A Survey of Software Log Instrumentation

BOYUAN CHEN and ZHEN MING (JACK) JIANG, York University, Canada

Log messages have been used widely in many software systems for a variety of purposes during software development and field operation. There are two phases in software logging: log instrumentation and log management. Log instrumentation refers to the practice that developers insert logging code into source code to record runtime information. Log management refers to the practice that operators collect the generated log messages and conduct data analysis techniques to provide valuable insights of runtime behavior. There are many open source and commercial log management tools available. However, their effectiveness highly depends on the quality of the instrumented logging code, as log messages generated by high-quality logging code can greatly ease the process of various log analysis tasks (e.g., monitoring, failure diagnosis, and auditing). Hence, in this article, we conducted a systematic survey on state-of-the-art research on log instrumentation by studying 69 papers between 1997 and 2019. In particular, we have focused on the challenges and proposed solutions used in the three steps of log instrumentation: (1) logging approach; (2) logging utility integration; and (3) logging code composition. This survey will be useful to DevOps practitioners and researchers who are interested in software logging.

CCS Concepts: • **Software and its engineering → Software creation and management**;

Additional Key Words and Phrases: Systematic survey, software logging, instrumentation

ACM Reference format:
Boyuan Chen and Zhen Ming (Jack) Jiang. 2021. A Survey of Software Log Instrumentation. *ACM Comput. Surv.* 54, 4, Article 90 (April 2021), 34 pages.
https://doi.org/10.1145/3448976

1 INTRODUCTION

Software logging is a common programming practice that developers use to track and record the runtime behavior of software systems. Software logging has been used extensively for monitoring [107, 117], failure diagnosis [55, 127], performance analysis [54, 75, 99, 115], test analysis [45, 70], security and legal compliance [22, 98, 103], and business analytics [31, 96].

As shown in Figure 1, software logging [48] consists of two phases: (1) **Log Instrumentation**, and (2) **Log Management**. The log instrumentation phase, which concerns about the development and maintenance of the logging code, consists of three steps: *Logging Approach*, ***Logging Utility (LU) Integration***, and **Logging Code (LC)** *Composition*. The log management phase, which focuses on processing the generated log messages once the system deploys, consists of three steps: *Log Generation*, *Log Collection*, and *Log Analysis*.

Authors' address: B. Chen and Z. M. (Jack) Jiang, York University, 4700 Keele St, Toronto, ON, Canada M3J 1P3; emails: {chenfsd, zmjiang}@eecs.yorku.ca.

0360-0300/2021/04-ART90 $15.00
https://doi.org/10.1145/3448976

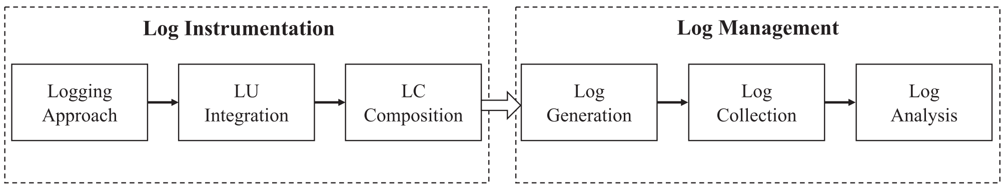


Fig. 1. The overall process of software logging.

There are many industrial-strength tools available already to aid effective log management. For example, the ELK stack (Elasticsearch [6], Logstash [15], and Kibana [11]) is a very popular platform for collecting, searching, and analyzing logs. Splunk [21], one of the most popular commercial log management platforms, provides an integrated solution for monitoring and managing complex telemetry data. According to Gartner, the market for log management tools is estimated to be a $1.5B market and has been growing rapidly every year [7].

However, the effectiveness of log management highly depends on the quality of the logging code produced from the log instrumentation phase. Low-quality logging code can cause issues in problem diagnosis [129], high maintenance efforts [41, 78, 95], performance slow-down [54], or even system crashes [9]. On one hand, there are many LUs available for developers to instrument their logging code. For example, Chen and Jiang [44] found that there are more than 800 open sourced third-party LUs being used by various Java-based GitHub projects. Among these LUs, `SLF4J` and `Android Logging` are two of the most popular LUs. On the other hand, unlike other aspects in the software development process (e.g., software design [61] and code refactoring [59]), recent empirical studies show that there are no well-established logging practices in practice [31, 60, 109]. The process of log instrumentation is generally ad hoc and usually relying on developers' common sense. As more systems are migrating to the cloud [36] with increasing layers of complexity [136], new software development paradigms like **Observability-Driven Development (ODD)** [121] are introduced. ODD, in which log instrumentation plays a key role, emphasizes the exposure of the state and the behavior of a **System Under Study (SUS)** during runtime. Unfortunately, other than describing the capability of their accompanying LUs, existing cloud platforms (e.g., Microsoft Azure [2], Amazon Web Service [3], and Google Cloud [8]) provide little information on effective development and maintenance of the logging code. Hence, in this article, we have conducted a systematic survey [86] on the instrumentation techniques used in software logging. The contributions of this article are:

- This is the first survey that systematically covers the techniques used in all three software log instrumentation approaches: conventional logging, rule-based logging, and distributed tracing.
- Through the process of systematic survey, we have identified nine challenges in four categories associated with log instrumentation and described their proposed solutions throughout the three steps in the log instrumentation phase. We have also discussed the limitations and future work associated with these state-of-the-art solutions if applicable. The challenges, the solutions, and the discussions will be useful for both practitioners and researchers who are interested in developing and maintaining software logging solutions.

Paper Organization

The structure of this article is organized as follows: Section 2 describes the workflow of the log instrumentation phase. Section 3 provides the overview of our systematic survey process and summarizes the findings from the studied papers. Section 4 discusses the challenges and solutions

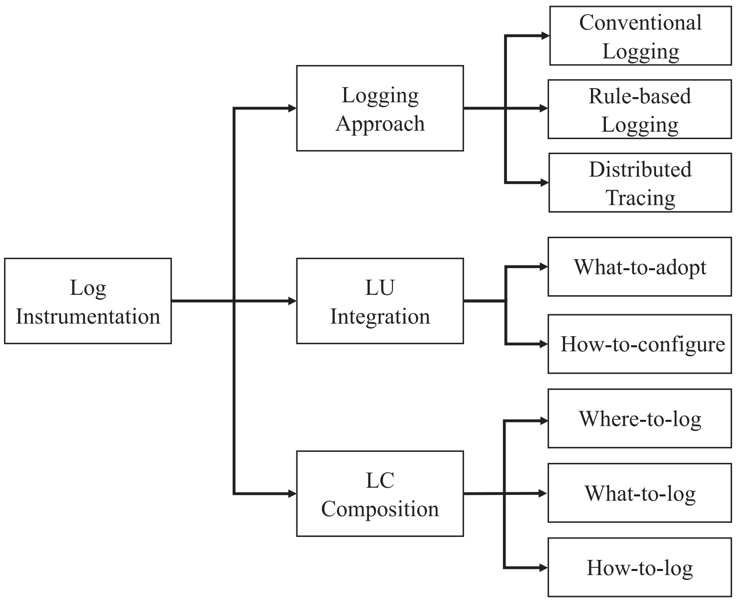


Fig. 2. The steps in the three phases of log instrumentation.

associated with log instrumentation. Section 5 presents the discussions and future research directions. Section 6 concludes this survey.

2 THE WORKFLOW OF SOFTWARE LOGGING INSTRUMENTATION

In this section, we will introduce the overall workflow of software log instrumentation through a running example. Section 2.1 provides an overview of the log instrumentation phase; Section 2.2 describes the three general logging approaches; Section 2.3 proposes two concerns in LU Integrations; and Section 2.4 illustrates three steps of logging code composition.

2.1 An Overview of the Log Instrumentation Phase

Software logging consists of two phases: (1) **Log Instrumentation**, which concerns about the development and maintenance of the logging code, and (2) **Log Management**, which concerns about the collection and the analysis of the generated log messages. Here, we further explain the three steps in the log instrumentation phase, which is the focus of this survey. The detailed steps are shown in Figure 2.

(1) *Logging Approach*: Logging is a cross-cutting concern, as the LC snippets are scattered across the entire system and tangled with the feature code [82]. In addition, logging incurs performance overhead [54] and if not careful may slow down the system execution and impact user experience. Hence, additional logging approaches have been proposed to resolve some of these issues. However, they also introduce additional problem(s). For example, although **Aspect-Oriented Programming (AOP)** improves the modularity of the LC snippets, it introduces steep learning curves of different programming paradigms and is difficult to generalize individual logging concerns into rules [44]. Developers have to first decide which logging approach to adopt for their SUS before instrumenting their SUS with LC.

(2) *LU Integration*: Instead of directly invoking the standard output functions like `System.out.pri nt`, developers prefer to instrument their systems using LUs such as

SLF4J [19] for Java and spdlog [20] for C++ for additional functionalities like thread-safety (synchronized logging in multi-threaded systems) and verbosity levels (controlling the amount of logs outputted). While integrating LUs, developers have to address the following two concerns: *what-to-adopt*: different LUs provide different functionalities [44]. Depending on the actual usage context, developers may integrate existing third-party LUs or develop their own; and *how-to-configure*: each project may contain one or more LU(s), each of which has many different configuration options. It is important to configure LUs correctly and effectively so the SUS can produce high-quality log messages during runtime.

(3) *LC Composition*: Once the developers integrate the LU(s), they need to insert LC into the SUS to expose the state and behavior of the SUS during runtime. While composing LC, developers have to address the following three concerns: *where-to-log*: determining the appropriate logging points; *what-to-log*: providing sufficient information in the LC; and *how-to-log*: developing and maintaining high-quality LC.

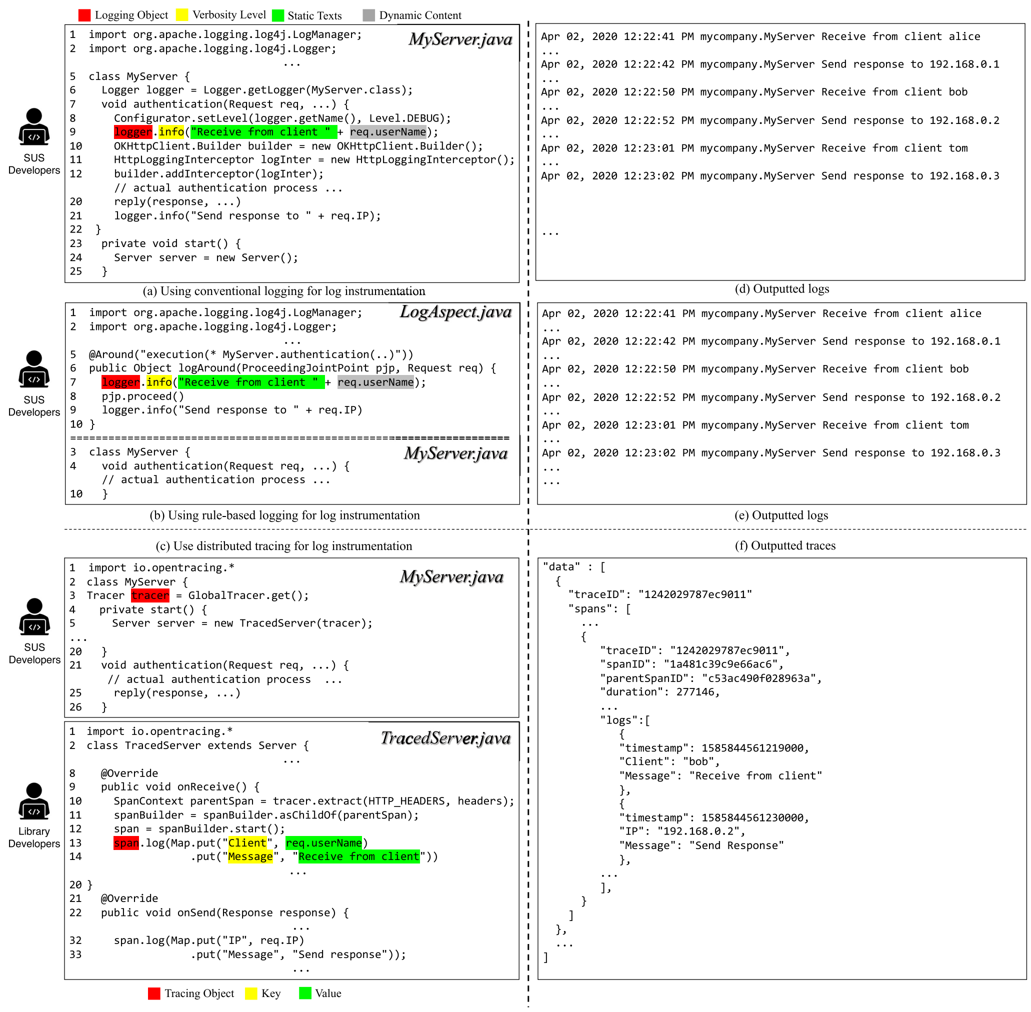
2.2 Logging Approach

The first step in the log instrumentation phase is to choose an appropriate logging approach. There are three general logging approaches, each of which has its pros and cons. We will explain and compare these logging approaches by going through a running example. The scenario is about logging the behavior of a web server during the user authentication process. This scenario is implemented using all three general logging approaches, as shown in Figure 3.

2.2.1 Conventional Logging. The code snippet using the conventional logging approach is shown in Figure 3(a). Before we can instrument the SUS with LC snippets, we have to first import the LU(s), which provides functionalities of conventional logging. As shown from line 1 and 2 of the code snippet, we use `Log4J 2` library [14], a popular LU for Java-based systems. Then a logging object, which is responsible for performing the log instrumentation in the rest of this example, is created at line 6. Line 9 and 21 show two lines of LC snippets, which record the user names and their IP addresses before and after the authentication process. Similar to standard I/O methods like `System.out.println` or `System.err`, an LC snippet contains static texts and dynamic contents. In addition, it also contains a logging object as well as a verbosity level to control the amount of outputted log messages. Take the LC snippet at line 9 as an example. The four components are highlighted in different colors: the logging object (`logger`) in red, the verbosity level (`info`) in yellow, the static texts (“`Received from client`”) in green, and the dynamic contents (`req.userName`) in grey.

