Drill: Log-based Anomaly Detection for Large-scale Storage Systems Using Source Code Analysis

Di Zhang[1], Chris Egersdoerfer[1], Tabassum Mahmud[2], Mai Zheng[2], and Dong Dai[1]

[1]Computer Science Department, University of North Carolina at Charlotte, {dzhang16, cegersdo, ddai}@uncc.edu
[2]Department of Electrical and Computer Engineering, Iowa State University, {tmahmud, mai}@iastate.edu

Abstract—**Large-scale storage systems, a critical part of modern computing systems, are subject to various runtime bugs, failures, and anomalies in production. Identifying their anomalies at runtime is thus critical for users and administrators. Since runtime logs record the important status of the systems, log-based anomaly detection has been studied extensively for timely identifying system malfunctions. However, existing log-based anomaly detection solutions share common limitations in representing log entries accurately and robustly, hence can not effectively handle log entries that were not seen in the historical logs, which is a common real-world scenario due to logs' inherent rarity and the continuous evolution of the systems. To address the issues of existing methods, we propose Drill, a new log pre-processing method to generate high-quality vector representation of runtime logs by leveraging both storage system-specific sentiment-classifying language models and log contexts built from the source code. Through extensive evaluations of two representative distributed storage systems (Apache HDFS and Lustre), we show that Drill can achieve up to 41% improvement when compared with state-of-the-art anomaly detection solutions, showing it is a promising solution for general anomaly detection.**

I. Introduction

Large-scale storage systems are a critical part of modern computing infrastructure in both Cloud and High-Performance Computing (HPC) environments [1], [2]. Due to increasing scale and complexity, they are subject to various bugs, failures and anomalies in production, which lead to data loss, service outages and degradation of the quality of service [3], [4], [5], [6]. It is thereby critical for anomaly detection mechanisms deployed to accurately and swiftly detect malfunctions, so that system operators can pinpoint the issues and resolve them promptly to mitigate losses.

The runtime logs, which record the internal status of storage systems, such as values of variables, function return values, and performance statistics, are considered valuable sources for detecting potential system anomalies. As a result, an extensive amount of research on log-based anomaly detection tools has been conducted recently [4], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19].

Log-based anomaly detection frameworks essentially involve four key components: *log collection*, *log pre-processing*, *pattern learning*, and the actual *online detection* [20]. The runtime logs, which are generated by *log statements* in the source code (using `printf` or logging libraries such as Log4J [21], [22]), are collected periodically. For the collected runtime logs, log pre-processing is applied to convert their free-form texts into a more structured representation, such as id or vector-based form [23], [24]. The pre-processing may further group logs using different grouping techniques, such as fixed windows, sliding windows, or session-based windows [17], [18], [25] to form a sequence of closely related events for the downstream machine learning models to learn the patterns. Extensive manual efforts are often needed in this stage to label each sequence of logs as normal or abnormal, for training the supervised machine learning models. After training, the model will be applied to future runtime logs in real-time to conduct the actual online anomaly detection. As described here, among these four components, *log pre-processing* turns chunks of runtime logs into sequences of log representations for later pattern learning, hence this step is critical for the accuracy of the downstream machine learning models and the performance of overall anomaly detection. How to generate accurate log representations effectively has been studied extensively.

The ID-based representation was initially widely used due to its simplicity. We can simply map each runtime log back to its corresponding log statement in the source code and assign a unique ID accordingly. However, this may generate unnecessary IDs since many log statements are similar and can be grouped. Researchers then leveraged the similarity of text characters in logs to generate a more concise ID representation [23], [24]. For instance, Spell used the longest common sequence to calculate the similarity of different log texts [24]. But these methods still share the same drawback of ID-based representation: similar to the one-hot representation of words in natural language processing, ID can not represent the similarity between different logs. This creates issues when a log ID does not exist in the training data but shows up later in the runtime logs. The previously trained models can not make accurate decisions on these unseen IDs.

Hence, vector-based representation has become popular recently. Different pre-trained natural language (NLP) models, such as BERT [26], have been applied to the raw runtime logs to understand their semantics and generate vectorized representation for each log. Because of their enhanced ability to generalize, the vector representation-based approaches lead to better training accuracy in downstream models, as well as better performance in actual anomaly detection [15]. For instance, NeuralLog uses the BERT model to encode logs and

1530-2075/23/$31.00 ©2023 IEEE
DOI 10.1109/IPDPS54959.2023.00028

achieves state-of-the-art accuracy in anomaly detection [27].

