our manual study helped developers fix 62 problematic duplicate logging code smells in the studied systems.

4 DLFinder: Automatically Detecting Problematic Duplicate Logging Code Smells

Section 3 uncovers five patterns of duplicate logging code smells, and provides guidance in identifying *problematic* logging code smells. To help developers detect such problematic code smells and improve logging practices, we propose an automated approach, specifically a static analysis tool, called DLFinder. DLFinder uses abstract syntax tree (AST) analysis, data flow analysis, and text analysis. Note that we exclude the detection result of IL (i.e., inconsistent log level) in this study, since based on the feedback from developers, none of the IL instances are problematic. Below, we discuss how DLFinder detects each of the four patterns of duplicate logging code smell (i.e., IC, IE, LM, and DP).

Detecting inadequate information in catch blocks (IC). DLFinder first locates the *try-catch* blocks that contain duplicate logging statements. Specifically, DLFinder finds the *catch* blocks of the same *try* block that catch different types of exceptions, and these *catch* blocks contain the same set of duplicate logging statements. Then, DLFinder uses data flow analysis to analyze whether the handled exceptions in the *catch* blocks are logged (e.g., record the exception message). DLFinder detects an instance of IC if none of the logging statements in the *catch* blocks record either the stack trace or the exception message.

Detecting inconsistent error-diagnostic information (IE). DLFinder first identifies all the *catch* blocks that contain duplicate logging statements. Then, for each *catch* block, DLFinder uses data flow analysis to determine how the exception is logged by analyzing the usage of the exception variable in the logging statement. Namely, the logging statement records 1) the entire stack trace, 2) only the exception message, or 3) nothing at all. Then, DLFinder compares how the exception variable is used/recorded in each of the duplicate logging statements. DLFinder detects an instance of IE if a set of duplicate logging statements that appear in *catch* blocks has an inconsistent way of recording the exception variables (e.g., the log in one *catch* block records the entire stack trace, and the log in another *catch* block records only the exception message, while the two catch blocks handle the same type of exception). Note that for each instance of IE, the multiple *catch* blocks with duplicate logging statements in the same set may belong to different *try* blocks. In addition, DLFinder decides if an instance of IE can be excluded if it belongs to one of the three justifiable cases (IE.1–IE.3) by checking the exception types, if the duplicate logging statements are in the same *catch* block, and if developers pass the exception variable to another method.

Detecting log message mismatch (LM). LM is about having an incorrect method or class name in the log message (e.g., due to copy-and-paste). Hence, DLFinder analyzes the text in both the log message and the class-method name (i.e., concatenation of class name and method name) to detect LM by applying commonly used text analysis approaches [41]. DLFinder detects instances of LM using four steps: 1) For each logging statement, DLFinder splits class-method name into a set of words (i.e., *name set*) and splits log message into a set of words (i.e., *log set*) by leveraging naming conventions (e.g., camel cases) and converting the words to lower cases. 2) DLFinder applies stemming on all the words using Porter Stemmer [42]. 3) DLFinder removes stop words in the log message. We find that there is a considerable number of words that are generic across the log messages in a system (e.g., on, with, and process). Hence, we obtain the stop words by finding the top 50 most frequent words (our studied systems has an average of 3,352 unique words in the static text messages) across all log messages in each system [43]. 4) For every logging statement, between the name set (i.e., from the class-method name) and its associated log set, DLFinder counts the number of common words shared

TABLE 4
The results of DLFinder in RQ1 and RQ2. In each pattern, Pro. is the number of problematic instances as the ground-truth, Tech. is the number of technical debt instances for DP, C.Det. is the combined number of problematic or technical debt instances correctly detected by DLFinder, and Det. is the number of instances detected by DLFinder.

Research questions		IC			IE			LM			DP		
		Pro.	C.Det.	Det.	Pro.	C.Det.	Det.	Pro.	C.Det.	Det.	Tech.	C.Det.	Det.
RQ1: How well can DLFinder detect duplicate logging code smells in the five manually studied systems?	**Cassandra**	1	1	1	0	0	0	0	0	4	2	2	2
	CloudStack	8	8	8	4	4	4	27	24	186	107	107	107
	Elasticsearch	1	1	1	0	0	0	1	0	15	3	3	3
	Flink	0	0	0	2	2	2	4	4	41	24	24	24
	Hadoop	5	5	5	0	0	0	9	7	44	27	27	27
	Total of RQ1	15	15	15	6	6	6	41	35	290	163	163	163
	Precision / Recall		100% / 100%			100% / 100%			12.1% / 85.4%			100% / 100%	
RQ2: How well can DLFinder detect duplicate logging code smells in the additional systems?	**Camel**	1	1	1	0	0	0	14	10	95	29	29	29
	Kafka	0	0	0	0	0	0	3	3	15	14	14	14
	Wicket	1	1	1	0	0	0	1	1	4	1	1	1
	Total of RQ2	2	2	2	0	0	0	18	14	114	44	44	44
	Precision / Recall		100% / 100%			- / -			12.3% / 77.8%			100% / 100%	
	Total	17	17	17	6	6	6	59	49	404	207	207	207

by both sets. Afterward, DLFinder detects an instance of LM if the number of common words is inconsistent among the duplicate logging statements in one set.

For the LM example shown in Table 2, the common words shared by the first pair (i.e., method *doScaleUp()* and its log) are "scale, up", while the common word shared by the second pair is "scale". Hence, DLFinder detects an LM instance due to this inconsistency. The rationale is that the number of common words between the class-method name and the associated logging statement is subject to change if developers make copy-and-paste errors on logging statements (e.g., copy the logging statement in *doScaleUp()* to method *doScaleDown()*), but forget to update the log message to match with the new method name "doScaleDown". However, the number of common words will remain unchanged (i.e., no inconsistency) if the logging statement (after being pasted at a new location) is updated respectively.

Detecting duplicate logs in polymorphism (DP). DLFinder generates an object inheritance graph when statically analyzing the Java code. For each overridden method, DLFinder checks if there exist any duplicate logging statements in the corresponding method of the sibling and the parent class. If there exist such duplicate logging statements, DLFinder detects an instance of DP. Note that, based on the feedback that we received from developers (Section 3), we do not expect developers to fix instances of DP. DP can be viewed more as technical debts [38] and our goal is to propose an approach to detect DP to raise the awareness from the research community and developers regarding this issue.

5 Case Study Results

In this section, we conduct a case study to investigate the prevalence of duplicate logging code smells and evaluate DLFinder by answering three research questions.

RQ1: How well can DLFinder detect duplicate logging code smells in the five manually studied systems?

Motivation. DLFinder was implemented based on the duplicate logging code smells uncovered from the manually studied systems (i.e., IC, IE, LM, and DP). Since we obtain the ground truth (i.e., all the duplicate logging code smell instances) in these five systems from our manual study, the goal of this RQ is to evaluate the detection accuracy of DLFinder.

Approach. We applied DLFinder on the same versions of the systems that we used in our manual study (Section 3). We calculated the precision and recall of DLFinder in detecting problematic instances for IC, IE, and LM, as well as the technical debt instances for DP. Precision is the percentage of correctly detected instances among all the detected instances, and recall is the percentage of problematic or technical debt instances that DLFinder is able to detect.