A sample snippet of the generated log messages is shown in Figure 3(d). In addition to the static texts and dynamic contents, each log message also contains basic information like timestamp and the location of the logging code. Similar to the natural language text, the resulting log messages are usually loosely formatted and cannot be easily parsed by the computer programs [138].

Conventional logging is very easy to set up and the resulting LC snippets can be placed almost anywhere in the SUS. However, there are two main issues concerning conventional logging: (1) *Cross-cutting Concerns*: the resulting LC snippets are scattered across the entire system and tangled with the feature code [32, 34, 82, 87]. This results in challenges in developing and maintaining high-quality LC, while the SUS evolves. To resolve this issue, the rule-based logging approach is introduced (Section 2.2.2). (2) *Lack of Execution Context*: It is very challenging to correlate log messages from different processes or even machines [133]. This is especially the case for large-scale distributed systems. Hence, distributed tracing is introduced (Section 2.2.3).



MyServer.java

```
1  import org.apache.logging.log4j.LogManager;
2  import org.apache.logging.log4j.Logger;
                                      ...
5  class MyServer {
6    Logger logger = Logger.getLogger(MyServer.class);
7    void authentication(Request req, ...) {
8      Configurator.setLevel(logger.getName(), Level.DEBUG);
9      logger.info("Receive from client " + req.userName);
10     OKHttpClient.Builder builder = new OKHttpClient.Builder();
11     HttpLoggingInterceptor logInter = new HttpLoggingInterceptor();
12     builder.addInterceptor(logInter);
       // actual authentication process ...
20     reply(response, ...)
21     logger.info("Send response to " + req.IP);
22  }
23   private void start() {
24     Server server = new Server();
25   }
```

(a) Using conventional logging for log instrumentation

```
Apr 02, 2020 12:22:41 PM mycompany.MyServer Receive from client alice
...
Apr 02, 2020 12:22:42 PM mycompany.MyServer Send response to 192.168.0.1
...
Apr 02, 2020 12:22:50 PM mycompany.MyServer Receive from client bob
...
Apr 02, 2020 12:22:52 PM mycompany.MyServer Send response to 192.168.0.2
...
Apr 02, 2020 12:23:01 PM mycompany.MyServer Receive from client tom
...
Apr 02, 2020 12:23:02 PM mycompany.MyServer Send response to 192.168.0.3



...
```

(d) Outputted logs

LogAspect.java

```
1  import org.apache.logging.log4j.LogManager;
2  import org.apache.logging.log4j.Logger;
                                      ...
5  @Around("execution(* MyServer.authentication(..)"))
6  public Object logAround(ProceedingJointPoint pjp, Request req) {
7    logger.info("Receive from client " + req.userName);
8    pjp.proceed()
9    logger.info("Send response to " + req.IP)
10 }
=====================================================================
3  class MyServer {
4    void authentication(Request req, ...) {
     // actual authentication process ...
10   }
```

MyServer.java

(b) Using rule-based logging for log instrumentation

```
Apr 02, 2020 12:22:41 PM mycompany.MyServer Receive from client alice
...
Apr 02, 2020 12:22:42 PM mycompany.MyServer Send response to 192.168.0.1
...
Apr 02, 2020 12:22:50 PM mycompany.MyServer Receive from client bob
...
Apr 02, 2020 12:22:52 PM mycompany.MyServer Send response to 192.168.0.2
...
Apr 02, 2020 12:23:01 PM mycompany.MyServer Receive from client tom
...
Apr 02, 2020 12:23:02 PM mycompany.MyServer Send response to 192.168.0.3
...
...
```

(e) Outputted logs

(c) Use distributed tracing for log instrumentation

MyServer.java

```
1  import io.opentracing.*
2  class MyServer {
3  Tracer tracer = GlobalTracer.get();
4    private start() {
5      Server server = new TracedServer(tracer);
...
20   }
21   void authentication(Request req, ...) {
      // actual authentication process  ...
25     reply(response, ...)
26   }
```

TracedServer.java

```
1  import io.opentracing.*
2  class TracedServer extends Server {
                                  ...
8    @Override
9    public void onReceive() {
10     SpanContext parentSpan = tracer.extract(HTTP_HEADERS, headers);
11     spanBuilder = spanBuilder.asChildOf(parentSpan);
12     span = spanBuilder.start();
13     span.log(Map.put("Client", req.userName)
14                .put("Message", "Receive from client"))
                                   ...
20 }
21   @Override
22   public void onSend(Response response) {
                                   ...
32     span.log(Map.put("IP", req.IP)
33                .put("Message", "Send response"));
                                   ...
```

(f) Outputted traces

```
"data" : [
  {
    "traceID": "1242029787ec9011"
    "spans": [
      ...
      {
        "traceID": "1242029787ec9011",
        "spanID": "1a481c39c9e66ac6",
        "parentSpanID": "c53ac490f028963a",
        "duration": 277146,
        ...
        "logs":[
          {
          "timestamp": 1585844561219000,
          "Client": "bob",
          "Message": "Receive from client"
          },
          {
          "timestamp": 1585844561230000,
          "IP": "192.168.0.2",
          "Message": "Send Response"
          },
        ...
        ],
      }
    ]
  },
  ...
]
```

Fig. 3. An example of user authentication scenario instrumented with three general logging approaches.

2.2.2 *Rule-based Logging.*

Different from the conventional logging approach, in which the LC inter-mixes with the feature code, the rule-based logging approach generalizes the logging behavior by specifying a set of rules. This greatly improves the modularity of the LC and hence provides much better support for developers to track and maintain their LC while the SUS evolves.

Aspect-Oriented Programming (AOP)-based logging is one of the most commonly used rule-based logging techniques. AOP is a programming paradigm, which is designed to improve modularity by reducing the amount of cross-cutting concerns [82], one of which is software logging. AOP-based logging has been used to support diagnosing functional failures [34, 87]. Developers define rules through aspect files. A typical aspect file consists of pointcuts and advice. A pointcut is to define the point of execution where the cross-cutting concern (e.g., logging) needs to be applied. An advice is the additional code (e.g., LC) being executed when the pointcut is reached.

Figure 3(b) continues our running example by using the `AspectJ` LU, which is a very popular AOP-based approach for Java-based systems. The file `LogAspect.java` acts as the aspect file. The rules (a.k.a. instrumented points) are defined through the Java annotation at line 5. In this example, the annotation @Around means both the beginning and the end of the methods will be instrumented. The value within the brackets specifies the instrumented methods. In this example, the method with name authentication within class MyServer will be instrumented. The instrumented code is defined at line 7 and line 9, which outputs the same log messages as the conventional logging example. The file Server.java does not include any LC, as all the LC is modularized and specified in the aspect files. As shown in Figure 3(e), the same log messages will be outputted during runtime.

On one hand, rule-based logging enables developers to separate rules with actual instrumentation. Updating LC is easy, as developers only need to revise the rules without modifying code at multiple locations. This improves the modularity of the LC. On the other hand, rule-based logging lacks flexibility, as LC cannot be instrumented anywhere due to the implementation limitation [27, 44, 104]. For example, when we adopt AOP in Spring framework, only public methods can be advised in Spring AOP, whereas private or protected methods cannot [4, 17].

2.2.3 Distributed Tracing. Although it is flexible and easy to perform conventional logging, the resulting log messages are free-formed text, which cannot be easily cross-linked across different processes or even machines. This will be a major problem for distributed systems, in which one scenario may be executed on multitple machines. To cope with this challenge, distributed tracing is introduced. Different from conventional logging, in which developers of a SUS mainly perform the log instrumentation activities, library developers are mainly responsible for the task of log instrumentation. Although developers of a SUS can perform additional log instrumentations, their main task is to import the tracing library and perform some setup actions. As a result, the generated log messages are structured and can be connected by a set of common variables. In the context of distributed tracing, these structured log messages connected together are also referred to as an end-to-end *trace*. For brevity, we call this as a *trace* in the rest of this section.

Figure 3(c) continues our running example by using distributed tracing as our log instrumentation approach. This example uses OpenTracing, a very popular LU supporting the distributed tracing-based logging approach [108]. For the code snippet at the top of Figure 3(c), a tracer object and a traced server instance are created at line 3 and line 5, respectively. Other than this, there are no additional log instrumentation efforts required from developers of the SUS. The actual LC composition is done by the library developers whose code snippet is shown at the bottom of Figure 3(c). They implement TracedServer, which extends the original Server class and overrides two important methods: onReceive and onSend. These two methods will be invoked when receiving a request and sending a reply, respectively.

A typical *trace* in the context of distributed tracing consists of multiple log messages that are connected together. These connected log messages form a complete request workflow. In OpenTracing, such log messages are called spans. In both the client side and the server side, library developers compose the span log using APIs like span.log. These spans are passed from the client side to the server side by certain communication protocols (e.g., HTTP) so spans from both ends can be connected. In this way, a complete trace recording information from both ends can be generated. In our example, we only show the code on the server side, because the tracing code on the client side is similar. To notice, inside method onReceive, where we receive the client's request, we extract the context data that is sent from the client at line 10. At line 11, we create the span as a child of the span that we extract from the client. At line 13, we store the same message as the example in conventional and rule-based logging. After the request is processed, inside the onSend method, the span log at line 32 will be sent to the client.

The generated traces are shown in Figure 3(f). As we can see, the traces are structured in JSON format. JSON stores information in a key-value fashion. For example, the recorded information

Table 1. Comparison among Three Logging Approaches

	Conventional Logging	**Rule-based Logging**	**Distributed Tracing**
Who	SUS developers	SUS developers	Library developers
Filtering	Verbosity level	Verbosity level	Sampling
Format	Free form	Free form	Structured
Domain	General	General	Distributed systems
Flexibility	High	Low	Medium
Scattering	High	Low	Low

of the LC snippet at line 13 has two keys: `Client` and `Message`. The key (`Client`) corresponds to the runtime information: the `userName` of the request and the key (`Message`) correspond to the static texts describing the logging context. Compared to conventional and rule-based logging, log messages generated by distributed tracing are more structured. The related log messages can easily be linked across different machines by the associated `traceID`.

2.2.4 Comparison among Three Logging Approaches. Table 1 compares these approaches among the following six dimensions:

- *Who* refers to the type of developers responsible for performing the log instrumentation tasks. The SUS developers are mainly responsible for the log instrumentation tasks if they adopt the conventional or rule-based logging approaches. If the SUS imports third-party libraries, then they may need to configure the LUs within these libraries to gain the full picture of the SUS behavior during runtime. On the contrary, third-party library developers are the ones responsible for most of the log instrumentation tasks if they adopt distributed tracing approach. Only if needed, SUS developers may add additional LC in the SUS.
- *Filtering* refers to the process of removing the unwanted log messages during runtime to reduce overhead. The verbosity level is used in conventional and rule-based logging to indicate the severity of LC. While in distributed tracing, sampling is adopted to filter log messages. The sampling decision can be controlled by pre-defined probability, rate or even adaptive.
- *Format* refers to the requirements on the structure and the contents of the generated log messages. Instrumenting free form LC is mostly adopted in conventional and rule-based logging. However, distributed tracing utilities record the information in a more structured way. For example, key-value pair is a popular paradigm to structure the generated log messages.
- *Domain* refers to the categories of applicable SUS for each logging approach. Conventional and rule-based logging can be applied in almost any types of SUS, while distributed tracing is mostly adopted in distributed systems.
- *Flexibility* refers to the feasibility and the effort needed for a particular logging approach to be applied under various instrumentation scenarios. Conducting conventional logging is most flexible, as SUS developers can insert LC in any program points. Instrumenting LC in distributed tracing is less flexible, as most of the LC snippets are in the boundaries of software components. Rule-based logging is the least flexible, because the locations of LC snippets need to follow pre-defined rules.
- *Scattering* refers to the spread of the instrumented LC snippets across the code base by adopting a particular logging approach. The degree of scattering of LC in conventional logging is high, because of its high flexibility. However, LC snippets in rule-based logging and

distributed tracing is less scattered, as its locations follow designated rules (e.g., beginning of a method) or certain patterns (e.g., before an RPC call).

2.3 LU Integration

Instead of directly invoking the standard output functions like `System.out.print`, developers prefer to instrument their SUS using LUs (e.g., `SLF4J` [19] for Java and `spdlog` [20] for C++) due to additional functionalities such as thread-safety (synchronized logging in multi-threaded systems), data archival configuration (automated rotation of the log files), and verbosity levels (controlling the amount of log messages outputted). There are generally two concerns associated with LU integration:

- **what-to-adopt**: With the increasing amount of LUs available in the wild [44], integrating appropriate LUs according to the requirements of individual SUS is important. The problem of what-to-adopt focuses on this matter with regard to the maintainability and security compliance of LUs. After that, it is also important to configure these LUs to increase their usability.

  Modern software often leverage the functionalities provided by the LUs to instrument their SUS. A study [44] on 11,194 Java-based GitHub projects shows that there are more than 3,000 LUs being adopted in the wild. For example, many developers adopt LUs such as `Log4j` [13] and `Apache Commons Logging` [1] to instrument their Java-based SUS [90]. Many of these projects adopt multiple LUs or even implement their own LUs. Developers need to decide which or whether existing LU(s) are needed for their SUS. Furthermore, for projects with LU(s) integrated already, developers have to determine if they would like to migrate to other or newer LU(s). For example, in Figure 3(a), developers adopt `Log4J 2` to record the runtime behavior of the server. Since there is a dependency of the third-party library `OKHttp3` to manage `HTTP` connections in this program, the LU `HTTPLoggingIntertor`, provided by `OKHttp3` shown in line 11, should be adopted as well.
- **how-to-configure**: LUs contain many different configuration options. They can be related to controlling the amount of log messages outputted, or the location of the log files, and the size of the log files. Developers need to properly configure LUs for their SUS to gather enough logging data while minimizing the performance overhead and storage requirements. For example, the default verbosity level in `Log4J 2` can be configured either statically (e.g., through a configuration file) or dynamically (through APIs). In our running example, line 8 in Figure 3(a) shows how to configure the default level of LUs. For LUs provided by third-party libraries, developers need to configure the options one-by-one.