However, there are still fundamental issues with existing vector-based pre-processing methods. First, the pre-trained natural language models (e.g., GLoVe [28], BERT [26]) used in the existing studies are based on general human language. Although log texts are also human-readable, they are written differently and work in different contexts. Similar words in human language may have different meanings in log texts. This impacts the accuracy of the generated vector-based representations and the performance of the trained anomaly detection models (as shown in our evaluations later). Second, the issues created by unseen logs, although lightened, still exist. In the real world, runtime logs observed in a given period are often a small portion of the logs which could be triggered. For instance, in HDFS v0.17, there are in total 956 distinct log statements in the source code. However, the widely used HDFS log trace collected from a 200-node cluster only covers 29 unique runtime logs [17]. Hence, it is challenging to train high-quality downstream models by observing only such a limited variety of runtime logs. The generated models still fall short at handling unseen runtime logs (shown in later evaluations). Last but not least, existing studies mainly focus on understanding the contents of the runtime logs using NLP models but neglect the contexts of a runtime log. Since logs are created by developers to indicate the important status of the software, context information in the source code, such as whether or not a given log is in an *if* block, is highly relevant to the purpose of the log and may accurately indicate the anomalies. An effective log-based anomaly detection framework should take the context into consideration.

In this study, we propose and implement Drill[1] to address the aforementioned challenges. Drill essentially is a new log pre-processing method to generate high-quality vector representations of runtime logs by leveraging both storage system-specific sentiment language models and log contexts in the source code. Instead of naively applying existing NLP language models on raw logs, Drill leverages the source code of multiple large-scale distributed storage systems to re-train a more specific and robust '*sentiment*' language model, which hints whether a runtime log entry is likely to be an anomaly or not. We further conduct static analysis of the source code to collect context-relevant features. By combining both of these features, Drill generates accurate vector representations and uses them to form sequences of logs for training a downstream model, which is a bidirectional Long Short Term Memory neural network, Bi-LSTM [29], [30]. Through extensive evaluations, we show that Drill can achieve better performance compared with state-of-the-art log-based anomaly detection methods on representative storage systems in both Cloud (HDFS [1]) and HPC environments (Lustre [2]). We also demonstrate how Drill performs when unseen logs continuously arrive, simulating a real-world scenario, and show its advantages. The contributions of this work are threefold:

- Different from previous work which considers only log contents or sentiment, we combine the content-relevant sentiment features and context-based features to better represent runtime logs to achieve more accurate anomaly detection, especially for cases with unseen logs.
- We propose to apply static program analysis on the source code of storage system to extract accurate context features for building accurate vector representations of logs.
- We conduct extensive evaluations to show the effectiveness and performance of Drill in a variety of scenarios.

[1]https://github.com/DIR-LAB/DRILL

The remainder of this paper is organized as follows: In §II we introduce the background about log-based anomaly detection and discuss the key observations to motivate Drill. In §III we present the overall design of Drill. We present the main results in §IV. We compare with related work in §V, and conclude this paper and discuss the future work in §VI.

II. Design Motivation

The runtime logs of large-scale storage systems play an important role in understanding the runtime status of systems. In Figure 1, we show a snippet of runtime logs generated while running Lustre. As the example shows, each log is a single line of free-form text containing multiple sections of information, such as the runtime and content information. The runtime information may include `timestamp`, `PID`, and `thread Id`, which are often generated automatically. The content information mostly originates from the developers, and generally consists of a constant part, such as `"connect to ..."`, which corresponds to the texts written in the source code, and a variable part, such as `10.0.0.8` and `4296114409`, which correspond to placeholders (e.g., `%d, %s`) in the source code. The log levels, such as *debug*, *info*, *warn*, and *error*, are often contained in the logs to denote the intention of the log, but these are not necessarily accurate for detecting anomalies due to the complexity of the runtime systems. This is also why advanced log-based anomaly detection tools are needed.

```
00000004:00080000:0.0:1607450691.765123:0:3263:0:(osp_object.c:1517:o
sp_create()) lustre-OST0002-osc-MDT0000: Wrote last used FID:
[0x100020000:0x316f:0x0], index 2: 0

00002000:00080000:0.0:1607317382.389082:0:6295:0:(ofd_dev.c:1752:ofd_
create_hdl()) lustre-OST0002: reserve 8 objects in group 0x0 at 10242

00080000:00020000:0.0:1607317006.208902:0:5057:0:(osd_handler.c:1588:
osd_trans_commit_cb()) transaction @0xffff9676bad55900 commit error:
2

00000004:00080000:0.0:1607448546.792679:0:2697:0:(osp_precreate.c:684
:osp_precreate_send()) lustre-OST0000-osc-MDT0000: current precreated
pool: [0x100000000:0x7e24:0x0]-[0x100000000:0x7ee1:0x0]
```