Results and discussion. The first five rows of Table 4 show the results of RQ1. For the patterns of IC, IE, and DP, DLFinder detects all the problematic and technical debt instances of duplicate logging code smells (100% in recall) with a precision of 100%. For the LM pattern, DLFinder achieves a recall of 85.4% (i.e., DLFinder detects 35/41 problematic LM instances). We manually investigate the six instances of LM that DLFinder cannot detect. We find that the problem is related to the various habits and coding conventions that developers use when writing log messages. For example, developers may write "mlockall" instead of "mLockAll" (i.e., the camelcase naming convention), which increases the challenge of log message analysis. Hence, the text in the log message cannot be matched with the method name when we split the word using camel cases. The precision of detecting problematic LM instances is modest because, in many false positive cases, the log messages and class-method names are at different levels of abstraction: The log message describes a local code block while the class-method name describes the functionality of the entire method. For example, *encodePublicKey()* and *encodePrivateKey()* both contain the duplicate logging statement *"Unable to create KeyFactory"*. The duplicate logging statement describes a local code block that is related to the usage of the *KeyFactory* class, which is different from the major functionalities of the two methods (i.e., as expressed by their class-method names). Nevertheless, DLFinder detects the LM instances with a high recall, and developers could quickly go through the results to identify the true positives (it took the first two authors less than 10 minutes on average to go through the LM result of each system to identify true positives).

To further evaluate our detection approach for LM, we compare our detection results with a baseline. We use random prediction algorithm as our baseline, which is commonly used as the baseline in prior studies [44]–[46]. The random prediction algorithm predicts the label of an item (i.e., whether a set of duplicate logging statements belong to LM) based on the distribution of the training data. For each system, we use our manually labeled results (which are discussed and verified in the previous sections) as the

training data. Note that we only compare the detection results of LM with the baseline. The reason is that pattern IC, IE, and DP are relatively independent and well-defined, unlike LM which depends on the semantics of the logging statement and its surrounding code. We repeat the random prediction 30 times (as suggested by previous studies [47], [48]) for each system to reduce the biases. Finally, we report the average precision and recall that are computed based on the 30 times of iterations. Figure 3 shows how the precision and recall of our approach compared to that of the baseline. The average precision and recall for the baseline are 3.1% and 3.0%, respectively, for the five studied systems. Our detection approach achieves a precision and recall of 12.1% and 85.4%, respectively. In short, our approach is better than the baseline and is able to have a very high recall in the five manually studied systems.

RQ2: How well can DLFinder detect duplicate logging code smells in the additional systems?

Motivation. The goal of this RQ is to study whether the uncovered patterns of duplicate logging code smells are generalizable to other systems.

Approach. We applied DLFinder to three additional systems that are not included in the manual study in Section 3: Camel, Kafka, and Wicket, which are all large-scale open source Java systems. Details of the systems are presented in Table 1. Similar to our manual study, the first two authors of this paper manually collect the problematic and technical debt duplicate logging code smells in the additional systems, i.e., the ground-truth used for calculating the precision and recall of DLFinder. Note that the collected ground-truth of the additional systems is only used in this evaluation, but not in designing the patterns in DLFinder (There are also no new patterns found in this process).

Results and discussion. The second half of Table 4 shows the results of the additional systems. In total, we found 20 problematic duplicate logging code code smell instances (DLFinder detects 16) in these systems and all of them are reported and fixed. Compared to the five systems in RQ1, DLFinder has similar precision and recall values in the additional systems. DLFinder detects DP instances with 100% in recall and precision; however, developers are reluctant to fix them due to limited support from logging frameworks. Similar to our observation in RQ1, we find that DLFinder cannot detect some LM instances due to the various habits and coding conventions when developers write log messages. We also compare our LM detection results with the baseline mentioned in RQ1 using the same approach. The average precision and recall for DLFinder are 12.3% and 77.8%, respectively, which are considerably better than the precision (2.2%) and recall (2.1%) of the baseline.

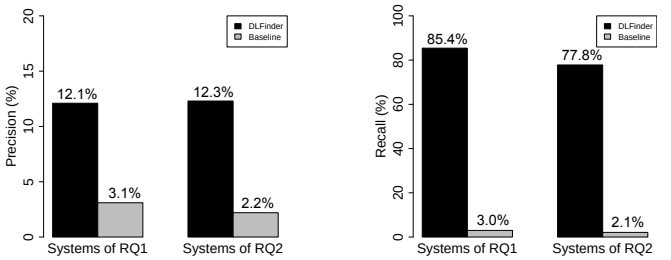
RQ3: Are new duplicate logging code smell instances introduced over time?

Motivation. In this RQ, we investigate if new instances of duplicate logging code smell are introduced during the evolution of systems. An automated detection tool may then help developers detect such problems overtime.

Approach. We applied DLFinder on the latest versions of the five studied systems, i.e., Hadoop, CloudStack, Elasticsearch, Cassandra and Flink, and compare the results with the ones on previous versions. The gaps of days between the manually studied versions and the new versions vary from 77 days to 301 days.

TABLE 5
The result of RQ3: applying DLFinder to the newer versions of the studied systems. Gap. shows the duration of time in days between the original (Org.) and the newer release (New.)

	Releases		IC	IE	LM	DP
	Org., New.	Gap.				
Cassandra	3.11.1, 3.11.3	294	0	0	0	1
CloudStack	4.9.3, 4.11.1	297	5	0	2	0
Elasticsearch	6.0.0, 6.1.3	77	0	0	0	0
Flink	1.7.1, 1.9.1	301	0	0	0	1
Hadoop	3.0.0, 3.0.3	208	0	0	2	21
Total	-	-	5	0	4	23



(a) Precision (b) Recall

Fig. 3. The precision (a) and recall (b) of DLFinder detecting LM on the systems of RQ1 and RQ2 respectively, compared with the baseline (random prediction).

Results and discussion. Table 5 shows that new instances of duplicate logging code smells are introduced during software evolution. All the detected problematic instances (i.e., instances of IC, IE, and LM) are reported and fixed. As mentioned in Section 3 and 4, our goal of detecting DP is to show developers the logging technical debt in their systems. The number of commits for the studied time periods are: 282 commits for Cassandra, 1,097 commits for Cloud Stack, 1,036 for Elasticsearch, 485 commits for Hadoop, and 3,036 commits for Flink. These 9 instances that we detected and fixed were introduced during the studied period. For the systems that we did not find new instances of IC, IE, and LM, the number of commits is either small (e.g., 282 commits for Cassandra) or have fewer log lines (e.g., Elasticsearch has only 1.7K log lines). However, we still find new instances of DP in Cassandra and Flink. In short, we found that duplicate logging code smells are still introduced over time, and an automated approach such as DLFinder can help developers avoid duplicate logging code smells as the system evolves.

The duplicate logging code smells exist in both manually studied and additional systems. In total, DLFinder is able to detect 81 out of 91 problematic duplicate logging code smell instances (combining the results of RQ1, RQ2, and RQ3 for pattern IC, IE, and LM). We also find that new instances of logging code smells are introduced as systems evolve.

6 RQ4: What are the Relationships between Problematic Duplicate Logging Code Smells and Code Clones?

Motivation. Code clone or duplicate code is considered a bad programming practice and an indication of deeper maintenance problems [49]. Prior studies often focus on studying clones in source code and understanding their potential impact. However, there may also be other negative side effects that are related to code clones. For example, logging statements can also be copied along with other code since cloning is often performed hastily without much attention on the context [27]. In the previous RQs, we focus on studying problematic and technical debt instances of duplicate logging code smells (i.e., IC, IE, LM, and DP). In this section, we further investigate the potential causes of these instances by examining their relationship with code clones (we refer both the problematic and technical debt instances as problematic instances in this section for simplification). Our findings may provide researchers and practitioners with insights of other possible effects of code clones, other ways to further improve logging practices, and inspire future code clone studies.