2.4 LC Composition

The last step of the log instrumentation phase is LC composition. There are three sub-steps in LC composition:

(1) The step of **where-to-log** is about deciding the appropriate logging points. Various studies [42, 60, 109, 128, 130] have shown that logging is pervasive in software development process. Developers usually rely on their experience or gut feelings when deciding on the logging points in the source code. In our running example, Figure 3(a) shows that developers choose to instrument logging code snippets at the entry and exit of the method `authentication` to record the program state. On one hand, logging too little will hinder the diagnosability of log messages. For example, missing logging statements in exception blocks will cause incomplete information of failures, making failure diagnosis more difficult [127]. Incomplete LC snippets also hinder developers' understanding, hurting LC

quality, since they can only recover ambiguous execution paths from the execution log messages [132]. On the other hand, although excessive logging—in which LC snippets are inserted everywhere in the source code—will provide rich runtime information, it will bring in huge performance overhead and high storage cost associated with the generated log messages. In addition, it is very challenging to diagnose problems by analyzing large volumes of log messages, most of which are not related to the problematic scenarios [74].

(2) The step of **what-to-log** is about providing sufficient information in the three components of each LC snippet:
   - *Verbosity level* specifies whether an LC snippet should be outputted during the execution of SUS. Choosing an appropriate verbosity level for an LC snippet is important. For example, if an LC snippet records information about a failed execution, then the verbosity level should be set as `error` or `fatal`. If it is mistakenly set as `debug`, such log messages may not be outputted or even if they do, developers may neglect them. Such neglection could impact customer experience and negatively impact the product quality.
   - *Static texts* describe the logging context in a human readable manner. Currently, developers are responsible for manually composing the static texts in the LC snippets. Poorly written or outdated static texts may cause confusion for the practitioners and impact their various log analysis tasks.
   - *Dynamic contents* reflect the state of SUS during runtime. They are the results of executing variables and method invocations included in each LC snippet. It is important to record the necessary runtime information to satisfy various logging needs from the developers and operators.

(3) The step of **how-to-log** is about developing and maintaining high-quality LC, which is scattered across the entire system and tangled with the feature code. Although the rule-based logging approach provides better management of LC, many industrial and open source systems still choose to inter-mix LC with feature code [40, 41]. A study [78] shows that 20%–45% of the LC has been changed at least once during their lifetime. The median number of days between an LC snippet is introduced and its first change ranges from 1 to 17 days. Unlike feature code, whose quality can be verified via testing, the correctness of LC is very difficult to verify. This can hinder program understanding or even cause runtime issues like crashes [9].

3 AN OVERVIEW OF OUR SYSTEMATIC SURVEY

In this section, we will first describe our systematic survey process in Section 3.1. Then, we will summarize our survey results in Section 3.2. Finally, we will explain the differences of our survey against existing works in Section 3.3.

3.1 Methodology

A systematic survey is a type of literature review, which uses systematic approaches to identifying and analyzing the primary studies related to a particular research topic [86]. The main benefits of conducting a systematic survey are: (1) the results of selected studies are less likely to be biased, since it applies a pre-defined search strategy; (2) the search process is documented so the study could be easily replicated.

Below, we briefly describe our systematic survey process. In addition to "logging" and "instrumentation," we also include the word "tracing" as our search keywords, as "logging" and "tracing" are used interchangeably in the research literature and practice. For example, "trace" is usually a common verbosity level defined in many LUs to capture information flow through the SUS

[24, 92]. In addition, many tracing frameworks [79, 99, 120] also use the term "logging" to record various runtime behavior of the SUS. Hence, to capture all the possible related works, we search IEEE Xplore, ACM, and DBLP publication repositories with the root form of these three words: *log*, *trace*, and *instrument*.

- For IEEE Xplore, we set the "Publication Title" field to be software, and check if the "Metadata" field satisfies one of the following criteria:
  - containing the words of logging and software logging, or logging practices; or
  - containing the words of tracing, software tracing, or tracing practices; or
  - containing the words of instrumentation or software instrumentation.
- Similarly, for ACM, we set the "acmdlCCS" field to be software and check if the title and the abstract contains the word "logging," "tracing," or "instrumentation."
- As DBLP's API does not support advanced search functions, we mainly use it for verification purposes.

We then manually check the search results by employing the following inclusion and exclusion criteria:

(1) We exclude all the papers that only study the issues in the log management phase by reading through the abstract. For example, there are many studies focusing on log analysis (e.g., References [75, 118, 124]) and log abstraction (e.g., References [67, 76]), which are not relevant to this survey.
(2) We only include the papers that are published in software engineering or computer systems–related venues, as our target audience is software practitioners or researchers.
(3) Since we are only focusing on SUS, which sits on top of operating systems and are connected by computer networks, we will exclude papers that focus on the logging at the kernel (e.g., Reference [62]) or network levels (e.g., Reference [64]).

After we gather the first batch of papers, we further apply the snowballing method [116] from the references of these papers as well as looking through the papers that cite them. This process results in a total of 69 papers that match our criteria. To verify the completeness of the surveyed papers, the final results include all the papers we knew beforehand that are related to software log instrumentation (e.g., References [41, 60, 128]). We include these 69 papers and their metadata (e.g., year, publication venues) as well as the initial search results from IEEE and ACM in our replication package [18].

3.2 Summary of Our Survey

We performed our paper collection process on October 30, 2019. Figure 4 illustrates the number of related papers between these 23 years (1997–2019), as the first research paper in this area [80] appears in 1997. There is a clear increasing trend in terms of the number of research papers over the years on this topic. In particular, the research interests in this area spiked after 2012, as 62% (43) of the studied papers have been published since then. This is also partially a result of the increasing number of research in areas in software engineering during this period [101].

After carefully studying each paper, we have identified nine challenges, which are further grouped into the following four major categories. Note that some papers may touch on multiple challenges.

(1) **Usability** refers to the log instrumentation techniques that facilitate the adoption of various logging techniques. There are two specific challenges in this category:
   (a) *Configurability* refers to the challenge of whether the studied paper provides support to ease the configuration process of various log instrumentation techniques.

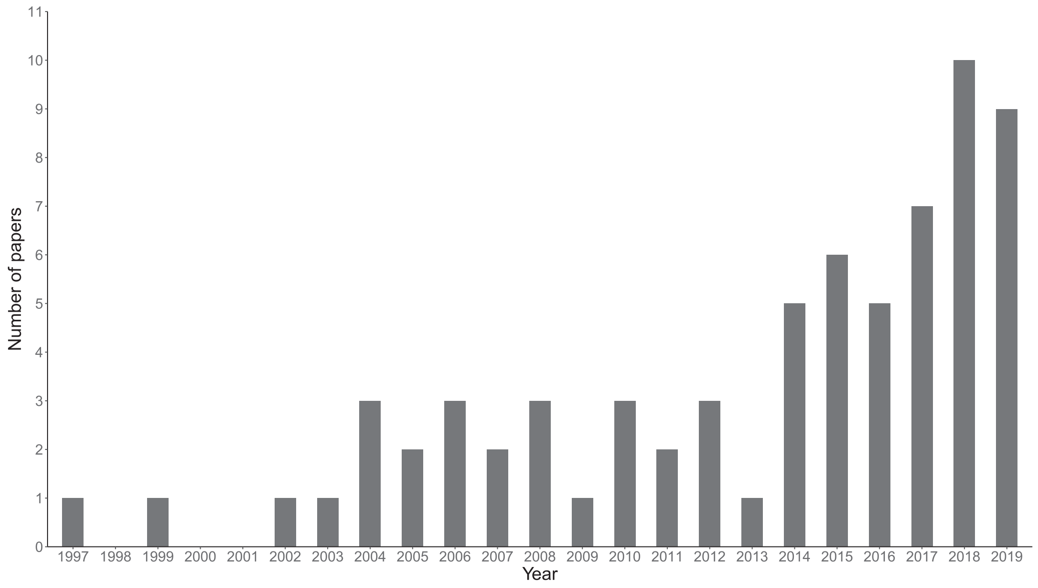


Fig. 4. Papers related to Log instrumentation from 1997 to 2019.

   (b) *Performance Overhead* refers to the challenge of whether the study aims to minimize the slowdown caused by logging.
(2) **Diagnosability** refers to the log instrumentation techniques that support the analysis and debugging tasks of various functional and non-functional problems. This category consists of the following two challenges:
   (a) *Failure Diagnosis* refers to the challenges associated with providing sufficient logging information to diagnose functional failures.
   (b) *Performance Analysis* refers to the challenges associated with providing sufficient logging information to detect and debug performance problems.
(3) **LC Quality** refers to the log instrumentation techniques that improve various development aspects of LC. This category consists of the following three challenges:
   (a) *Clarity* refers to the challenge of making LC easy to understand and less ambiguous to both developers and operators.
   (b) *Maintainability* refers to the challenge of supporting the maintenance and evolution of LC, as LC is scattered across the entire system and tangled with the constantly evolving feature code.
   (c) *Consistency* refers to the challenges on ensuring uniform styles of logging across different components of SUS.
(4) **Security Compliance** refers to the log instrumentation techniques that address the safety or legal concerns of SUS. This category consists of the following two challenges:
   (a) *Auditing* refers to the challenge of recording a serial of security relevent events to meet various legal regulations like the Sarbanes-Oxley Act of 2002 [22].
   (b) *Forensic Analysis* refers to the challenges of recording the user activities to support investigations on criminal activities such as an intrusion or fraud detection.

Table 2 further breaks down the surveyed papers by associating each study with the tackled challenge(s), the proposed solution(s), and the applied step(s). Multiple solutions can be proposed to solve one challenge. The solution(s) can also be applied at multiple steps or sub-steps from the

Table 2. An Overview of This Survey by Categorizing Different Papers under the Challenge Categories, the Detailed Challenges, the Solutions That They Developed, as Well as the Applied Steps

Category	**Challenges**	**Solutions**	**Applied Steps**
Usability	Configurability	History [134]	LU/How-to-configure
	Performance Overhead	Post-processing [100, 125]	LA/Conventional logging
		Sampling [39, 57, 58, 79, 99, 115, 120, 122]	LA/Distributed tracing LU/How-to-configure
		Cost-optimization [54, 126, 131, 132]	LC/Where-to-log
Diagnosability	Failure	Rule-generation [28, 49, 51, 52, 123]	LA/Rule-based logging
		Program analysis [50, 53, 72, 73, 127, 129]	LC/Where-to-log LC/What-to-log
	Performance Analysis	Causality tracking [29, 30, 39, 46, 47, 56–58, 68, 79, 80, 99, 102, 111, 115, 120, 122, 135]	LA/Distributed tracing LC/Where-to-log LC/What-to-log
LC Quality	Clarity	NLP [66] Visualization [110] Entropy [113]	LC/What-to-log
	Consistency	Machine learning [60, 83, 88, 89, 91–94, 97, 137] Code cloning [128]	LC/*
	Maintainability	Domain-specific Language [26, 35]	LA/Rule-based logging
		History [41, 65, 95]	LC/How-to-log
Security	Auditing	AOP [26] LU design [44]	LA/Rule-based LU/What-to-adopt
	Forensics	Heuristics [84, 85]	LC/What-to-log

Under the applied steps column, "*" means all the sub-steps within this step will be applied.

log instrumentation phase. For example, Dapper [120] tries to solve the challenge of controlling the performance overhead of logging. It is classified as the sampling-based solution. The steps that can be applied are the distributed tracing approach from the Logging Approach step and the how-to-configure sub-step of the LU Integration step. One study can also provide solutions for multiple challenges. For example, in addition to addressing the performance overhead challenge mentioned above, Dapper [120] also proposes a causality tracking-based solution for supporting performance analysis.

Figure 5 shows the distribution of studied papers by the challenges that they tackled. Note that they do not add up to 69, as some papers tackle multiple challenges. The majority of them focus on the diagnosability (45%) or LC quality (43%) aspects. 25% of papers tackle the usability challenge and only 6% of papers study security-related challenges.

Section 4 will explain each challenge and the proposed solutions by the studied papers.

3.3 Comparing Against Existing Surveys

There are three existing works [37, 112, 114] that are related to our survey on log instrumentation:

- Rong et al. [112] conducted a systematic review on the log instrumentation practices on one type of logging approach, conventional logging. They did not cover other logging

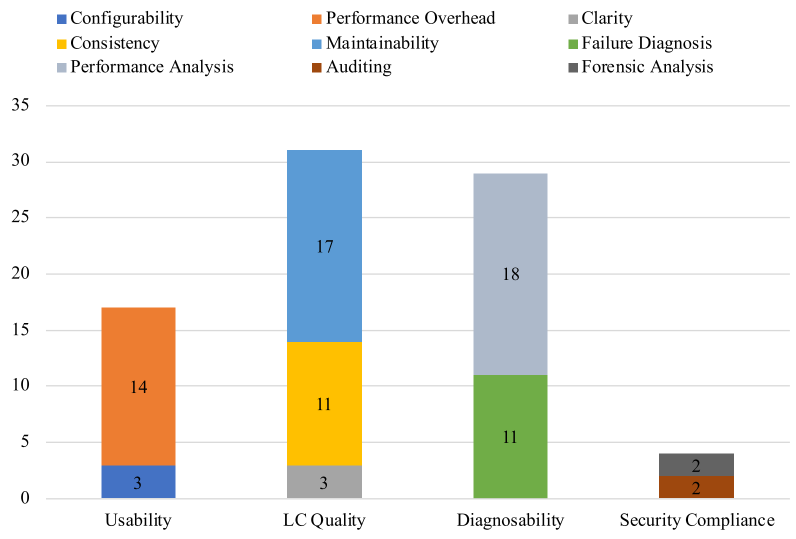


Fig. 5. Paper distribution classified by associated challenges.

approaches nor discussed the techniques in LU integration. Furthermore, compared to Reference [112], we have placed each studied paper in the context of associated challenges so practitioners or researchers can easily locate the relevant solutions to adopt and understand their limitations if any.