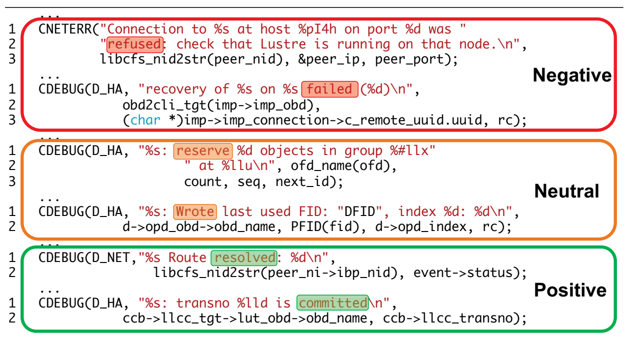
Fig. 1: A runtime log example from Lustre.

Terminology In this study, we use some specific terminology to refer to specific parts of the logging system for simplicity. We use *log statement* to denote the source code written by developers to produce the logs; *runtime log* to refer to the collected logs at runtime; *log template* to refer to the constant parts of the log content, such as `''*: connect to NID *@tcp last attempt *''`, whereas the wildcards (*) can be replaced using variable parts such as `10.0.0.8`. The log templates can be identified by parsing the runtime logs using tools like Drain [23] or Spell [24]. Multiple log entries can be gathered together into a *session* or *sequence* of logs, which serve as the main inputs for the downstream anomaly detection models.

A. Key Observations and Design Motivations

Drill is designed based on two key observations we made from the file system source code and runtime logs. In this subsection, we discuss these observations before introducing Drill's design in the next section.

1) Storage system-specific language model: As described earlier, it is known that a simple index ID does not contain enough information for logs. Hence, many recent studies have focused on leveraging natural language models to turn each runtime log into a vector with enriched features [15], [31], [27]. For example, NeuralLog [27] uses BERT [26], which was pre-trained on general human language texts to encode logs. However, these methods are reliant upon the assumption that storage system logs can be treated identically to natural language, which may not be the case. In fact, although the log content contains human-readable texts, they are noisy, brief, and often do not strictly follow grammar rules, such as `"%s Route resolved: %d"` in Fig. 2. We argue that the language model used to vectorize logs should be storage system domain specific. However, training a new or even fine-tuning an existing language model is known to require a large amount of training data. For example, BERT was trained on 3.2 billion words [26]. In contrast, if we look at the unique lines of logs in storage systems, such as Lustre, only thousands of unique training data examples are available (detailed numbers are in Table I), far less than what would be required to meet the needs of re-training or even fine-tuning a language model.



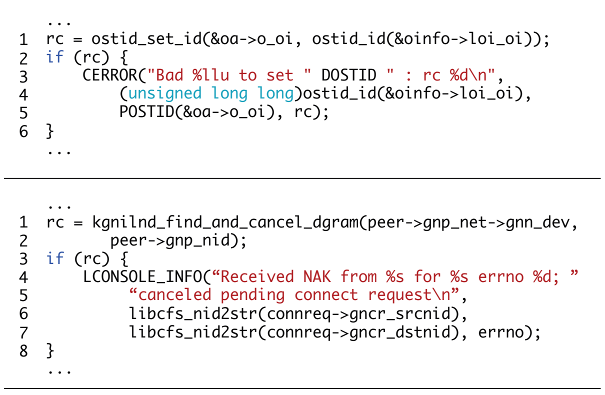
```
...
CNETERR("Connection to %s at host %pI4h on port %d was "
        "refused: check that Lustre is running on that node.\n",
        libcfs_nid2str(peer_nid), &peer_ip, peer_port);
...
CDEBUG(D_HA, "recovery of %s on %s failed (%d)\n",
           obd2cli_tgt(imp->imp_obd),
           (char *)imp->imp_connection->c_remote_uuid.uuid, rc);
...
CDEBUG(D_HA, "%s: reserve %d objects in group %#llx"
                   " at %llu\n", ofd_name(ofd),
                   count, seq, next_id);
...
CDEBUG(D_HA, "%s: Wrote last used FID: "DFID", index %d: %d\n",
           d->opd_obd->obd_name, PFID(fid), d->opd_index, rc);
...
CDEBUG(D_NET,"%s Route resolved: %d\n",
               libcfs_nid2str(peer_ni->ibp_nid), event->status);
...
CDEBUG(D_HA, "%s: transno %lld is committed\n",
           ccb->llcc_tgt->lut_obd->obd_name, ccb->llcc_transno);
```

Fig. 2: Example log statements from Lustre source code and their sentiment indicators (colored).