Our approach of mapping code clones to problematic instances of duplicate logging code smells. Due to the large number of duplicate logging statements in the studied systems (Appendix A also studies the relationship between general duplicate logging statements and code clone), we first leverage automated clone detection tools to study whether these instances (i.e., DP and fixed instances of IC, IE, and LM) reside in cloned code. In particular, we use NiCad [28] as our clone detection tool. NiCad uses hybrid language-sensitive text comparison to detect clones. We choose NiCad because, as found in prior studies [28], [50], it has high precision (95%) and recall (96%) when detecting near-miss clones (i.e., code clones that are very similar but not exactly the same) and is actively maintained (latest release was in July 2020). Note that, we find NiCad's precision to be 96.8% in our manual verification, which is consistent with the results from prior studies (more details in Appendix A). Note that, we find NiCad's precision to be 96.8% in our manual verification, which is consistent with the results from prior studies (more details in Appendix A).

In NiCad, the source code units of comparison are determined by partitioning the source code into different granularities. The structural granularity of the source code units could be set as the method-level or block-level (e.g., the blocks of *catch*, *if*, *for*, or *method*, etc). In our study, we set the level of granularity to block-level and use the default configuration (i.e., similarity threshold is 70% and the minimum lines of a comparable code block is 10), which is suggested by prior studies indicating this configuration could achieve remarkably better results in terms of precision and recall [28], [51], [52]. Block-level provides finer-grained information, since logging statements are usually contained in code blocks for debugging or error diagnostic purposes [3]. Note that if the block is nested, the inner block is listed twice: once inside its parent block and once on its own. Hence, all blocks with lines of code above the default threshold will be compared for detecting clones. We run NiCad on the eight studied open source systems that are mentioned in Section 5. We then analyze the clone detection results and match the location of the clones with that of problematic instances. If two or more cloned code snippets contain the same set of instances, we consider the instances are related to the clone.

To reduce the effect of false negatives, we also manually study the code of *all* the remaining instances that are not identified as clones by NiCad. We manually classify the clones into the three following categories:

Clones: The code around the logging statements is more than 10 lines of code (same as the threshold of the clone detection tool). The code is exactly the same, or only with differences in identifier names (i.e., Type 1 and Type 2 clones [53]) but not detected by the clone detection tool.

Micro-clones: The code around the logging statements is very similar but is less than the minimum size of regular code clones [54]. Prior studies show that micro-clones are also important for consistent updates and they are more difficult to detect due to their small size [54], [80], [81]. However, the effect of micro-clones on code maintenance and quality is similar to regular code clones [55], [56]. Micro-clones should not be ignored when making decisions of clone management.

Non-clones: We classify other situations as non-clones.

Result of code clone analysis on problematic instances. *We find that 240 out of 289 (83%) of the problematic instances of duplicate logging code smells reside in cloned code snippets.* Table 6 presents the results of our code clone analysis. Clone (A) refers to the number of problematic instances that are detected by NiCad as in code clones. Clone (M) refers to the number of problematic instances that are manually found as in code clones. Micro. refers to the number of problematic instances that are manually found as in Micro clones (i.e., less than 10 lines of code). In general, our findings show that these problematic instances are potentially caused by code clones. **In other words, in addition to the finding from prior code clone studies, which indicates that code clones may introduce subtle program errors [57], [58], we find that code clones may also result in bad logging practices that could increase maintenance difficulties.** Future studies should further investigate the negative effect of code clones on the quality of logging statements and provide a comprehensive logging guideline.

We find that 64.2% (88/137) of the problematic instances of duplicate logging code smells that are labeled as Non-clones by the automated code clone detection tool are actually from cloned code snippets. Among them, more than half (55.7%, 49/88) reside in micro clones, which often do not get enough attention in the process of code clone management. As mentioned in the approach section of this RQ, to overcome potential false negatives, we manually study *all* the 137 problematic instances that are labeled as Non-clones by NiCad. We classify each instance that we study into three categories:

Category 1: Code clones reside in part of a large code block. Since the structural granularity level of the source code units is block-level (i.e., the minimal comparable source code unit of the tool is a block), the similarity of the code is computed by comparing blocks. However, developers may copy a small part of the code into a large code block. In such cases, the similarity would be low between two different

TABLE 6
The result of code clone analysis on problematic instances and code clones. Clone (A) shows the number of problematic duplicate logging code smell instances that are detected as clones by NiCad. Clone (M) shows the number of problematic duplicate logging code smell instances that are identified as clones by manual study. Micro. shows the number of problematic duplicate logging code smell instances that are identified as Micro clones by manual study.

	IC				IE				LM				DP			
	Clone (A)	Clone (M)	Micro.	Clone/Total	Clone (A)	Clone (M)	Micro.	Clone/Total	Clone (A)	Clone (M)	Micro.	Clone/Total	Clone (A)	Clone (M)	Micro.	Clone/Total
Cassandra	0	0	0	0/1	0	0	0	0/0	0	0	0	0/0	2	0	0	2/2
CloudStack	5	0	3	8/8	4	0	0	4/4	20	1	5	26/27	60	12	9	81/107
Elasticsearch	0	0	0	0/1	0	0	0	0/0	0	0	1	1/1	0	1	0	1/3
Flink	0	0	0	0/0	1	0	1	2/2	2	0	2	4/4	19	0	3	22/24
Hadoop	1	0	2	3/5	0	0	0	0/0	0	3	3	6/9	5	14	6	25/27
Camel	0	0	1	1/1	0	0	0	0/0	6	2	6	14/14	22	2	4	28/29
Kafka	0	0	0	0/0	0	0	0	0/0	1	0	1	2/3	3	3	2	8/14
Wicket	0	0	0	0/1	0	0	0	0/0	0	1	0	1/1	1	0	0	1/1
Total	6	0	6	12/17	5	0	1	6/6	29	7	18	54/59	112	32	24	168/207

large code blocks which only have a few lines of cloned code.

Category 2: Code clones reside in code with very similar semantics but have minor differences. The surrounding code of duplicate logging statements share highly similar semantics (i.e., implement a similar functionality), but have minor differences (e.g., additions, deletions, or partial modification on existing lines). Such scattered modifications might reduce the similarity between the code structures, and thus, result in miss detection [28], [52]. For example, there is a code block in `FTPConsumer` of Camel which does a series of operations based on the file transfer protocol (FTP). Due to the similarity between FTP and secured file transfer protocol (SFTP), Camel developers copied the code block and made modifications (e.g., change class and method names) to the all the places where SFTP is needed (e.g., `SFTPConsumer`). Therefore, clone detection tools may fail to detect this kind of cloned code blocks as due to minor yet scattered changes.

Category 3: Short methods/blocks. The logging statements reside in very short methods or code blocks with only a few lines of code. For example, there is a method in CloudStack named *verifyServicesCombination()* containing only six lines of code and duplicately locates in three different classes. The method verifies the connectivity of services, and generates a warning-level log if it fails the verification. Clone detection tool fails to detect this category of cases due to their small size compared to regular methods.

IC & IE: *30% (7/23) of the IC and IE instances in cloned code are related to micro-clones.* Since both of IC and IE reside in *catch* blocks, which usually contain only a few lines of code, we discuss these two duplicate logging code smells together. As shown in Table 6, , 7 (6 IC + 1 IE) out of 23 (17 IC + 6 IE) instances are labeled as *Micro-clones*, and 11 instances are identified as clones by the clone detection tool. The remaining five instances are labeled as *Non-clones*, since they are single logging statement thrown with multiple types of exceptions (e.g., `catch (Exception1 | Exception2 e)`). We find that all of the seven *Micro-clones* instances belong to Category 1 (i.e., short code snippets within a large code block). The reason might be that these logging statements all reside in *catch* blocks, which are usually very short. Thus, although the code in these short code blocks are identical or highly similar, they are not long enough to be considered as comparable code blocks by the clone detection tool.