- Sambasivan et al. [114] conducted a survey on distributed tracing systems, which are logging frameworks to support monitoring and diagnosing problems for distributed systems. They examined the features of 15 frameworks (e.g., preserving causal relationships or visualization) and focused on the diagnosability category of the log instrumentation techniques. Our survey covers all three general logging approaches, which include distributed tracing. Furthermore, on top of diagnosability, we have identified and examined additional challenge categories (e.g., usability and security) and their associated solutions.
- Candido et al. [37] conducted a systematic review on logging techniques for contemporary software monitoring. They focused their study on the following four dimensions: log engineering, log infrastructure, log analysis, and log platforms. Log engineering is only focused on logging code composition of conventional logging, while our survey not only discusses other logging approaches, but also covers LU integration. The remaining three dimensions are all related to the log management phase, which is not the focus of this survey.

In addition, there are also surveys related to the dynamic instrumentation techniques used during monitoring [38, 63, 69], as the focus of this article is on log instrumentation, which generally refers to static instrumentation techniques applied during the software development and maintenance.

3.4 Comparing against Industrial Work and Open Source Software

Several existing studies were conducted in an industrial environment. A study is considered as industrial work, if the affiliation of the first author is an industrial organization (e.g., research labs or software companies) and the study was conducted mainly on industrial software systems. Two studies have been conducted on researching and developing the infrastructure of distributed tracing systems from Google [120] and Facebook [79]. Three studies from Microsoft [31, 54, 60] tackled the where-to-log step in the LC Composition phase. Two studies from Critiware [28, 109] proposed rule-based logging in the Logging Approach phase. There is a lack of industrial work in other steps or phases, such as the phase of LU integration. The reasons are two-fold: (1) log instrumentation is highly tangled with the source code, due to confidentiality reasons, many companies are not willing to make their internal infrastructure or practices public; and (2) log instrumentation is commonly conducted as after-thought efforts [128], because industrial organizations might not be aware of the benefits of proactively developing and maintaining LC snippets.

Similar to other systematic literature surveys [37, 112, 116], in this article, our main focus is on published research papers instead of the tools and frameworks. Additionally, in favor of the practitioners, we also list the tools and frameworks we have found so far. There are many industrial systems and open source software available for the phase of Logging Approach. For example, the most commonly used logging utilities include Log 4J 2 [14] and Apache Commons Logging [1], which are using the conventional logging approach. Software systems such as OpenTracing [108], OpenCensus (Google) [16], Jaeger (Uber) [10], Zipkin (Twitter) [25] are widely used for distributed tracing. Closed source commercial solutions and platforms also exist for log instrumentation and management, such as Datadog [5], Lightstep [12], and Splunk [21]. However, to our best knowledge, there is a lack of widely adopted industrial and open source tools and frameworks focusing on the phases of LU Integration and LC Composition.

4 CHALLENGES AND SOLUTIONS

In this section, we discuss in detail about the nine different challenges in four categories and report the solutions extracted from the existing literature. For each of the solutions, we identify the applied steps/sub-steps of the log instrumentation phase and discuss the pros and cons.

4.1 Usability

The usability of log instrumentations relates to how effectively users can adopt various logging techniques and manage logging behavior during runtime. This is challenging because: (1) logging behavior is managed through various configuration functionalities. As large-scale software systems evolve rapidly, their logging behavior needs to be changed accordingly; and (2) intensive logging, although helpful to many tasks to a certain degree, will inevitably downgrade the performance of the SUS and may even interfere with the normal operation. Hence, the performance overhead caused by logging has to be controlled within a certain threshold. In this subsection, we will discuss these two challenges (configurability and performance overhead) and the proposed solutions to address them.

4.1.1 Configurability. Configuring LUs is a non-trivial task. A study conducted by Hassani et al. [65] shows that log configuration–related issues could possibly cause serious problems such as runtime exceptions and missing logs. These issues are often caused by inconsistencies between the log configurations and feature code. For example, a wrongly named logger could cause a `File Not Found` exception, preventing the logs from being written to disks. There is a lack of automated tool support for detecting such inconsistencies. There is one solution proposed for this challenge:

- *History-based solution.* Zhi et al. [134] conduct an empirical study by analyzing the evolution of logging configurations of 10 open source projects and 10 industrial projects from Alibaba. Many of the studied open source projects (e.g., Hadoop, HBase) are also being widely used in industry. They find, for the ease of managing logging behavior in different components of a large-scale system, the loggers will be named after three conventions: topic-based, package-based, and mixed naming. In industrial projects, topic-based naming is used most often while in open source projects, and package-based naming is used most frequently. Furthermore, the names of loggers are changed frequently due to inconsistencies and software evolution. Based on this finding, they propose a history-based solution to identify invalid loggers. It compares the names of all the loggers in configuration files and the ones mentioned in source code files. The unmatched ones between the two sets are reported to developers. All the detected issues are confirmed by developers of the SUS. This solution is applied in the `how-to-configure` sub-step in the LU Integration step. Although it is able to statically detect inconsistencies in log configurations, many other types of issues, such as low readability of loggers and performance-related parameter tuning, still cannot be detected automatically. Hence, more research is needed to facilitate log configurations so they can co-evolve with other components of the SUS.

4.1.2 Performance Overhead. On one hand, effective log analysis requires rich logs that are generated by the execution of instrumented LC snippets. On the other hand, excessive log instrumentation will cause runtime performance overhead and impact customer experience. This challenge is about how to balance the performance cost and the benefits of log instrumentation. The following three types of solutions have been proposed for this challenge:

- *Post-processing-based solution*: To reduce the I/O cost associated with conventional logging, a post-processing-based solution has been proposed. The main idea is to delay the output of log messages only when needed. NanoLog [125] is a nanosecond scale logging system implemented in C++. It first statically analyzes the source code during compilation time and generates a compression function for each LC snippet. During runtime, only the compact log messages would be generated. The full textual version of the log messages will only be generated during the post-processing time. In this way, NanoLog can achieve 1–2 order of magnitude faster than conventional logging libraries (e.g., Log4J 2 [14], spdlog [20]). Similarly, `Log++` [100] is a logging system that optimizes the logging performance for the Node.js platform by postponing log generation offline.

  The post-processing-based solution is applied in the `Conventional Logging` sub-step in the Logging Approach step. Although the proposed solutions are faster than existing logging libraries, they are limited to certain programming languages (C++ and Javascript). In addition, this solution would not be applicable if logs are also used for real-time analysis purposes (e.g., monitoring), as real-time analysis requires the logs to be immediately available for various automated tools. Hence, the runtime would be the same as the common conventional logging approach.
- *Sampling-based solution*: One of the main issues associated with distributed tracing is the performance overhead incurred to the SUS. Many LUs (e.g., Google's Dapper [120] or Facebook's Canopy [79]) implementing distributed tracing-based logging approaches usually support sampling, which is a technique to selectively generate and preserve log messages to reduce the runtime overhead. There are three sampling techniques: head-based sampling, tail-based sampling, and unitary sampling [114, 119]:
  - *Head-based Sampling*: The sampling decision is made at the beginning of every trace. It either preserves the whole trace (including every trace point) or no trace at all. Head-based sampling techniques can be further divided into the following three types of sampling techniques: (1) *Probability sampling*, which makes the sampling decision based on a pre-defined probability; (2) *Rate-limit sampling*, which makes sampling decision based

on a pre-defined sampling rate; and (3) *Adaptive sampling*, which dynamically adjusts the sampling decisions during runtime.

- *Tail-based sampling* makes the sampling decision at the end of each trace. Compared to head-based sampling, it can make a more informed decision after all the collected traces are available. Hence, developers can pay more attention to the traces that may contain anomalies and discard the repetitive normal traces. However, tail-based sampling is not supported by many LUs due to its high resource requirements on memory/disks for temporarily storing all the generated traces.
- *Unitary sampling* makes the sampling decision at every trace point. Hence, a complete trace cannot be recovered through this approach. This technique only has very limited usage scenarios.

The sampling-based solution is applied in the `Distributed Tracing` sub-step in the Logging Approach step and `how-to-configure` sub-step in the LU integration step. It is a widely adopted solution in practice to reduce the performance overhead in distributed tracing. The downside of the sampling-based solution is that important logs might be filtered out due to low sampling rate. However, in a high-throughput SUS, crucial events will be caught eventually. Low-throughput SUS are recommended to preserve every trace if the system is not tolerant of missing information [120].

- *Cost-optimization-based solution*: Three cost-optimization-based solutions have been proposed to determine the optimal instrumentation points in the SUS under a certain threshold:
  - *Information Theory*: To better recover execution paths using log messages, Zhao et al. [131, 132] propose *Log20*, which is a tool to automatically insert LC snippets. To evaluate such capability, the concept of entropy is used. Entropy is originated from Shannon's information theory. In the context of problem diagnosis, the higher entropy, the more uncertain execution paths exist in a code block. At the same time, the performance costs of the instrumented LC snippets should not exceed a customized threshold to minimize performance overhead. The best logging points are those that resolve the most uncertainty during problem diagnosis within an acceptable range of performance overhead.
  - *Constraint Solving*: To dynamically control the performance overhead, Ding et al. [54] propose a constraint solving method to determine the optimal logging points that incur minimum performance overhead with maximum amount of runtime information. This approach provides a configuration that dynamically adjusts the types of log messages outputted during runtime based on the performance of SUS.
  - *Statistical Modeling*: To better monitor the performance of SUS, Yao et al. [126] propose Log4Perf to suggest logging points. They first build performance models by running performance tests. Through these models, source code snippets that are performance-influencing are identified. For all the methods in the source code, the entry points and the exit points are instrumented with LC snippets. After re-executing the performance tests, the methods that cost constant execution time are identified and their corresponding LC snippets are removed. The remaining set of LC snippets will assist developers in diagnosing and optimizing system performance by increasing the visibility of performance issues.

Cost-optimization-based solutions are applied in the `where-to-log` sub-step in the LC Composition step. They are able to reduce performance overhead caused by log instrumentation while preserving their ability for tasks such as code path recovery and performance analysis. However, logs are used in many other scenarios (e.g., security compliance). It is important to expand these mutli-objective optimization problems to other factors of logging.

4.2 Diagnosability

Diagnosability is to what degree practitioners can leverage log instrumentation to diagnose system problems, functional or non-functional. More than often, logs are the only source that practitioners rely on for this task. However, around one-third of log instrumentation is conducted in an ad hoc manner [42, 128], which implies that practitioners often add log instrumentation after they realize there is a lack of information. This will cause delay in diagnosis and potential financial losses. Hence, in this subsection, we will describe two challenges in diagnosability (failure diagnosis and performance analysis) and the corresponding solutions to address them.

4.2.1 Failure Diagnosis. Practitioners often leverage logs for failure diagnosis. They often start with mapping logs to the LC snippets and work backwards to understand the executed code paths during runtime. However, if the recorded information in logs is not enough for pinpointing the exact executed code paths, a number of possible paths need to be identified and analyzed thoroughly to infer the real paths. This will dramatically increase the analysis time and is harmful to the product quality. Hence, proactive log instrumentation techniques are needed to identify the key program points that need to be instrumented. The following two solutions have been proposed to address the challenge of failure diagnosis:

- *Rule-generation-based solution*: Cinque et al. [49] find that the quality of generated telemetry data has a strong correlation with the ability to report failure, which highlights the importance of log quality. Motivated by the findings, they [51, 52] further summarize eight general rules for log instrumentation by studying system design artifacts such as architectural models and UML diagrams. To do this, they first abstract these artifacts into a system representation model, which consists of the interactions among a set of entities in the system. Then the rules for instrumenting LC snippets are composed to record the key interactions, which should cover the needed data for failure diagnosis. These rules are designed to meet the needs of diagnosing four types of functional failures: (1) service error, (2) service complaint, (3) interaction error, and (4) crash error. For instance, when a service is unable to reach an exit point, it is considered to be a service error. To handle such errors, the events of service start and service end must be logged. Crash errors refer to any abnormal stop of an entity. To handle such errors, a heartbeat LC snippet needs to be periodically invoked to monitor the service execution.

  Baccanico et al. [28] find several logging issues existing in Selex ES, a top-leading Finmeccanica company. They propose to use logging policies, i.e., a set of rules, to support the log reengineering across all the product teams. For example, they propose that an error-reporting logging statement must be inserted after checking an assertion. They aim to support the failure diagnosis by enhancing the log instrumentation via rule-based approach.

  Varvaressos et al. [123] propose a rule-based framework for detecting bugs in video games. They improve the traditional instrumentation process by identifying the key properties of video games. That is, every video game has a main game loop and it is thus the only place that needs to be instrumented. Key events that represent the program states can be captured. They instrument five games using the same rule: identifying the game loop, instrumenting the template instructions, and checking if the properties (represented as a series of game status) are violated. This log instrumentation framework is able to detect existing bugs of the studied video games precisely.

  The Rule-generation-based solution is applied in the `Rule-based Logging` sub-step in the Logging Approach step. It helps to complement the existing log instrumentation in the SUS by automatically inserting log instrumentations according to the generated rules.

The overhead of additional instrumentation is also controlled at an acceptable level. However, all of the above approaches derive rules manually and require deep domain knowledge. Hence, an interesting future area of research is to investigate automated approaches to discover and update rules based on the SUS's runtime behavior and historical problems.