LM: *25/54 (46%) of the LM instances in cloned code cannot be detected by automated clone detection tools. 92% (54/59) of LM are related code clones.* As shown in Table 6, 36 out of 59 instances are labeled as *Clones* (29 instances by tool + 7 instances by manual study), 18 out of 59 instances are labeled as *Micro-clones*, and the remaining 5 instances are labeled as *Non-clones*. For the seven instances that are identified as *Clones* by manual study, they all belong to Category 2 (i.e., they share highly similar semantics, but have minor differences). The reason might be that developers copy and paste a piece of code along with the logging statement to another location, and apply some modifications to the code. However, developers forgot to change the log message. Similarly, for the five instances that are labeled as *Non-clones*, we find that even though the code is syntactically different, the log messages do not reflect the associated method. For the 18 *Micro-clones* instances, 11 out of 18 instances belong to Category 3 (short methods), and the remaining 7 are Category 2 (short code snippets within a larger code block). As confirmed by the developers (in Section 3), these LM instances are related to logging statements being copied from other places in the code without the needed modification (e.g., updating the method name in the log).

Our manual analysis on LM instances provides insights on possible maintenance problems that are related to the modification and evolution of cloned code. Moreover, 92% of the LM instances are related to code clones. Future studies may further investigate the inconsistencies in the source code and other software artifacts (e.g., logs or comments) that are caused by code clone evolution.

DP: *81% (168/207) of the DP instances are either Clones or Micro-clones, which shows that developers may often copy code along with the logging statements across sibling classes.* In total, 144 out of 207 DP instances are labeled as *Clones* (112 by tool + 32 by manual study), 24 are labeled as *Micro-clones*, and the remaining 39 instances are *Non-clones*. For the 32 instances that are labeled as *Clones* by manual study, 16 instances are Category 1 (part of a large code block), the remaining 16 instances are Category 2 (very similar semantics with minor differences). For the 24 *Micro-clones* instances, 11 instances belong to Category 3 (short methods), and the remaining 13 are categorized as Category 1 (short code snippets within a larger code block). Combined with the results from the clone detection tool, 81% (112 detected by the tool + 32 Clones + 24 Micro-clones identified by manual study, out of 207 total instances) of the DP instances are related to code clones. One possible reason that many DP instances are related to code clone is that DP is related to inheritance. Classes that inherit from the same parent class may share certain implementation details. Nevertheless, due to the similarity of the code, developers should consider updating the log messages to distinguish the executed methods during production to assist debugging runtime errors.

For all of the remaining problematic instances (49/289)

that are not classified as clones by the automated tool and manual analysis, they mostly reside in very short code blocks (e.g., only 1∼3 lines of code). Even though these code blocks may be similar or even identical, we cannot tell whether they are clones or not. It is possible that developers implemented such similar code by coincidence, or the code was copied from other places and are then modified (but forgot to modify the log-related code).

Implication and highlights of our code clone analysis. Our finding shows that most of problematic instances of duplicate logging code smells are indeed related to code clones, and many of which cannot be easily detected by state-of-the-art clone detection tools. Our finding shows additional maintenance challenges that may be introduced by code clones – maintaining logging statements and understanding the runtime behaviour of system execution. Hence, future code clone detection studies should consider other possible side effects of code clones in addition to code maintenance and refactoring overheads. Future studies may also consider integrating different information in the software artifacts (e.g., duplicate logging statements or comments) to further improve clone detection results.

> 83% of the problematic instances of duplicate logging code smells (240 out of 289 instances, combining the results of tool detection and manual study) are related to code clones. Our finding further shows the potential negative effect of code clones on system maintenance. Moreover, 17% of the instances reside in short code blocks, which might be difficult to detect by using existing code clone detection tools.

Discussion: the potential of using code clone detection tool to assist in finding problematic instances of duplicate logging code smells. In the previous section, we found that most of the problematic instances of duplicate logging code smells (83%) are related to code clones. Therefore, we use the results of our code clone analysis to compare with and/or assist our detection approach of LM. We focus on studying LM for two reasons. First, we found that 92% (54/59) of the LM instances are related to code clones. Second, unlike other patterns that have a detection accuracy of 100%, our current detection approach for LM analyzes textual similarity of the logging statement and its surrounding code, which has a lower precision and recall. Using clone detection results may further help improve our detection accuracy.

We first use the clone detection result as a baseline and compare the result with the detection approach implemented in DLFinder. If two duplicate logging statements reside in cloned code, we consider them as a possible instance of LM. Overall, the average precision and recall of using clone detection result are 3.7% and 53.7%, respectively, in the studied systems in RQ1. The average precision and recall in the additional systems are 1.5% and 38.9%, respectively, in the studied systems in RQ2. Compared to using clone detection result as a baseline, our approach has a better precision and recall (around 12% in precision and 80% in recall). However, among the 10 LM instances that cannot be detected using our approach, four of them are detected by this baseline approach. After manual investigation, we found that, in these four instances, the log message describes a local code block while the class-method name describes the functionality of the entire method. Hence, in such cases, using clone detection results may be more effective in detecting LM.

Inspired by the analysis result, we then study if clone detection result can assist DLFinder in finding LM. We use the automated clone detection results from NiCad to filter the LM instances that are detected by DLFinder. Namely, DLFinder only reports that a set of duplicate logging statements is a potential LM instance if they reside in cloned code. We find that, after using clone detection results to filter out potential false positives, the average precision and recall for the eight studied systems are 17.7% and 42.4%, respectively. Compared to DLFinder's detection result (Table 4), the precision increases by around 5% but the recall decreases by around 40%. The reason may be that many problematic LM instances reside in code clones that are difficult to detect by clone detection tool (e.g., micro clones). As shown in Table 6, NiCad only detects 29/54 of the LM instances that reside in cloned code. As we discussed in Section 5, we believe that recall is more important when detecting LM, since we found the manual effort of evaluating LM instances to be small (i.e. within a few minutes). However, our findings also shed some light on balancing the precision and recall of detecting duplicate logging code smells. Future studies may consider further improving code clone detection techniques to detect code smells that are related to logging statements.

7 RQ5: What are the Relationships Between Duplicate Logging Statements and Code Clones?

Motivation. In Section 6, we investigate the relationship between code clones and *problematic instances of duplicate logging code smells*. As discussed in Section 3, duplicate logging code smells are duplicate logging statements with specific patterns that may be indications of logging problems. In this section, we further investigate the relationship between *duplicate logging statements* and code clones. We also study the potential impact of duplicate logging statements on detecting code clones.

Approach. Similar to Section 6, we use both an automated and a manual approach to study the relationship between code clones and duplicate logging statements. We first leverage NiCad to automatically detect clones. Although we found that NiCad has a great precision (i.e., 96.8%, as shown in Appendix A), there may still exist false negatives (i.e., the duplicate logging statements are code clones, but are missed by the tool). Therefore, we manually investigate a statistical sample of duplicate logging statements, which reside in code snippets that are classified by NiCad as *Non-clones* to study the false negative rate.

Results of automated code clone analysis on duplicate logging statements. *We find that a considerable number of duplicate logging statements (43.7% on average) reside in cloned code snippets.* Table 7 presents the results of our code clone analysis. DupSet refers to the total sets of duplicate logging statements (a set contains two or more logging statements with the same text message). CloneSet refers to the subset of duplicate logging statement sets (DupSet) that are from

cloned code snippets. The percentage number is the proportion of CloneSet out of DupSet. Finally, Avg. Sim. refers to the average code clone similarity score among the cloned code snippets. As shown in Table 7, 11.5% to 51.1% sets of duplicate logging statements are from the cloned code snippets in the studied systems. Overall, 1,042 out of 2,382 (43.7%) sets of duplicate logging statements are related to code clones (with an average 80% similarity score).