- *Program analysis-based solution*: Program analysis [106] is a technique that analyzes the behavior of SUS automatically through static [33] or dynamic analysis [105]. Both types of techniques have been used to facilitate the diagnosis of functional failures:
  - *Static analysis*-based techniques analyze source code without executing the SUS. There are many program analysis tools, which automatically scan through the source code of SUS, output abstract representations like **ASTs (abstract syntax trees)** and call graphs, and reveal deficiencies in the source code. For example, call graphs can be analyzed further to identify logging points that are suitable for failure diagnosis. Yuan et al. have conducted two prior works to improve failure diagnosis by leveraging static analysis [127, 129]. For example, they investigate 250 bug reports and characterize exception patterns that need additional logging [127]. A static checking tool, Errlog, is proposed to scan the code base for these types of exception blocks and automatically instrument LC snippets to record the error locations and error context. They also propose LogEnhancer [127] to instrument additional variables in the existing LC snippets. These variables are extracted by statically analyzing the control flow and data flow of the source code of SUS, so log messages can contain complete runtime information to closely replay and diagnose the failure context. Different from the above approaches, in which the logging points are suggested manually one-by-one, SmartLog [73] is proposed to automatically instrument LC snippets by leveraging the data mining techniques on the static analysis results. The context (e.g., residing functions) of LC snippets are analyzed to generate log intention models. The log intention models represent the logging decisions (a.k.a. whether this LC snippet is logged or not logged) of a code snippet. Data mining models are then trained on such dataset and used subsequently to suggest program points for log instrumentation.
  - *Dynamic Analysis*-based techniques suggest logging points by analyzing the runtime behavior of SUS. Compared to static analysis-based techniques, dynamic analysis-based techniques need to execute the SUS. The output data generated by the execution is then analyzed for various tasks. The ambiguous and missing information in the outputs leads back to logging suggestions. Hence, dynamic analysis-based solution usually consists of three steps:

    (1) *Running SUS under different settings*: Developers can inject customized faults into the SUS or just run with common workloads. The goal of this step is to collect output data, such as log messages, stack traces, and memory dumps.
    (2) *Log analysis*: The goal of this step is to check whether the current output data is capable of diagnosing failures. If not, then the missing information indicates the potential logging points and what key variables need to be recorded.
    (3) *Update instrumentation*: Additional instrumentation will be performed on the SUS. Then the same experiments from step 1 will be executed again. The goal of this step is to evaluate if the newly instrumented LC can improve the failure diagnosis process.

    There are three research works that leverage dynamic analysis-based techniques to suggest additional logging points. Cinque et al. [50] propose a technique to increase the failure diagnosability of log messages. They first inject faults into three popular open source projects and execute the SUS to collect log messages and memory dumps. Analyzing the output data, they summarize the top 10 frequent executed functions from halt failures and silent failures. Additional LC snippets are inserted into these functions. Crameri et al. [53] also propose similar techniques to suggest which branches to log. They first repeatedly execute the SUS with different inputs using a symbolic engine. After each run terminates, they record the constraints between the symbolic variables and the executed branches. Additional log instrumentation is performed by identifying the associations of executed branches and variables. Jia et al. [72] propose an approach to inserting LC snippets to ease fault localization. They first run the SUS with injected faults. Then they compare the log messages between successful runs and failed runs to identify key variables to log.

Program analysis-based solutions are applied in the `where-to-log` and `what-to-log` sub-steps in the LC composition step.

Static analysis-based techniques incur low overhead, as the analysis can be run offline. Dynamic analysis-based techniques can provide more accurate results, as they are based on real execution instead of offline inference. However, with the rapid software release cycle nowadays, both approaches need to be re-run frequently to reflect the feature code changes. Hence, to prevent the results from becoming stale, tools for integrating these solutions into Continuous Integration/Continuous Delivery processes are needed.

4.2.2 Performance Analysis. Performance problems are often caused by interactions among different software components. With the increasing popularity of cloud native applications and microservice architecture, one system could contain hundreds or even thousands of small services, many of which are developed by different engineering teams. Logs generated by traditional logging approaches are insufficient for analyzing performance problems, as the lack of contextual information makes it impossible to reconstruct the ordering of events. Without such information, it is very challenging to pinpoint the performance bottleneck.

Causality tracking-based solutions are applied in the `Distributed Tracing` sub-step in the Logging Approach step and in the `where-to-log` and `what-to-log` in the LC Composition step. They are designed to instrument and preserve contextual data in logs to facilitate complex performance analysis. They are widely adopted in modern distributed software systems. There is one solution proposed to address the challenge of performance analysis:

- *Causality tracking-based solution* To enable thorough end-to-end performance analysis, the causality tracking-based solution is proposed. In particular, causality tracking is conducted in two ways [114, 119]:
  — *Schema-based techniques*: Schema-based techniques correlate the relevant logs based on pre-defined rules [29, 30, 47, 68]. Developers need to design event schemas to join individual event to reconstruct a complete request. For example, event `A` and event `B` share the same value of variable `x`, event `B` and event `C` share the same value of variable `y`, then event `A`, `B`, and `C` will be joined. The causality is then decided through the happened-before relation.
  — *Propagation-based techniques*: Propagation-based techniques track the causality within a request by propagating the context metadata between instrumented components. A complete trace of a request consists of multiple log messages (i.e., spans) linked by the metadata. The metadata contains a unique global trace ID. The metadata is passed around as the request flows from one component to another one. Apart from the global trace ID, it also records other necessary information such as parent ID to keep the causality relations between individual span. The format of metadata needs to follow a certain standard or protocol so both the senders and receivers can pack and unpack the information.

Most of the modern distributed tracing frameworks adopt propagation-based techniques instead of schema-based techniques. There are three reasons for this:

- *Performance Overhead*: Schema-based techniques do not support sampling, because they cannot decide which log messages to be discarded without compromising the ability to conduct the join operations. Hence, for SUS that generates large volumes of log messages every day, it would be too expensive to adopt schema-based techniques.
- *Generalizability*: Propagation-based techniques are general across different SUS. On the contrary, for schema-based techniques, developers need to implement the join schema for every different SUS based on its characteristics, which can be time-consuming and error-prone.
- *Real-time feedback*: Schema-based techniques are mostly conducted offline after all the log messages have been collected. For SUS that need monitoring or online analysis and detection, propagation-based techniques are more appropriate.

4.3 LC Quality

Developing and maintaining high-quality logging code is very challenging, especially in constantly evolving SUS. There are two main reasons: (1) Management: software logging is a cross-cutting concern, which tangles with the feature code. Although there are language extensions (e.g., AspectJ) to support better management of logging code, many industrial and open source projects still choose to inter-mix logging code with feature code. (2) Verification: unlike feature code or other types of cross-cutting concerns (e.g., exception handling or configuration), whose correctness can be verified via software testing; it is very challenging to verify the correctness of the logging code. In this subsection, we present three challenges that are related to LC Quality: clarity, consistency, and maintainability. For each challenge, we discuss the solutions that address it.

4.3.1 Clarity. There are two components within an LC snippet, which provides the context of logging: the static texts and the dynamic information. The static texts generally describe the surrounding of the logging context (e.g., "Transaction completed") whereas dynamic information, including variables and method invocations, provides the runtime information/state (e.g., the total amount for that transaction) of the SUS. Unclear static texts and missing dynamic information would cause confusion and even mislead developers' decisions. However, as the clarity of LC snippets usually does not impact the normal execution of the SUS, it is challenging to identify the clarity issues in the existing LC snippets. Automated approaches to improving the clarity of the LC snippets are needed to ensure the LC quality.

Hence, it is important to clearly describe the static context of each LC snippet to avoid confusion and to add the appropriate dynamic information in the LC to capture a clear runtime context. There are three solutions reported in the literature. NLP-based solution is proposed for automatically generating static text. Visualization-based and Entropy-based solutions are proposed for clarifying dynamic information.

- *NLP-based solution*: He et al. [66] conduct a study on characterizing the static texts of logging code using **Natural Language Processing (NLP)** techniques. They find that static texts in LC snippets are endemic, i.e., LC snippets within the same file or in the same context tend to use similar static texts to describe the program behavior. Inspired by this finding, they propose a solution to automatically generate static texts for an LC snippet. The solution consists of three steps: (1) for the candidate LC snippet, the surrounding code snippet (*Snippet A*) are extracted; (2) within a corpus of code snippets that contain an LC snippet, we extract the most similar code snippet *Snippet B* compared to *Snippet A*. The similarity is calculated by Levenshtein distance; (3) the static text of the LC snippet in *Snippet B* is then used as the static text of the candidate LC snippet.

The NLP-based solution is applied in the `what-to-log` sub-step in the LC Composition step. This solution is the first work on automated static text generation and can achieve decent BLEU and ROUGE score (two metrics that are commonly used in NLP studies). It does suffer from several limitations. For example, LC snippets in similar code snippets do not necessarily have the same static texts, because they are in different methods. Other techniques such as clone detection and deep learning might be helpful for this task.

- *Visualization-based solution*: Rabkin et al. [110] propose a visualization-based solution for adding missing variables of LC snippets. For each log message, they first create a graphical representation that contains all the identifiers in it (e.g., a transaction or operation ID). The identifier graph is then constructed by linking the log messages with the same identifers. As the identifiers can be missing, inconsistent, or ambiguous, the identifier graph can be used to visualize the deficiencies. For example, a log message with insufficient identifiers would cause missing edges in the graph, where the semantics of the source code shows that the edge should exist. Based on the finding, suggestions like adding the missing identifier are made to improve the clarity of the LC snippet. They evaluated this solution in three popular open source projects and demonstrate it can effectively pinpoint the deficiencies in the LC snippets. The visualization-based solution is applied in the `what-to-log` sub-step in the LC Composition step. It is straightforward and easy to apply. However, it requires manual efforts for examining the visualization and thus cannot scale to large volumes of LC snippets.
- *Entropy-based solution*: Salfner et al. [113] propose an entropy-based solution to measure the comprehensiveness and expressiveness of logs. They define a set of metrics by adapting Shannon's information entropy to the context of logs. This enables practitioners to compare the clarity of different LC snippets. They evaluate this solution on a set of real logs taken from a Java Enterprise Beans container. Results show that certain information is considered redundant and should be carefully removed. The entropy-based solution is applied in the `what-to-log` sub-step in the LC Composition step. Quantifying the clarity of logs is helpful for practitioners to make an informed decision on how to improve the LC snippets. However, automated techniques that support the improvement process for legacy LC snippets are needed.

4.3.2 Consistency. The consistency of LC snippets denotes if the location, contents, or style of LC snippets are uniform. As LC snippets usually scatter across the whole code base, it is challenging to identify and fix the existing inconsistencies. Consistent LC snippets generally have three benefits: (1) in the instrumentation phase, it helps developers to make informed decisions on *where-to-log*, *what-to-log*, and *how-to-log*; (2) in the runtime, information loss is avoided by the consistent verbosity levels; and (3) in the operation phase, consistent logs can facilitate automated analysis such as failure diagnosis and performance analysis.

There are two solutions proposed to ensure the consistency of LC snippets: machine learning-based solution and code cloning-based solution:

- *Machine learning-based solution*: As logging is pervasive in software [42, 128], it is natural to apply machine learning techniques to automatically learn from the existing practices and provide useful insights. Depending on the objectives of tasks, there are two common types of ML: supervised learning and unsupervised learning. Supervised learning requires the training data to have readily available labels and predicts the labels of new data. However, unsupervised learning does not require the training data to have labels. It is mostly used for deriving patterns in a dataset.

— *Supervised learning* techniques are adopted for assisting LC composition, as the objectives are labelling if a code snippet should be logged (`where-to-log`) or if a particular variable/verbosity level should be used for logging (`what-to-log`). The general process usually consists of the following four steps: (1) *Data gathering:* This step is to collect training data for performing the tasks. The training data consists of a set of instances with labels; (2) *Feature engineering:* This step is to extract features that are to describe the instances. These features are the input for the machine learning models; (3) *Model building:* This step is to build machine learning models from the labelled training data. The parameters are tuned in this step to improve the model performance; and (4) *Making predictions:* This step is to apply the model on new data to predict the labels. This step is the final output of the supervised learning process.

  Supervised learning techniques have been applied in the following two sub-steps in the LC composition phase:

  * *Predicting where-to-log*: Fu et al. [60] propose a technique to predict if a code snippet should be logged. In particular, they focus on two types of code snippets: `catch` blocks and `return-value-check` blocks. Each logged or unlogged code snippets are collected and labelled. Contextual keywords, such as the residing function name, are then collected as the features. A decision tree model is built for the task based on the collected features. They evaluate the solution on two Microsoft systems and achieve more than 80% of precision and recall. Zhu et al. [137] propose LogAdvisor, which improves the previous technique by collecting more types of features. Their features include structural features, which are the contextual keywords, the textual features, which are generated by transforming the code snippet into stemmed words, and the syntactic features, which are the properties of the code snippets such as the total lines of the code snippet. In addition, they also include new processes such as noise handling and cross-project evaluation. A user study shows 70% of participants think suggestions made by LogAdvisor is useful. Lal et al. propose a similar technique using a different set of features for predicting the insertion of LC snippets in two types of code blocks: `catch blocks` [89] and `if` code blocks [88]. Li et al. [91] use automatically computed topics to describe the functionality of a code snippet. This information is then used as an additional feature for an existing machine learning model, which is used to predict whether a code snippet needs to be logged. Results show that such a topic-based feature is helpful in explaining the likelihood of log instrumentation. With this feature, the value of AUC and balanced accuracy increases more than 10%. Other than code snippets, they [93] also propose a similar technique to predict if a new commit needs just-in-time changes for LC snippets.
  * *Predicting what-to-log*: Supervised learning techniques are also used to predict the modification of contents in an LC snippet. They can be further divided into the following two areas depending on the types of logging components:
    . *Verbosity levels*: Li et al. [92] propose to recommend the most appropriate verbosity level for newly added LC snippets. Since there are more than two verbosity levels for an LC snippet, they build an ordinal regression model for this task. They gather the training data from development histories and collect three types of features: file metrics, change metrics, and historical metrics. Results show that the model outperforms the baseline model. Features such as the characteristics of the containing block of a newly added LC snippet and the existing LC snippets in the containing source

code file are important in explaining the predictions. Kim et al. [83] propose to use semantic features of LC snippets for validating the consistency of verbosity levels. They first collect similar sets of features as Li et al. [92]. In addition, they apply the `word2vec` model to generate the word embedding of LC snippets as the semantic features. Then they build randomforest and KNN models as the classifiers. Results demonstrate that this technique achieves more than 85% of precision and recall.