Our finding shows that a considerable number of duplicate logging statements are related to code clones, and developers may not change the log messages when they copy a piece of code to another location. However, due to the importance of logging for understanding system runtime behaviour [1], [30], [82], developers should avoid directly copying logging statements. Developers should consider modifying the log messages (e.g., to include the class name, modify the message to reflect code changes, or record new important dynamic variables) to assist debugging and workload understanding.

Results of manual code clone analysis on duplicate logging statements. *We find that more than 50% of the sampled duplicate logging statements reside in cloned code snippets that are difficult to detect using automated code clone detection tools. In particular, 24.5% of the manually studied duplicate logging statements are related to code clones, and 26.2% are related to micro-clones.* In total, we randomly sample 298 sets of duplicate logging statements to achieve a confidence of 95% and a confidence interval of 5%. For each set of the sampled duplicate logging statements, we manually classify them into three types: *Clone* (i.e., but not detected by code clone detection tools), *Micro-clone* (i.e., code blocks with less than 10 lines of code), and *Non-clone*.

Table 8 presents the results of our manual study. Overall, 73 out of the 298 (24.5%) manually-studied sets of duplicate logging statements are labeled as *Clones*. 78 out of 298 (26.2%) sets are labeled as *Micro-clones*. The remaining 147 out of 298 (49.3%) sets are labeled as *Non-clones*. For 42 out of the 73 cases of *Clones*, and 32 out of 78 cases of *Micro-clones*, we find that developers often only copy and paste part of the code into another large code block (Category 1 discussed in Section 6). Hence, only small parts of large code blocks are similar, which reduces the similarity score. For 53 out of the 73 cases that are manually identified as *Clones*, they reside in code with very similar semantics but have minor differences (Category 2). Note that some cases belong to multiple categories. For 46 out of 78 cases they are classified as *Micro-clones*, which reside in very short methods with only a few lines of code (Category 3).

In summary, we find that more than half of the duplicate logging statements reside in cloned code snippets. Our manual study also highlights that many duplicate logging statements reside in cloned code that may be difficult to detect by clone detection tools.

Discussion: The Potential Impact of Duplicate Logging Statements on Detecting Code Clones. In this RQ, we find that a noticeable number of duplicate logging statements reside in cloned code snippets. We further investigate the impact of duplicate logging statements on the detection of code clones, namely, whether considering duplicate logging statements helps detect code clones. Specifically, for each set of CloneSet presented in Table 7, we first remove the duplicate logging statements from the related code snippets. We then re-examine how many code snippets related to prior DupSet are still identified as cloned code snippets and how many are not, by using NiCad.

TABLE 7
Automated code clone analysis results on duplicate logging statements. DupSet: Total sets of duplicate logging statements. CloneSet: Sets of duplicate logging statements that are from cloned code snippets. Avg. Sim.: Average similarity of the cloned code snippets.

	DupSet	CloneSet	Avg. Sim.
Cassandra	46	14 (30.4%)	79.7
CloudStack	865	442 (51.1%)	80.3
Elasticsearch	40	17 (42.5%)	72.2
Flink	203	92 (45.3%)	78.8
Hadoop	217	25 (11.5%)	76
Camel	886	421 (47.5%)	80.7
Kafka	104	23 (22.1%)	75.4
Wicket	21	8 (38.1%)	83.1
Overall	2,382	1,042 (43.7%)	80.0

TABLE 8
Manual study results on the recall of clone detection tool on duplicate logging statements. Both the Clones and Micro-clones are labeled manually and they are not detected by the clone detection tool.

	Clones	Micro-clones	Non-clones	Total
Cassandra	1	3	3	7
CloudStack	22	26	46	94
Elasticsearch	1	1	3	5
Flink	5	4	16	25
Hadoop	12	6	25	43
Camel	28	30	45	103
Kafka	3	7	8	18
Wicket	1	1	1	3
Total	73	78	147	298

Table 9 shows the results of our experiments on investigating the impact of duplicate logging statements on detecting code clones. CloneSet refers to the sets of cloned code snippets with duplicate logging statements. CloneSet-NDL refers to the sets of cloned code snippets after removing the related duplicate logging statements. CloneSet-Reduced represents the number of sets reduced by comparing CloneSet-DL with CloneSet-NDL. Per. Reduced shows the percentage of CloneSet-Reduced given CloneSet-DL. On average, 28.6% CloneSet are not detected by NiCad as cloned code snippets after removing duplicate logging statements. Specifically for each studied system, the reduction ranges from 25.0% in Wicket to around a 47.1% in Elasticsearch.

We then manually investigate the code snippets that are not detected as cloned code snippets after removing duplicate logging statements (i.e., CloneSet-Reduced). We find two potential reasons that the clone detection tool could not detect them as cloned code snippets. 1) *Reduced total lines of similar code after removing duplicate logging statements:* The

TABLE 9
The results of investigating the impact of duplicate logging statements on detecting code clones.

	CloneSet	CloneSet-NDL	CloneSet-Reduced	Per. Reduced
Cassandra	14	10	4	28.6%
CloudStack	442	329	113	25.6%
Elasticsearch	17	9	8	47.1%
Flink	92	64	28	30.4%
Hadoop	25	16	9	36.0%
Camel	421	299	122	29.0%
Kafka	23	13	10	43.5%
Wicket	8	6	2	25.0%
Total	1042	744	298	28.6%

logging statements usually span across one to three, and sometimes even more, lines of code. However, these lines of code in the duplicate logging statements are the main part of the clones. After removing the duplicate logging statements, the total number of similar lines of code snippets is too small for a clone detection tool to consider as clones. 2) *Reduced similarity after removing duplicate logging statements:* Duplicate logging statements have exactly the same log message and are represented as Method Invocation nodes in the Abstract Syntax Tree. Removing duplicate logging statements will decrease the similarity of code snippets, both syntactically and semantically. Hence, the similarity might become smaller than the threshold of the clone detection tool and the code snippets are not detected as clones.

In summary, we find that a large portion of the cloned code snippets with duplicate logging statements (from 25.0% to 47.1%) are not detected as cloned code snippets after removing the duplicate logging statements. The results show that duplicate logging statements have a non-negligible impact on the detection of code clones. Future code clone studies may consider the effect of logging code in order to further improve the code clone detection techniques.

> More than half of the duplicate logging statements reside in cloned code snippets, and a large portion of them reside in short code blocks which are difficult to detect using existing code clone detection tools. We also find that duplicate logging statements have a non-negligible impact on helping the detection of code clones. Future works may leverage duplicate logging statements to further improve code clone detection tools.

8 Threats to Validity

Construct validity. In this paper, we study duplicate logging statements from a static point of view. There may be other types of unclear log messages that are dynamically generated during system runtime. Using such dynamic information can also be helpful in identifying unclear log messages. However, the generated log messages are highly dependent on the executed workloads (i.e., hard to achieve a high recall). DLFinder statically identifies and improves duplicate logging statements, is useful as it does not require any run-time information. Future studies may consider studying runtime-generated logs and further improve logging practices. We detect duplicate logging code smells by analyzing the surrounding code of logging statements as their context. Apart from that, the sequence of generated logs may also provide context information (e.g., the relationship among preceding logs and subsequent logs). However, for most of the duplicate logging code smells discussed in this paper, they are not directly related to the log sequences (e.g., the patterns of IC and IE are related to the logging statements and their surrounding catch blocks). Even though analyzing the generated log sequences may provide more information, the duplicate logging code smells can still cause challenges and increase maintenance costs, as acknowledged by the developers in the studied systems. Future study may consider the execution path of logging statements as the context information to further improve logging practice.

Internal validity. We conducted manual studies to uncover the patterns of duplicate logging code smells, study their potential impact and examine duplicate logging statements that are not classified by the automated clone detection tool as clones. Involving external logging experts may uncover more patterns of logging statements or have different manual study results. To mitigate the biases, two of the authors examine the data independently. For most of the cases the two authors reach an agreement. Any disagreement is discussed until a consensus is reached. In order to reduce the subjective bias from the authors, we have contacted the developers to confirm the uncovered patterns and their impact. When detecting LM instances, using different approaches to split the text into words may have different results. We follow common text pre-processing techniques to split the text by space and camel case [41]. We define duplicate logging statements as two or more logging statements that have the same static text message. We were able to uncover five patterns of duplicate logging code smells and detect many duplicate logging code smell instances. However, logging statements with non-identical but similar static texts may also cause problems to developers (e.g., when analyzing dynamically generated logs). Future studies should consider different types of duplicate logging statements (e.g., logs with similar text messages). We remove the top 50 most frequent words when detecting LM, because there is a considerable number of generic words across different log messages. However, this might also introduce false negatives. Future studies may consider applying more advanced techniques to better detect the instances of LM. There is a considerable number of code clone detection tools proposed by prior studies [28], [59]–[63]. We use NiCad [28] to detect code clone, as it has high precision (95%), recall (96%) and outperforms the state-of-the-art code clone detection tools [28], [50], [52] when detecting near-miss clones, and is actively maintained (latest release was in July 2020). We also manually examine the precision of Nicad in Appendix A, where we find its precision to be 96.8% in our manual verification, which is consistent with the results from prior studies [50], [52].

External validity. We conducted our study on five large-scale open source systems in different domains. We found that our uncovered patterns and the corresponding problematic and justifiable cases are common among the studied systems. However, our finding may not be generalizable to other systems. Hence, we studied whether the uncovered patterns exist in three other systems. We found that the patterns of duplicate logging code smells also exist in these systems and we did not find any new duplicate logging code smell patterns in our manual verification. Our studied systems are all implemented in Java, so the results may not be generalizable to systems in other programming languages. Future studies should validate the generalizability of our findings in systems in other programming languages.

9 Related Work

Empirical studies on logging practices. There are several studies on characterizing the logging practices in software systems [3], [30], [64]. Yuan et al. [30] conducted a quantitative characteristics study on log messages for large-scale

open source C/C++ systems. Chen et al. [64] replicated the study by Yuan et al. [30] on Java open-source projects. Both of their studies found that log message is crucial for system understanding and maintenance. Fu et al. [3] studied where developers in Microsoft add logging statements in the code and summarized several typical logging strategies. They found that developers often add logs to check the returned value of a method. Different from prior studies, in this paper, we focus on manually understanding duplicate logging code smells. We also discuss potential approaches to detect and fix these code smells based on different contexts (i.e., surrounding code).

Improving logging practices. Zhao et al. [17] proposed a tool that determines how to optimally place logging statements given a performance overhead threshold. Zhu et al. [16] provided a tool for suggesting log placement using machine learning techniques. Yuan et al. [1] proposed an approach that can automatically insert additional variables into logging statements to enhance the error diagnostic information. Chen et al. [19] concluded five categories of logging anti-patterns from code changes, and implemented a tool to detect the anti-patterns. Hassani et al. [20] identified seven root-causes of the log-related issues from log-related bug reports. Compared to prior studies, we study logging code smells that may be caused by duplicate logs, with a goal to help developers improve logging code. The logging problems that we uncovered in this study are not discovered by prior work. We conducted an extensive manual study through obtaining a deep understanding on not only the logging statements but also the surrounding code, whereas prior studies usually only look at the problems that are related to the logging statement itself.

Code smells and code clones. Code smells can be indications of bad design and implementation choices, which may affect software systems' maintainability [65]–[68], understandability [69], [70], and performance [71]. To mitigate the impact of code smells, studies have been proposed to detect code smells [72]–[76]. Duplicate code (or code clones) is a kind of code smells which may be caused by developers copying and pasting a piece of code from one place to another [27], [58]. Such code clones may indicate quality problems. There are many studies that focus on studying the impact of code clones [77]–[79], and detecting them [28], [59], [60]. In this paper, we study duplicate logging code smells, which are not studied in prior duplicate code studies. We also investigate the relationship between duplicate logging statements and code clones. Some instances of the problematic duplicate logging code smells in our study might also be related to micro-clones (i.e., cloned code snippets that are smaller than the minimum size of the regular clones [54]). A small number of prior studies investigate the characteristics and impact of micro-clones in evolving software systems [54]–[56], [80], [81]. Specifically, micro-clones may have similar tendencies of replicating severe bugs as regular clones [55], [56]. However, the potential impact of micro-clones on logging code are not studied in these works. Our study provides insights for future studies on the relationship between micro-clones and logging code. The investigation on duplicate logging code smells and duplicate logging statements may also help identify micro-clones and further alleviate the impact of micro-clones on software maintenance and evolution.

10 Conclusion

Duplicate logging statements may affect developers' understanding of the system execution. In this paper, we study over 4K duplicate logging statements in five large-scale open source systems (Hadoop, CloudStack, Elasticsearch, Cassandra and Flink). We uncover five patterns of duplicate logging code smells. Further, we assess the impact of each uncovered code smell and find not all are problematic and need fixes. In particular, we find six justifiable cases where the uncovered patterns of duplicate logging code smells may not be problematic. We received confirmation from developers on both the problematic and justifiable cases. Combining our manual analysis and developers' feedback, we developed a static analysis tool, DLFinder, which automatically detects problematic duplicate logging code smells. We applied DLFinder on the five manually studied systems and three additional systems. In total, we reported 91 problematic duplicate logging code smell instances in the eight studied systems to developers and all of them are fixed. DLFinder successfully detects 81 out of the 91 instances. We further investigate the relationship between duplicate logging statements and code clones, in order to provide a more comprehensive understanding of duplicate logging statements and duplicate logging code smells. We find that most of the problematic instances of duplicate logging code smells and almost half of the duplicate logging statements reside in cloned code snippets. Among them, a large portion reside in very short code blocks which might be difficult to detect using existing code clone detection tools.

Our study highlights the importance of the context of the logging code, i.e., the nature of logging code is highly associated with both the structure and the functionality of the surrounding code. Future studies should consider the code context when providing guidance to logging practices, more advanced logging libraries are needed to help developers improve logging practice and to avoid logging code smells. Our findings also provide an initial evidence on the prevalence of duplicate logging statements that reside in cloned code snippets, and the potential impact of code clones on logging practices. Future studies may also consider integrating different information in the software artifacts (e.g., duplicate logging statements) to further improve clone detection results.

References

[1] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS '11, 2011, pp. 3–14.

[2] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: Error diagnosis by connecting clues from run-time logs," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '10, 2010, pp. 143–154.

[3] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do developers log? an empirical study on logging practices in industry," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE-SEIP '14, 2014, pp. 24–33.

[4] T.-H. Chen, M. D. Syer, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Analytics-driven load testing: An industrial experience report on load testing of large-scale systems," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE-SEIP '17, 2017, pp. 243–252.

[5] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automatic identification of load testing problems," in *Proceedings of 24th International Conference on Software Maintenance*, ser. ICSM '08, 2008, pp. 307–316.

[6] B. Chen, J. Song, P. Xu, X. Hu, and Z. M. J. Jiang, "An automated approach to estimating code coverage measures via execution logs," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18', 2018, pp. 305–316.

[7] A. E. Hassan, D. J. Martin, P. Flora, P. Mansfield, and D. Dietz, "An Industrial Case Study of Customizing Operational Profiles Using Log Compression," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08, 2008, pp. 713–723.