. *Dynamic Contents*: Liu et al. [97] present a deep learning-based technique to recommend variables in LC snippets. Different from predicting a verbosity level, which has a fixed set of labels, variables in LC snippets are chosen from a dynamic set of candidates. In addition, the names of variables may not exist in the training dataset, which is known as the "out-of-vocabulary" problem. To address the first problem, they first use a neural network with an **RNN (recurrent neural network)** layer and a self-attention layer to learn the representation of each program token and predict which token should be included in the LC snippet using a binary classifier. To address the second problem. they map program tokens to word embeddings of natural language tokens. They evaluate the technique on nine open source projects and show that it outperforms the baseline methods by large margins.

— *Unsupervised learning* is applied to assist the modification of LC snippets at the step of `how-to-log`. Li et al. [94] find that LC snippets with similar contexts tend to share similar modifications. Inspired by this finding, they propose LogTracker, a clustering-based technique to automatically learn log revision rules. For each instance of LC snippet, the features are generated from the semantics of context. Then they apply the agglomerative hierarchical clustering algorithm to mine log revision rules. The log revision rules are then used to suggest modifications to LC snippets in a similar group. More than 65% of reported issues detected by LogTracker are confirmed and fixed.

Machine learning-based solutions are applied at all three sub-steps of LC composition. They can help practitioners make informed logging decisions without manual intervention. However, they generally rely on the size and the quality of the existing logging data. Hence, they may not be applicable to new or small-sized projects.

- *Code Cloning-based solution*: The idea of the code cloning-based solution is to validate the quality of LC by searching for similar LC snippets. For example, Yuan et al. [128] propose a code cloning-based technique to fix inconsistent verbosity levels in LC snippets. They first extract all the groups of code clones and identify the groups of code snippets that contain LC snippets. If within the same clone group the LC snippets have inconsistent verbosity levels, then at least one of them is considered to be incorrect. The code cloning-based solution is applied in the `how-to-log` sub-step in the LC Composition step. Although the code cloning-based solution is straightforward and easy to implement, its capability is limited. This is because not every code snippet containing LC snippets has clones [43].

4.3.3 Maintainability. The maintainability challenge is regarding the problem of supporting the evolution and maintenance of LUs and LC. As log instrumentation is a cross-cutting concern, it is challenging to keep the logging code updated with the feature code while it evolves rapidly. The instrumented LC snippets may become obsolete, insufficient, and misleading due to lack of continuous maintenance. Although many projects use general-purpose LUs, some LUs are used specifically for improving the maintainability of the LC [87]. For example, Bartsch and Harrison [32] compare SUS written in the Object-Oriented to the Aspect-Oriented manner. They conclude that the clear separation of logging concerns has a positive impact on the maintainability of LC. JBoss logging [71] supports multiple human-readable languages in the logs by providing internationalization-related API. It is quite common to find the current LUs not statisfying the constantly evolved software requirements. Hence, developers [77] migrate LUs for better flexibility, performance, and maintenance. However, over 70% of the successfully migrated projects may suffer from post-migration bugs. Existing studies show that logging is pervasive and LC snippets are often updated as after-thoughts to reflect feature code changes [40–43, 128, 130]. Most of the LC snippets have been changed at least once during their life cycle [78]. There are two types of solution reported in the existing literature to improve the maintainability:

- *Domain-specific Language-based solution*: One of the main issues associated with rule-based logging is the steep-learning curve [44]. To ease the migration efforts from conventional logging to rule-based logging, Bruntink et al. [35] and Mohammadian et al. [26] propose techniques to easily express logging concerns in a modularized manner in domain-specific languages. These concerns can be generated to AspectC [23] code and then be removed from the original code base. The Domain-specific Language-based solution is applied in the `Rule-based Logging` sub-step in the Logging Approach step. It provides utilities for automating the separation of concerns and reduces the technical difficulty of applying rule-based logging. However, this process still needs deep domain knowledge and is thus hard to be extended to other systems.
- *History-based solution*: Without careful maintenance, the number of outdated LC snippets will increase as the SUS evolve. Outdated LC snippets exist for a variety of reasons. First, as LC snippets are mostly composed manually, human errors (e.g., typos in the static texts) are unavoidable. Furthermore, as LC snippets are scattered across the entire code base and cross-cuts with the feature code, it is hard to keep track of them efficiently. As the SUS evolves rapidly, developers may forget to update the LC accordingly along with feature code. Hence, solutions to automatically maintain LC are needed. Existing works [41, 65, 95] mainly rely on the development history to guide the maintenance of LC snippets. They usually consist of the following steps:

  (1) Examine the development history, e.g., code changes and issue reports.
  (2) Characterize the code changes that are fixing logging code issues.
  (3) Identify and extract anti-patterns (a.k.a. common problems) in logging code from the previous step and implement automated tools to detect them.

  Depending on the type of studied development artifacts, there are three kinds of techniques:
  — *Commit-based*: Chen and Jiang [41] propose the first work on characterizing and detecting anti-patterns in the logging code. They have manually examined 352 pairs of independently changed logging code snippets, which are extracted from code commits in the development history, from three well-maintained open source systems. Six anti-patterns in the logging code are identified. To demonstrate the value of the findings, they have encoded these anti-patterns into a static code analysis tool, LCAnalyzer. Case studies show that LCAnalyzer has an average recall of 95% and precision of 60% and can be used to automatically detect previously unknown anti-patterns in the logging code.
  — *Source code-based*: Li et al. [95] propose DLFinder to characterize and detect duplicate logging code smells. They manually studied over 3K LC snippets that have duplicate static texts and their surrounding context from the source code of four open source projects. They identify whether the duplicate static texts are problematic based on the context. For the problematic ones, they encode the anti-patterns into static rules and develop a code checker called DLFinder. Results show that DLfinder is able to detect previously unknown duplicate logging code smells.

— *Issue report-based*: Instead of analyzing the historical commits or the source code, Hassani et al. [65] study log-related issue reports and manually summarize seven root causes of log-related issues. They implement a rule-based tool to detect four of the seven root causes. Results show that their tool can find previously unknown log-related issues.

History-based solutions are applied in the `how-to-log` sub-step in the LC Composition step. They have been used to detect issues in the logging code of open source software projects. However, as reported by Chen et al. [43], only a small portion of logging code issues can be detected by existing techniques. Research is needed to detect more types of logging code issues. Besides, history-based solutions often come with static code checkers, which generally have a high false positive rate. It is worthwhile to look into the possibility of other types of techniques (e.g., machine learning-based, dynamic analysis-based) to complement current solutions.

4.4 Security Compliance

Security compliance means that the log instrumentation should serve the purposes of safety and legal compliance. It is essential, as logs are often the sole source available for recording the user activities. In this subsection, we will describe two challenges: auditing challenge for recording security-related activities and forensics challenge for recording crime-related activities.

4.4.1 Auditing. Auditing refers to the practice of recording a serial of security-relevant events to meet various legal regulations. The auditing requirements are often described in text-based specifications. Transforming the specifications to instrumentations is not straightforward. In addition, large-scale software systems usually contain intensive log instrumentation, so it is challenging to manage the different behavior of audit log instrumentation and regular log instrumentation, as they have different requirements. There are two types of solutions that address this challenge:

- *AOP-based solution*: Some SUS (e.g., medical and aircraft control system) need correct and sound audit logs to satisfy the security requirements. They have detailed specifications on the audit logs. Tools have been developed to transform these specifications into AOP instrumentation, so the audit logs can be generated and managed easily. Mohammadian et al. [26] propose to use Spring AOP to transform specification to logging instrumentations. They devise a code rewriting algorithm and implement a prototype on OpenMRS, a popular medical record system. The AOP-based solution is applied in the `Rule-based Logging` sub-step in the Logging Approach step. It improves system modularity and reduces code tangling and scattering. It makes the auditing logs easy to maintain and monitor. However, AOP-based solution has high learning curves and the transformation process still needs high manual efforts.
- *LU Design-based solution*: Some projects (e.g., Hadoop) develop their own LUs to better support auditing. Auditing logs are generally more structured compared to the regular log messages. The format of the audit logs can vary across different software components. Hence, to facilitate code reuse, internal LUs for auditng purposes are developed. The LU Design-based solution is applied in the `what-to-adopt` sub-step in the LU Integration step. On one hand, it can be highly customized, as practitioners can manually design auditing logging utilities. On the other hand, this solution incurs additional maintenance overhead.

4.4.2 Forensics. Computer forensics is the practice of collecting data for legal purposes. This data is further exploited for the investigation of crimes. Log messages are one of the most important sources of evidence [81]. It is challenging to decide what user activities are must-logged. There is one solution proposed to address this challenge:

- *Heuristic-based solution*: King et al. [84] propose a heuristic-based solution to identify whether a user event should be logged or not from the forensic perspective. They first extract verb-object pairs from natural-language artifacts such as specifications and requirement documents. Then they propose 12 heuristic-based rules to identify the **mandatory logging events (MLEs)** from these verb-object pairs. They follow up by comparing the heuristic-based solution [85] with the other baseline methods: standards-driven and resource-driven. The standards-driven method requires the participants to identify MLEs from real-world logging specifications, such as existing healthcare and payment-card industry logging standards. The resource-driven method requires the participants to identify the sensitive resources before MLEs so they can extract MLEs from the interactions with sensitive resources. A controlled experiment is conducted on 103 computer science students. The heuristic-based solution is applied in the `what-to-log` sub-step in the LC Composition step. Unfortunately, the evaluation results show that there is no recommended method, which outperforms the other two methods at a statistically significant level. More research is needed towards identifying correct MLEs efficiently.

5 DISCUSSIONS AND FUTURE WORK

In this section, we discuss limitations in the current research and present future research directions among the three steps of log instrumentation.

5.1 Logging Approach

Among all three logging approaches, conventional logging is the most widely used approach. However, as the software release cycle increases rapidly, LC snippets instrumented by conventional logging suffers from issues such as code tangling and scattering. In addition, to enable reusability in software components and support scalability, many software systems choose to adopt microservice architecture instead of traditional monolithic architecture. It is essential to record the interactions among multiple services, as lack of such information may impact the effectiveness of many post-mortem analysis tasks.

Rule-based logging complements conventional logging by providing pre-defined instrumentation rules. As the rules are usually clearly specified (e.g., log at the beginning of every method), the instrumentation process can be easily automated. Hence, this approach can reduce manual efforts in composing LC snippets. However, previous studies [27, 104] find that the rule-based approach is not widely adopted among open source projects. It remains unclear to us why practitioners feel reluctant to adopt this approach. Further investigation is needed to address this concern.

Distributed tracing becomes very popular in recent years. Compared to conventional logging, it provides an end-to-end picture of how a request flows through multiple software services. However, to use distributed tracing utilities, extensive changes on the source code need to be done to convert LC snippets written in conventional logging. The migration process is difficult, error-prone, and time-consuming. Hence, it is worthwhile to characterize the challenges within the process and come up with best practices, guidelines, and tool support for the ease of efforts.

We propose three open problems based on the above discussion:

- *Can we identify the usage context of logging approaches?* Different logging approaches can be used in heterogeneous contexts and serve different purposes. It is important to extract and summarize the complex requirements of software logging under various contexts. For example, if the SUS adopts the microservice architecture, then identifying the causal relations between events is one of the important requirements for logging, as it helps to monitor and troubleshoot the system behavior. Under this context, distributed tracing is more appropriate than conventional logging due to its ability to propagate metadata. Gathering such documentation helps us understand the usage context of each logging approach and explore new types of logging approaches.

- *Can we understand the rationale behind adopting different logging approaches?* Many software systems might adopt more than one logging approach. To understand the benefits and drawbacks of each logging approach, we need to conduct both quantitative and qualitative studies. Quantitatively, we should propose metrics that evaluate the impact logging approaches have on the SUS, such as maintainability, configurability, and so on. Qualitatively, we need to conduct interviews or online surveys (e.g., questionnaire, open problems, Likert scale questions) with practitioners to understand why they choose certain logging approaches for their projects. The findings can then be used for providing guidance on adopting logging approaches.
- *Can we provide support on introducing new logging approaches?* The cost of introducing a new logging approach or migrating from one logging approach to another into a software system is high. Due to the lack of tool support, this process can only be done manually and is often error-prone. It would be valuable to come up with a set of metrics to measure the efforts to help practitioners to decide whether it is worthwhile to conduct such a process. In addition, tool support is needed to automate this process. For example, to transform the LC snippets using conventional logging into LC snippets using distributed tracing, we can leverage static code transformation tools such as JDT and Spoon.

5.2 LU Interaction

As suggested by our previous study [44], many software systems adopt more than one logging utility to fulfill different kinds of requirements. Practitioners need to interact with these LUs through different APIs. This will cause additional configuration and maintenance overhead. Zhi et al. [134] find that managing the configurations of one LU for constantly evolving software systems is already non-trivial and configuration bugs such as invalid loggers may occur during this process. With more LUs being used in software systems, the process would become more complex and error-prone. Hence, we propose the following three open problems:

- *How to recommend the appropriate LUs?* There are many LUs available in the wild that provide various functionalities about software logging. At the same time, new LUs are also constantly introduced. There is a lack of thorough and quantitative comparison of different LUs from various dimensions such as performance, configurability, compatibility, and so on. Such a comparison will provide guidance for practitioners on selecting the appropriate LU(s) for the SUS.
- *How to manage multiple LUs?* Large-scale software systems usually depend on numerous third-party libraries, which may contain their own LUs or leverage external LUs. It is necessary to look into all these LUs to ensure they behave as expected. There are two benefits: (1) it helps to achieve full observability of the entire system; (2) it helps to identify previously unknown issues in the interactions. Therefore, one potential opportunity is to develop techniques to automatically configure the logging behavior across multiple LUs in one SUS.
- *Can we provide guidance and summarize best practices for developing internal LUs?* Although there are many external LUs available in the wild, practitioners still implement their own LUs for the sake of customization. Few studies explore the rationale behind implementing home-grown LUs and evaluate their benefits and costs. Findings from such studies can

be further synthesized as guidelines and best practices to support the process of selecting/customizing LUs.