[8] W. Shang, M. Nagappan, A. E. Hassan, and Z. M. Jiang, "Understanding log lines using development knowledge," in *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME '14, 2014, pp. 21–30.

[9] N. Busany and S. Maoz, "Behavioral log analysis with statistical guarantees," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16, 2016, pp. 877–887.

[10] H. Barringer, A. Groce, K. Havelund, and M. H. Smith, "Formal analysis of log files," *JACIC*, vol. 7, no. 11, pp. 365–390, 2010.

[11] T.-H. Chen, W. Shang, A. E. Hassan, M. Nasser, and P. Flora, "Cacheoptimizer: Helping developers configure caching frameworks for hibernate-based database-centric web applications," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, 2016, pp. 666–677.

[12] K. Yao, G. B. d. Pádua, W. Shang, S. Sporea, A. Toma, and S. Sajedi, "Log4perf: Suggesting logging locations for web-based systems' performance monitoring," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18, 2018, pp. 21–30.

[13] "Log4j," http://logging.apache.org/log4j/2.x/.

[14] S. Kabinna, C.-P. Bezemer, W. Shang, and A. E. Hassan, "Logging library migrations: A case study for the apache software foundation projects," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16, 2016, pp. 154–164.

[15] A. Pecchia, M. Cinque, G. Carrozza, and D. Cotroneo, "Industry practices and event logging: Assessment of a critical software development process," in *Proceedings of th 37th International Conference on Software Engineering*, ser. ICSE '15, 2015, pp. 169–178.

[16] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE '15, 2015, pp. 415–425.

[17] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou, "Log20: Fully automated optimal placement of log printing statements under specified overhead threshold," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17, 2017, pp. 565–581.

[18] H. Pinjia, Z. Chen, S. He, and M. R. Lyu, "Characterizing the natural language descriptions in software logging statements," in *Proceedings of the 33rd IEEE international conference on Automated software engineering*, 2018, pp. 1–11.

[19] B. Chen and Z. M. J. Jiang, "Characterizing and detecting anti-patterns in the logging code," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17, 2017, pp. 71–81.

[20] M. Hassani, W. Shang, E. Shihab, and N. Tsantalis, "Studying and detecting log-related issues," *Empirical Software Engineering*, 2018.

[21] B. Russo, G. Succi, and W. Pedrycz, "Mining system logs to learn error predictors: a case study of a telemetry system," *Empirical Software Engineering*, vol. 20, no. 4, pp. 879–927, 2015.

[22] A. J. Oliner, A. Ganapathi, and W. Xu, "Advances and challenges in log analysis," *Commun. ACM*, vol. 55, no. 2, pp. 55–61, 2012.

[23] I. Fronza, A. Sillitti, G. Succi, M. Terho, and J. Vlasenko, "Failure prediction based on log files using random indexing and support vector machines," *Journal of Systems and Software*, vol. 86, no. 1, pp. 2–11, 2013.

[24] M. Pettinato, J. P. Gil, P. Galeas, and B. Russo, "Log mining to re-construct system behavior: An exploratory study on a large telescope system," *Information & Software Technology*, vol. 114, pp. 121–136, 2019.

[25] D. Budgen, *Software Design*. Addison-Wesley, 2003.

[26] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*, ser. Addison-Wesley object technology series, 1999.

[27] F. Rahman, C. Bird, and P. Devanbu, "Clones: What is that smell?" in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, May 2010, pp. 72–81.

[28] C. K. Roy and J. R. Cordy, "NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *The 16th IEEE International Conference on Program Comprehension*, ser. ICPC '08, 2008, pp. 172–181.

[29] Z. Li, T. P. Chen, J. Yang, and W. Shang, "DLFinder: Characterizing and detecting duplicate logging code smells," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*, 2019, pp. 152–163.

[30] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *ICSE 2012: Proceedings of the 2012 International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2012, pp. 102–112.

[31] H. Li, W. Shang, and A. E. Hassan, "Which log level should developers choose for a new logging statement?" *Empirical Software Engineering*, vol. 22, no. 4, pp. 1684–1716, Aug 2017.

[32] H. Li, T.-H. P. Chen, W. Shang, and A. E. Hassan, "Studying software logging using topic models," *Empirical Software Engineering*, vol. 23, pp. 2655—-2694, Jan 2018.

[33] "Simple logging facade for Java (SLF4J)," http://www.slf4j.org, last checked Feb. 2018.

[34] S. Boslaugh and P. Watters, *Statistics in a Nutshell: A Desktop Quick Reference*, ser. In a Nutshell (O'Reilly). O'Reilly Media, 2008.

[35] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia Medica*, vol. 22, no. 3, pp. 276–282, 2012.

[36] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14, 2014, pp. 249–265.

[37] "Changes to JobHistory makes it backward incompatible," https://issues.apache.org/jira/browse/HADOOP-4190, last checked April 4th 2018.

[38] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Softw.*, vol. 29, no. 6, pp. 18–21, 2012.

[39] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, 2013, pp. 672–681.

[40] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '16, 2016, pp. 858–870.

[41] T.-H. Chen, S. W. Thomas, and A. E. Hassan, "A survey on the use of topic models when mining software repositories," *Empirical Software Engineering*, vol. 21, no. 5, pp. 1843–1919, 2016.

[42] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[43] J. Yang and L. Tan, "SWordNet: Inferring semantically related words from software context," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1856–1886, 2014.

[44] X. Xia, E. Shihab, Y. Kamei, D. Lo, and X. Wang, "Predicting crashing releases of mobile applications," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2016, pp. 1–10.

[45] X. Xia, D. Lo, E. Shihab, X. Wang, and X. Yang, "Elblocker: Predicting blocking bugs with ensemble imbalance learning," *Information & Software Technology*, vol. 61, pp. 93–106, 2015.

[46] H. Valdivia Garcia and E. Shihab, "Characterizing and predicting blocking bugs in open source projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014, 2014, pp. 72–81.

[47] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, 2007, pp. 57–76.

[48] T.-H. Chen, S. Weiyi, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Detecting performance anti-patterns for applications developed using object-relational mapping," in *Proceedings of the*

36th International Conference on Software Engineering (ICSE), 2014, pp. 1001–1012.

[49] M. Fowler, *Refactoring - Improving the Design of Existing Code*, ser. Addison Wesley object technology series. Addison-Wesley, 1999.

[50] C. K. Roy and J. R. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," in *Proceedings of the 2nd International Conference on Software Testing Verification and Validation, ICST 2009*, 2009, pp. 157–166.

[51] J. Svajlenko and C. K. Roy, "Evaluating modern clone detection tools," in *Proceedings of the 30th International Conference on Software Maintenance and Evolution*, 2014, pp. 321–330.

[52] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470 – 495, 2009.

[53] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Trans. Software Eng.*, vol. 33, no. 9, pp. 577–591, 2007.

[54] M. Mondal, C. K. Roy, and K. A. Schneider, "Micro-clones in evolving software," in *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*, 2018, pp. 50–60.

[55] J. F. Islam, M. Mondal, C. K. Roy, and K. A. Schneider, "Comparing bug replication in regular and micro code clones," in *Proceedings of the 27th International Conference on Program Comprehension*, ser. ICPC '19, 2019, pp. 81–92.

[56] J. F. Islam, M. Mondal, and C. K. Roy, "A comparative study of software bugs in micro-clones and regular code clones," in *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019*, 2019, pp. 73–83.

[57] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2007, pp. 55–64.

[58] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: a review of current knowledge," *Journal of Software Maintenance*, vol. 23, no. 3, pp. 179–202, 2011.

[59] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[60] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Trans. Software Eng.*, vol. 32, no. 3, pp. 176–192, 2006.

[61] S. Livieri, Y. Higo, M. Matushita, and K. Inoue, "Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder," in *Proc. of the 29th Int. conference on Software Engineering*, 2007.

[62] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01*, 2001, pp. 301–309.

[63] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *30th International Conference on Software Engineering (ICSE 2008)*, 2008, pp. 321–330.

[64] B. Chen and Z. M. (Jack) Jiang, "Characterizing logging practices in java-based open source software projects – a replication study in apache software foundation," *Empirical Software Engineering*, vol. 22, no. 1, pp. 330–374, Feb 2017.

[65] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. D. Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, May 2015, pp. 403–414.

[66] I. Ahmed, C. Brindescu, U. A. Mannan, C. Jensen, and A. Sarma, "An empirical examination of the relationship between code smells and merge conflicts," in *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '17, 2017, pp. 58–67.

[67] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, Aug 2013.

[68] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen, "Understanding code smells in android applications," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, 2016, pp. 225–234.

[69] C. Chapman, P. Wang, and K. T. Stolee, "Exploring regular expression comprehension," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct 2017, pp. 405–416.

[70] S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "The effect of lexicon bad smells on concept location in source code," in *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, Sept 2011, pp. 125–134.

[71] X. Xiao, S. Han, C. Zhang, and D. Zhang, "Uncovering javascript performance code smells relevant to type mutations," in *Programming Languages and Systems*, X. Feng and S. Park, Eds., 2015, pp. 335–355.

[72] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2013, pp. 268–278.

[73] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Detection of embedded code smells in dynamic web applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012, 2012, pp. 282–285.

[74] C. Parnin, C. Görg, and O. Nnadi, "A catalogue of lightweight visualizations to support code smell inspection," in *Proceedings of the 4th ACM Symposium on Software Visualization*, ser. SoftVis '08, 2008, pp. 77–86.

[75] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw, "Building empirical support for automated code smell detection," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '10, 2010, pp. 8:1–8:10.

[76] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting and refactoring code smells in spreadsheet formulas," *Empirical Software Engineering*, vol. 20, no. 2, pp. 549–575, 2015.

[77] E. Jürgens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Proceedings of the 31st International Conference on Software Engineering, ICSE 2009*, 2009, pp. 485–495.

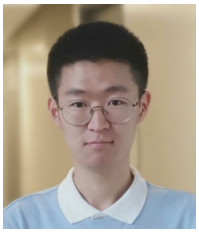
[78] C. Kapser and M. W. Godfrey, "Cloning considered harmful," *Reverse Engineering, Working Conference on*, vol. 0, pp. 19–28, 2006.

[79] N. Göde and R. Koschke, "Frequency and risks of changes to clones," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011*, 2011, pp. 311–320.

[80] M. Mondal, B. Roy, C. K. Roy, and K. A. Schneider, "Investigating near-miss micro-clones in evolving software," in *ICPC '20: 28th International Conference on Program Comprehension*. ACM, 2020, pp. 208–218.

[81] ——, "Ranking co-change candidates of micro-clones," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, CASCON 2019*, T. Pakfetrat, G. Jourdan, K. Kontogiannis, and R. F. Enenkel, Eds., pp. 244–253.

[82] T. Chen, M. D. Syer, W. Shang, Z. M. Jiang, A. E. Hassan, M. N. Nasser, and P. Flora, "Analytics-driven load testing: An industrial experience report on load testing of large-scale systems," in *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering, ICSE-SEIP*, 2017, pp. 243–252.



Zhenhao Li Zhenhao Li is a Ph.D student at the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. He obtained his M.ASc degree from Concordia University and B.Eng. from Harbin Institute of Technology. His work has been published at renowned venues such as ICSE and ASE. His research interests include software log analysis, improving logging practices, program analysis, and mining software repositories.

Tse-Hsun (Peter) Chen Tse-Hsun (Peter) Chen is an Assistant Professor in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. He leads the Software PErformance, Analysis, and Reliability (SPEAR) Lab, which focuses on conducting research on performance engineering, program analysis, log analysis, production debugging, and mining software repositories. His work has been published in flagship conferences and journals such as ICSE, FSE, TSE, EMSE, and MSR. He serves regularly as a program committee member of international conferences in the field of software engineering, such as ASE, ICSME, SANER, and ICPC, and he is a regular reviewer for software engineering journals such as JSS, EMSE, and TSE. Dr. Chen obtained his BSc from the University of British Columbia, and MSc and PhD from Queen's University. Besides his academic career, Dr. Chen also worked as a software performance engineer at BlackBerry for over four years. Early tools developed by Dr. Chen were integrated into industrial practice for ensuring the quality of large-scale enterprise systems. More information at: http://petertsehsun.github.io/.

Jinqiu Yang Jinqiu Yang is an Assistant Professor in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. Her research interests include automated program repair, software testing, software text analytics, and mining software repositories. Her work has been published flagship conferences and journals such as ICSE, FSE, EMSE. She serves regularly as a program committee member of international conferences in Software Engineering, such as ASE, ICSE, ICSME and SANER. She is a regular reviewer for Software Engineering journals such as EMSE and JSS. Dr. Yang obtained her BEng from Nanjing University, and MSc and PhD from University of Waterloo. More information at: https://jinqiuyang.github.io/.

Weiyi Shang Weiyi Shang is an Assistant Professor and Concordia University Research Chair in Ultra-large-scale Systems at the Department of Computer Science and Software Engineering at Concordia University, Montreal. He has received his Ph.D. and M.Sc. degrees from Queens University (Canada) and he obtained B.Eng. from Harbin Institute of Technology. His research interests include big data software engineering, software engineering for ultra-largescale systems, software log mining, empirical software engineering, and software performance engineering. His work has been published at premier venues such as ICSE, FSE, ASE, ICSME, MSR and WCRE, as well as in major journals such as TSE, EMSE, JSS, JSEP and SCP. His work has won premium awards, such as SIGSOFT Distinguished paper award at ICSE 2013 and best paper award at WCRE 2011. His industrial experience includes helping improve the quality and performance of ultra-large-scale systems in BlackBerry. Early tools and techniques developed by him are already integrated into products used by millions of users worldwide. Contact him at shang@encs.concordia.ca.

APPENDIX A
PRECISION OF NICAD ON DETECTING DUPLICATE LOGGING STATEMENTS THAT RESIDE IN CLONED CODE

We rely on NiCad for automated clone detection. To examine the false positives of NiCad, we then manually verify a randomly sampled set of duplicate logging statements (281 sets in total, with 95% confidence level and 5% confidence interval) that are classified as clones by NiCad. For each set of the sampled duplicate logging statements, we manually go through the logging statements and their surrounding code to verify whether they are clones or not. Overall, we find that 272 out of the 281 sampled sets (96.8%) are clones, which is similar to the performance of NiCad that is reported in prior studies. For the 9 false positives, 3 of them are duplicate logging statements located in different branches of a nested method (i.e., developers define a method within a method). In such cases, NiCad would analyze the code block twice. For example, in ElasticSearch [2], two duplicate logging statements with the same static text message *"Failed to execute NodeStatsAction for ClusterInfoUpdateJob"* are located in different branches of the same nested method *onFailure(Exception e)*. However, since the method *onFailure(Exception e)* is defined in the method *refresh()*, NiCad would analyze the same code block twice and detect them as clones. For the remaining 6 out of 9 false positives, we could not identify the reasons that they are classified as clones, since the code snippets look neither structurally nor semantically similar.

2. https://github.com/elastic/elasticsearch/blob/70b8d7bc64f165735502de9d8c5fa673fa21e02b/server/src/main/java/org/elasticsearch/cluster/InternalClusterInfoService.java