5.3 LC Composition

Existing studies on LC composition mostly focus on automatically suggesting the insertion of new LC snippets or update of existing LC snippets. Some of the solutions described in Section 4 have already been applied in practice. However, intelligent LC composition is still at an early stage, as these solutions are only able to find a small portion of all the issues in LC [43]. Furthermore, it is difficult to compare these techniques due to the lack of well-accepted benchmarks. Hence, we propose the following four open problems:

- *How to effectively bootstrap logging practices?* So far, many studies aiming to improve logging practices were built upon existing logging practices of the SUS. For example, machine learning models need to be trained from a large corpus of the training dataset. The features in these training datasets are extracted from existing LC snippets. However, there are many software projects with few or even no LC snippets. Bootstrapping logging instrumentation in these projects requires cross-project knowledge. One possible solution is to explore if we can leverage logging practices in mature software systems, which have a long development history, to help the starting software systems.
- *How to evaluate the effectiveness of logging practices?* Currently, there are no well-accepted standards or metrics defined in evaluating the effectiveness of logging practics. For example, in software testing field, code coverage is an important metric for evaluating the quality of test suites. Similar metrics should be developed to evaluate logging practices from different dimensions such as usability, diagnosability, and so on. Such metrics can help practitioners to understand the maturity of the current logging practices and further identify opportunities for improvement.
- *Can we provide benchmarks for improving LC composition?* Chen and Jiang [43] have made the first attempt to extract the **Logging-Code-Issue-Introducing (LCII)** changes by mining historical development data. Every LCII change corresponds to a potential logging issue. Such a dataset can be very useful for interested researchers to develop new techniques in the *how-to-log* sub-step of LC composition. However, more benchmarks are needed for evaluating solutions in the other two sub-steps of LC composition.
- *What are the characteristics of logging practices for the other two logging approaches?* Existing studies (e.g., References [42, 128]) mainly focus on the logging practices using conventional logging. Few works to date focus on characterizing logging practices using the other two logging approaches. As we previously discussed, these three logging approaches are different in many ways, including the adoption context and maintenance efforts, and so on. These differences will highly impact the way that practitioners develop and maintain the instrumented LC. Hence, such a study would greatly help us to understand and improve the step of LC Composition using the other two logging approaches.

6 CONCLUSION

Software logging is used widely by DevOps practitioners for a variety of purposes. Software logging consists of two phases: log instrumentation and log management. Unlike log management, which has extensive tool support, there is little research and practice done in the area of log instrumentation. Different from existing surveys, which focus on conventional logging [112], distributed tracing [114], and log management [37], this survey provides a comprehensive view on all three steps of log instrumentation and summarizes the associated challenges and solutions. Some of our findings are consistent with the existing surveys. Rong et al. [112] find that it is very challenging to maintain good quality LC snippets and it is hard to balance the cost and benefits of log instrumentation. Sambasivan et al. [114] find that distributed tracing is widely used for performance analysis and sampling is important in controlling the overhead. Candido et al. [37] find that there are three steps in logging code composition. However, our survey presents some unique findings that none of the existing surveys discuss. First, we survey existing studies based on the three steps in the log instrumentation phase: logging approach, LU interaction, and LC composition. This is the first survey that systematically compares three different logging approaches. Second, we find there are few discussions on interactions of multiple LUs, which could bring challenges in configurability and maintainability. Third, we identify the need to create benchmarks to effectively evaluate different LC composition techniques.

We hope this survey will be helpful to researchers, as we summarize the state-of-the-art techniques on log instrumentation and identify several future research directions. We also hope that this survey can be helpful to practitioners for their daily development and maintenance tasks. To ease replicability and further study on this survey, we have provided our dataset at Reference [18].

REFERENCES

[1] Apache Software Foundation. 2020. Apache Commons Logging. Retrieved from https://commons.apache.org/proper/commons-logging/.
[2] Azure. 2020. Azure security logging and auditing. Retrieved from https://docs.microsoft.com/en-us/azure/security/fundamentals/log-audit.
[3] AWS. 2020. Centralized Logging. Retrieved from https://aws.amazon.com/solutions/implementations/centralized-logging.
[4] Baeldung. 2020. Comparing Spring AOP and AspectJ. Retrieved from https://www.baeldung.com/spring-aop-vs-aspectj.
[5] Datadog. 2020. Datadog. Retrieved from https://datadoghq.com.
[6] Elastic. 2020. Elastic. Retrieved from https://www.elastic.co.
[7] Gartner. 2014. Gartner 2014 SIEM Magic Quadrant Leadership Report. Retrieved from http://www.gartner.com/document/2780017.
[8] Google. 2020. Google Cloud Operation. Retrieved from https://cloud.google.com/products/operations.
[9] Apache Software Foundation. 2020. HBASE-750: NPE caused by StoreFileScanner.updateReaders. Retrieved from https://issues.apache.org/jira/browse/HBASE-750/.
[10] Jaeger. 2020. Jaeger Tracing. Retrieved from http://www.jaegertracing.io.
[11] Kibana. 2020. Kibana. Retrieved from https://www.elastic.co/kibana.
[12] Lightstep. 2020. Lightstep. Retrieved from https://lightstep.com.
[13] Apache Software Foundation. 2020. Log4J. Retrieved from http://logging.apache.org/log4j/1.2.
[14] Apache Software Foundation. 2020. LOG4J 2 Apache Log4j 2. Retrieved from http://logging.apache.org/log4j/2.x.
[15] LogStash. 2020. logstash - open source log management. Retrieved from http://logstash.net/.
[16] OpenCensus. 2020. OpenCensus. Retrieved from https://opencensus.io/.
[17] Dzone. 2020. Overview of Spring Aspect Oriented Programming (AOP). Retrieved from https://dzone.com/articles/overview-of-spring-aspect-oriented-programming-aop.
[18] Boyuan Chen. 2021. The replication package. Retrieved from https://www.eecs.yorku.ca/~chenfsd/resources/survey.zip.
[19] SLF4J. 2020. Simple Logging Facade for Java (SLF4J). Retrieved from https://www.slf4j.org/.
[20] GitHub. 2020. spdlog - Very fast, header-only/compiled, C++ logging library.Retrieved from https://github.com/gabime/spdlog.
[21] Splunk. 2020. Splunk. Retrieved from http://www.splunk.com/.
[22] Sarbanes-Oxley Act. 2002. Summary of Sarbanes-Oxley Act of 2002. Retrieved from http://www.soxlaw.com/.
[23] UBC. 2020. The AspectC Project. Retrieved from https://www.cs.ubc.ca/labs/spl/projects/aspectc.html.
[24] Apache Software Foundation. 2020. The JavaDoc of Log4J 2. Retrieved from https://logging.apache.org/log4j/2.x/log4j-api/apidocs/org/
apache/logging/log4j/Level.html.
[25] Zipkin. 2020. Zipkin. Retrieved from https://zipkin.io/.

[26] Sepehr Amir-Mohammadian, Stephen Chong, and Christian Skalka. 2016. Correct audit logging: Theoryl. In *Proceedings of the 5th International Conference on Principles of Security and Trust , Held as Part of the European Joint Conferences on Theory and Practice of Software*. 139–162.

[27] Sven Apel and Don Batory. 2008. How AspectJ is used: An analysis of eleven Aspectj programs. *J. Obj. Technol.* 9, 1 (2008).

[28] Fabio Baccanico, Gabriella Carrozza, Marcello Cinque, Domenico Cotroneo, Antonio Pecchia, and Agostino Savignano. 2014. Event logging in an industrial development process: Practices and reengineering challenges. In *Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering Workshops*.

[29] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating System Design and Implementation*. 259–272.

[30] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. 2003. Magpie: Online modelling and performance-aware systems. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*. 85–90.

[31] Titus Barik, Robert DeLine, Steven M. Drucker, and Danyel Fisher. 2016. The bones of the system: A case study of logging and telemetry at Microsoft. In *Proceedings of the 38th International Conference on Software Engineering*. 92–101.

[32] Marc Bartsch and Rachel Harrison. 2008. An exploratory study of the effect of aspect-oriented programming on maintainability. *Softw. Qual. J.* 16, 1 (2008), 23–44.

[33] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*. 470–481.

[34] Lionel C. Briand, Wojciech J. Dzidek, and Yvan Labiche. 2005. Instrumenting contracts with aspect-oriented programming to increase observability and support debugging. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*. 687–690.

[35] Magiel Bruntink, Arie van Deursen, and Tom Tourwé. 2005. Isolating idiomatic crosscutting concerns. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*. 37–46.

[36] Rajkumar Buyya, Satish Narayana Srirama, Giuliano Casale, Rodrigo N. Calheiros, Yogesh Simmhan, Blesson Varghese, Erol Gelenbe, Bahman Javadi, Luis Miguel Vaquero, Marco A. S. Netto, Adel Nadjaran Toosi, Maria Alejandra Rodriguez, Ignacio Martín Llorente, Sabrina De Capitani di Vimercati, Pierangela Samarati, Dejan S. Milojicic, Carlos A. Varela, Rami Bahsoon, Marcos Dias de Assunção, Omer Rana, Wanlei Zhou, Hai Jin, Wolfgang Gentzsch, Albert Y. Zomaya, and Haiying Shen. 2019. A manifesto for future generation cloud computing: Research directions for the next decade. *ACM Comput. Surv.* 51, 5 (2019), 105:1–105:38.

[37] Jeanderson Cândido, Maurício Aniche, and Arie van Deursen. 2019. Contemporary Software Monitoring: A Systematic Literature Review. arxiv:cs.SE/1912.05878 (2019).

[38] Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. 2017. A survey of runtime monitoring instrumentation techniques. In *Proceedings of the 2nd International Workshop on Pre- and Post-deployment Verification Techniques*.

[39] Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. 2007. Whodunit: Transactional profiling for multi-tier applications. In *Proceedings of the EuroSys Conference*. 17–30.

[40] Boyuan Chen. 2019. Improving the software logging practices in DevOps. In *Proceedings of the 41st International Conference on Software Engineering*.194–197.

[41] Boyuan Chen and Zhen Ming (Jack) Jiang. 2017. Characterizing and detecting anti-patterns in the logging code. In *Proceedings of the 39th International Conference on Software Engineering*. 71–81.

[42] Boyuan Chen and Zhen Ming (Jack) Jiang. 2017. Characterizing logging practices in Java-based open source software projects—A replication study in Apache Software Foundation. *Empir. Softw. Eng.* 22, 1 (2017), 330–374.

[43] Boyuan Chen and Zhen Ming (Jack) Jiang. 2019. Extracting and studying the Logging-Code-Issue- Introducing changes in Java-based large-scale open source software systems. *Empir. Softw. Eng.* 24, 4 (2019), 2285–2322.

[44] Boyuan Chen and Zhen Ming (Jack) Jiang. 2020. Studying the use of Java logging utilities in the wild. In *Proceedings of the 42nd International Conference on Software Engineering.*

[45] Boyuan Chen, Jian Song, Peng Xu, Xing Hu, and Zhen Ming (Jack) Jiang. 2018. An automated approach to estimating code coverage measures via execution logs. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 305–316.

[46] Mike Y. Chen, Anthony J. Accardi, Emre Kiciman, David A. Patterson, Armando Fox, and Eric A. Brewer. 2004. Path-based failure and evolution management. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*. 309–322.

[47] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric A. Brewer. 2002. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the International Conference on Dependable Systems and Networks*. 595–604.

[48] Anton Chuvakin, Kevin Schmidt, and Chris Phillips. 2013. *Logging and Log Management.* Syngress.

[49] Marcello Cinque, Domenico Cotroneo, Raffaele Della Corte, and Antonio Pecchia. 2014. Assessing direct monitoring techniques to analyze failures of critical industrial systems. In *Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering.* 212–222.

[50] Marcello Cinque, Domenico Cotroneo, Roberto Natella, and Antonio Pecchia. 2010. Assessing and improving the effectiveness of logs for the analysis of software faults. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks.* 457–466.

[51] Marcello Cinque, Domenico Cotroneo, and Antonio Pecchia. 2009. A logging approach for effective dependability evaluation of complex systems. In *Proceedings of the 2nd International Conference on Dependability.* 105–110.

[52] Marcello Cinque, Domenico Cotroneo, and Antonio Pecchia. 2013. Event logs for the analysis of software failures: A rule-based approach. *IEEE Trans. Softw. Eng.* 39, 6 (2013), 806–821.

[53] Olivier Crameri, Ricardo Bianchini, and Willy Zwaenepoel. 2011. Striking a new balance between program instrumentation and debugging time. In *Proceedings of the 6th European Conference on Computer Systems.* 199–214.

[54] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. 2015. Log2: A cost-aware logging mechanism for performance diagnosis. In *Proceedings of the USENIX Annual Technical Conference.* 139–150.

[55] Yuan Ding, Mai Haohui, Xiong Weiwei, Tan Lin, Zhou Yuanyuan, and Pasupathy Shankar. 2010. SherLog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems.*

[56] Rodrigo Fonseca, Prabal Dutta, Philip Levis, and Ion Stoica. 2008. Quanto: Tracking energy in networked embedded systems. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation.* 323–338.

[57] Rodrigo Fonseca, Michael J. Freedman, and George Porter. 2010. Experiences with tracing causality in networked services. In *Proceedings of the Internet Network Management Workshop/Workshop on Research on Enterprise Networking.*

[58] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. 2007. X-Trace: A pervasive network tracing framework. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation.*

[59] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc.

[60] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where do developers log? An empirical study on logging practices in industry. In *Proceedings of the 36th International Conference on Software Engineering.* 24–33.

[61] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc.

[62] Mohamad Gebai and Michel R. Dagenais. 2018. Survey and analysis of kernel and userspace tracers on Linux: Design, implementation, and overhead. *ACM Comput. Surv.* 51, 2 (2018), 26:1–26:33.

[63] Marco Grochowski, Stefan Kowalewski, Melanie Buchsbaum, and Christian Brecher. 2019. Applying runtime monitoring to the industrial Internet of Things. In *Proceedings of the 24th IEEE International Conference on Emerging Technologies and Factory Automation.*

[64] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, David A. Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. 2015. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication.* 139–152.

[65] Mehran Hassani, Weiyi Shang, Emad Shihab, and Nikolaos Tsantalis. 2018. Studying and detecting log-related issues. *Empir. Softw. Eng.* 23, 6 (2018), 3248–3280.

[66] Pinjia He, Zhuangbin Chen, Shilin He, and Michael R. Lyu. 2018. Characterizing the natural language descriptions in software logging statements. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.* 178–189.

[67] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R. Lyu. 2016. An evaluation study on log parsing and its use in log mining. In *Proceedings of the 46th IEEE/IFIP International Conference on Dependable Systems and Networks.* 654–661.

[68] Joseph L. Hellerstein, Mark M. Maccabee, W. Nathaniel Mills III, and John Turek. 1999. ETE: A customizable approach to measuring end-to-end response times and their components in distributed systems. In *Proceedings of the 19th International Conference on Distributed Computing Systems.* 152–162.

[69] Nicolas Hili, Mojtaba Bagherzadeh, Karim Jahed, and Juergen Dingel. 2020. A model-based architecture for interactive run-time monitoring. *Softw. Syst. Model.* 19, 4 (2020).

[70] Ferenc Horváth, Tamás Gergely, Árpád Beszédes, Dávid Tengeri, Gergő Balogh, and Tibor Gyimóthy. 2017. Code coverage differences of Java bytecode and source code instrumentation tools. *Softw. Qual. J.* (Dec. 2017).

[71] JBoss Logging. 2019. Retrieved from https://developer.jboss.org/wiki/JBossLoggingTooling.

[72] Tong Jia, Ying Li, Chengbo Zhang, Wensheng Xia, Jie Jiang, and Yuhong Liu. 2018. Machine deserves better logging: A log enhancement approach for automatic fault diagnosis. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops*. 106–111.

[73] Zhouyang Jia, Shanshan Li, Xiaodong Liu, Xiangke Liao, and Yunhuai Liu. 2018. SMARTLOG: Place error log statement by deep understanding of log intention. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*. 61–71.

[74] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. 2008. Automatic identification of load testing problems. In *Proceedings of the 24th IEEE International Conference on Software Maintenance*. 307–316.

[75] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. 2009. Automated performance analysis of load tests. In *Proceedings of the 25th IEEE International Conference on Software Maintenance.*

[76] Zhen Ming (Jack) Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. 2008. An automated approach for abstracting execution logs to execution events. *J. Softw. Maint.* 20, 4 (2008), 249–267.

[77] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E. Hassan. 2016. Logging library migrations: A case study for the Apache software foundation projects. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 154–164.

[78] Suhas Kabinna, Weiyi Shang, Cor-Paul Bezemer, and Ahmed E. Hassan. 2016. Examining the stability of logging statements. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*. 326–337.

[79] Jonathan Kaldor, Jonathan Mace, Michal Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 34–50.

[80] Michael James Katchabaw, Stephen L. Howard, Hanan Lutfi Lutfiyya, Andrew D. Marshall, and Michael Anthony Bauer. 1997. Making distributed applications manageable through instrumentation. In *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*. 84–94.

[81] Joakim Kävrestad. 2018. *Fundamentals of Digital Forensics—Theory, Methods, and Real-life Applications*. Springer.

[82] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. *Aspect-oriented Programming*. Springer Berlin Heidelberg.

[83] Tae-young Kim, Suntae Kim, Cheol-Jung Yoo, Soohwan Cho, and Sooyong Park. 2018. An automatic approach to validating log levels in Java. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference*. 623–627.

[84] Jason King, Rahul Pandita, and Laurie A. Williams. 2015. Enabling forensics by proposing heuristics to identify mandatory log events. In *Proceedings of the Symposium and Bootcamp on the Science of Security*. 6:1–6:11.

[85] Jason Tyler King, Jonathan Stallings, Maria Riaz, and Laurie Williams. 2017. To log, or not to log: Using heuristics to identify mandatory log events—A controlled experiment. *Empir. Softw. Eng.* 22, 5 (2017), 2684–2717.

[86] Barbara Kitchenham and Stuart Charters. 2007. *Guidelines for Performing Systematic Literature Reviews in Software Engineering*. EBSE Technical Report. Keele University and Durham University Joint Report.

[87] Kimmo Kiviluoma, Johannes Koskinen, and Tommi Mikkonen. 2006. Run-time monitoring of architecturally significant behaviors using behavioral profiles and aspects. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis.*

[88] Sangeeta Lal, Neetu Sardana, and Ashish Sureka. 2016. LogOptPlus: Learning to optimize logging in catch and if programming constructs. In *Proceedings of the 40th IEEE Computer Software and Applications Conference*. 215–220.

[89] Sangeeta Lal and Ashish Sureka. 2016. LogOpt: Static feature extraction from source code for automated catch block logging prediction. In *Proceedings of the 9th India Software Engineering Conference*. 151–155.

[90] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. 2011. Large-scale, AST-based API-usage analysis of open-source Java projects. In *Proceedings of the ACM Symposium on Applied Computing*. 1317–1324.

[91] Heng Li, Tse-Hsun (Peter) Chen, Weiyi Shang, and Ahmed E. Hassan. 2018. Studying software logging using topic models. *Empir. Softw. Eng.* 23, 5 (2018), 2655–2694.

[92] Heng Li, Weiyi Shang, and Ahmed E. Hassan. 2017. Which log level should developers choose for a new logging statement? *Empir. Softw. Eng.* 22, 4 (2017), 1684–1716.

[93] Heng Li, Weiyi Shang, Ying Zou, and Ahmed E. Hassan. 2017. Towards just-in-time suggestions for log changes. *Empir. Softw. Eng.* 22, 4 (2017), 1831–1865.

[94] Shanshan Li, Xu Niu, Zhouyang Jia, Xiang-Ke Liao, Ji Wang, and Tao Li. 2019. Guiding log revisions by learning from software evolution history. *Empir. Softw. Eng.* 25, 3 (09 2019).

[95] Zhenhao Li, Tse-Hsun (Peter) Chen, Jinqiu Yang, and Weiyi Shang. 2019. DLFinder: Characterizing and detecting duplicate logging code smells. In *Proceedings of the 41st International Conference on Software Engineering*. 152–163.

[96] Jimmy J. Lin and Dmitriy V. Ryaboy. 2012. Scaling big data mining infrastructure: The Twitter experience. *SIGKDD Explor.* 14, 2 (2012), 6–19.

[97] Zhongxin Liu, Xin Xia, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2019. Which variables should I log? *IEEE Trans. Softw. Eng.* (2019).

[98] Di Ma and Gene Tsudik. 2009. A new approach to secure logging. *Trans. Softw.* 5, 1 (2009), 2:1–2:21.

[99] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles.* 378–393.

[100] Mark Marron. 2018. Log++ logging for a cloud-native world. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages.* 25–36.

[101] George Mathew and Tim Menzies. 2018. Software engineering's top topics, trends, and researchers. *IEEE Softw.* 35, 5 (2018).

[102] Jhonny Mertz and Ingrid Nunes. 2019. On the practical feasibility of software monitoring: A framework for low-impact execution tracing. In *Proceedings of the 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems.* 169–180.

[103] Daniel Le Métayer, Eduardo Mazza, and Marie-Laure Potet. 2010. Designing log architectures for legal evidence. In *Proceedings of the 8th IEEE International Conference on Software Engineering and Formal Methods.* 156–165.

[104] Freddy Munoz, Benoit Baudry, Romain Delamare, and Yves Le Traon. 2013. Usage and testability of AOP: An empirical study of AspectJ. *Inf. Softw. Technol.* 55, 2 (2013).

[105] Nicholas Nethercote. 2004. *Dynamic Binary Analysis and Instrumentation.* PhD thesis. University of Cambridge, United Kingdom.

[106] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2010. *Principles of Program Analysis.* Springer Publishing Company, Incorporated.

[107] Adam Oliner, Archana Ganapathi, and Wei Xu. 2012. Advances and challenges in log analysis. *Commun. ACM* 55, 2 (2012).

[108] Opentracing: Vendor-neutral APIs and instrumentation for distributed tracing. 2019. Retrieved from https://opentracing.io/.

[109] Antonio Pecchia, Marcello Cinque, Gabriella Carrozza, and Domenico Cotroneo. 2015. Industry practices and event logging: Assessment of a critical software development process. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering.* 169–178.

[110] Ariel Rabkin, Wei Xu, Avani Wildani, Armando Fox, David A. Patterson, and Randy H. Katz. 2010. A graphical representation for identifier structure in logs. In *Proceedings of the Workshop on Managing Systems via Log Analysis and Machine Learning Techniques.*

[111] Patrick Reynolds, Charles Edwin Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. 2006. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation.*

[112] Guoping Rong, Qiuping Zhang, Xinbei Liu, and Shenghui Gu. 2017. A systematic review of logging practice in software engineering. In *Proceedings of the 24th Asia-Pacific Software Engineering Conference.* 534–539.

[113] Felix Salfner, Steffen Tschirpke, and Miroslaw Malek. 2004. Comprehensive logfiles for autonomic systems. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium.*

[114] Raja R. Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H. Sigelman, Rodrigo Fonseca, and Gregory R. Ganger. 2016. Principled workflow-centric tracing of distributed systems. In *Proceedings of the 7th ACM Symposium on Cloud Computing.* 401–414.

[115] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. 2011. Diagnosing performance changes by comparing request flows. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation.*

[116] Mohammed Sayagh, Noureddine Kerzazi, Bram Adams, and Fabio Petrillo. 2018. Software configuration engineering in practice: Interviews, survey, and systematic literature review. *IEEE Trans. Softw. Eng.* 46, 6 (2018).

[117] Weiyi Shang, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Michael W. Godfrey, Mohamed Nasser, and Parminder Flora. 2014. An exploratory study of the evolution of communicated information about the execution of large software systems. *J. Softw: Evol. Proc.* 26, 1 (2014).

[118] Weiyi Shang, Zhen Ming Jiang, Hadi Hemmati, Bram Adams, Ahmed E. Hassan, and Patrick Martin. 2013. Assisting developers of big data analytics applications when deploying on Hadoop clouds. In *Proceedings of the 35th International Conference on Software Engineering.* 402–411.

[119] Yuri Shkuro. 2019. *Mastering Distributed Tracing: Analyzing Performance in Microservices and Complex Systems.* Packt Publishing Ltd.

[120] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.* Technical Report. Google, Inc. Retrieved from https://research.google.com/archive/papers/dapper-2010-1.pdf.

[121] Cindy Sridharan. 2018. *Distributed Systems Observability.* O'Reilly Media, Inc.

[122] Eno Thereska, Brandon Salmon, John D. Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio López Hernandez, and Gregory R. Ganger. 2006. Stardust: Tracking activity in a distributed storage system. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*. 3–14.

[123] Simon Varvaressos, Kim Lavoie, Alexandre Blondin Massé, Sébastien Gaboury, and Sylvain Hallé. 2014. Automated bug finding in video games: A case study for runtime monitoring. In *Proceedings of the 7th IEEE International Conference on Software Testing, Verification and Validation*. 143–152.

[124] Wei Xu. 2010. *System Problem Detection by Mining Console Logs*. Ph.D. Dissertation. University of California, Berkeley, CA. Retrieved from http://www.escholarship.org/uc/item/6jx4w194.

[125] Stephen Yang, Seo Jin Park, and John K. Ousterhout. 2018. NanoLog: A nanosecond scale logging system. In *Proceedings of the USENIX Annual Technical Conference*. 335–350.

[126] Kundi Yao, Guilherme B. de Pádua, Weiyi Shang, Steve Sporea, Andrei Toma, and Sarah Sajedi. 2018. Log4Perf: Suggesting logging locations for web-based systems' performance monitoring. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*.

[127] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*. 293–306.

[128] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing logging practices in open-source software. In *Proceedings of the 34th International Conference on Software Engineering*. 102–112.

[129] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2011. Improving software diagnosability via log enhancement. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–14.

[130] Yi Zeng, Jinfu Chen, Weiyi Shang, and Tse-Hsun Chen. 2019. Studying the characteristics of logging practices in mobile apps: A case study on F-Droid. *Empir. Softw. Eng.* (02 2019). DOI: https://doi.org/10.1007/s10664-019-09687-9

[131] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. The game of twenty questions: Do you know where to log? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. 125–131.

[132] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 565–581.

[133] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. 2014. lprof: A non-intrusive request flow profiler for distributed systems. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*. 629–644.

[134] Chen Zhi, Jianwei Yin, Shuiguang Deng, Maoxin Ye, Min Fu, and Tao Xie. 2019. An exploratory study of logging configuration practice in Java. In *Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution*.

[135] Jingwen Zhou, Zhenbang Chen, Haibo Mi, and Ji Wang. 2014. MTracer: A trace-oriented monitoring framework for medium-scale distributed systems. In *Proceedings of the 8th IEEE International Symposium on Service Oriented System Engineering*. 266–271.

[136] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Trans. Softw. Eng.* 47, 2 (2018).

[137] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2015. Learning to log: Helping developers make informed logging decisions. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering*.

[138] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R. Lyu. 2019. Tools and benchmarks for automated log parsing. In *Proceedings of the 41st International Conference on Software Engineering*. 121–130.

Received June 2020; revised December 2020; accepted January 2021