Not only is it difficult to train a complete storage system-specific language model, we further argue that it is actually not necessary to do so. The complete language models are useful in many NLP tasks such as *machine translation* and *question answering* because these tasks require a deep understanding of the sentences, including their word selection, grammar, tense, and so on. However, in log-based anomaly detection, we can generally disregard grammar and tense, and the key information we need is simply whether logs indicate anomalies or not. These simplifications should allow for the use of a far less complicated language model.

If we take a closer look at the log statements in the source code, as shown by Fig. 2, although logs texts are informal and noisy, they do use different tones to describe abnormal and normal system statuses, simply because code is written by developers and developers in the same community likely share common tones and vocabulary. For instance, developers typically use negative tones such as 'error' or 'exception' for anomalies and use neutral or positive tones such as 'connection successes' for normal behaviors. We believe such sentimental difference is generic across different software in the same community and can be captured via sentiment analysis. For example, in Figure 2, the first log statement expresses negative sentiment as it describes the failure of a connection. The second log statement shows a neutral sentiment as it just reports routine updates of the system. The last log statement seems to be positive as it contains words such as 'resolved' and 'committed'.

Given this example, we believe the sentiment of the texts could reflect the developers' intentions and may be a strong indicator of the system anomalies we are trying to detect. Additionally, utilizing sentiment to encode the logs can be more feasible due to lower training complexity, which makes using the limited amount of training data feasible.

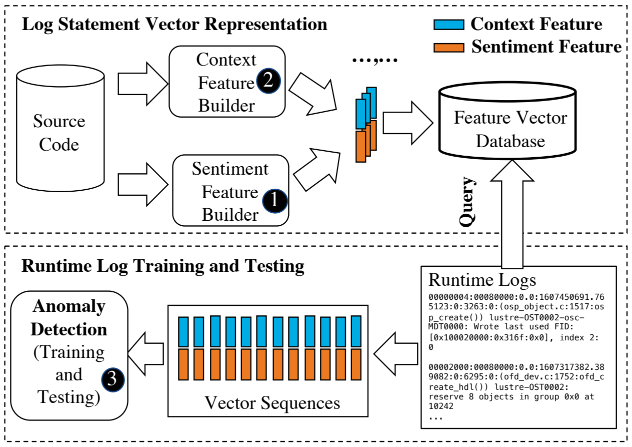


```
...
rc = ostid_set_id(&oa->o_oi, ostid_id(&oinfo->loi_oi));
if (rc) {
    CERROR("Bad %llu to set " DOSTID " : rc %d\n",
        (unsigned long long)ostid_id(&oinfo->loi_oi),
        POSTID(&oa->o_oi), rc);
}
...
```

```
...
rc = kgnilnd_find_and_cancel_dgram(peer->gnp_net->gnn_dev,
       peer->gnp_nid);
if (rc) {
    LCONSOLE_INFO("Received NAK from %s for %s errno %d; "
        "canceled pending connect request\n",
        libcfs_nid2str(connreq->gncr_srcnid),
        libcfs_nid2str(connreq->gncr_dstnid), errno);
}
...
```

Fig. 3: Context feature of Lustre log statements.

2) Context of the source code: While previous studies have focused on the contents or even the sentiment of logs for anomaly detection, we argue that they neglect the valuable information buried in the source code to denote whether an actual anomaly may be captured by the log statements. For instance, if developers check the return value of a function call and further print some logs if the return value is not expected, then such a log statement is more likely to indicate a possible anomaly. These actions (e.g., assign function calls return code to a variable, use *if-statement* to check the return code) represent the contexts of a log statement, which can be potential anomalies. For instance, Fig. 3 shows two examples of log contexts, which include both the log statement


Log Statement Vector Representation
Context Feature
Sentiment Feature
Source Code
Context Feature Builder 2
Sentiment Feature Builder 1
...,...
Feature Vector Database
Query
Runtime Log Training and Testing
Anomaly Detection (Training and Testing) 3
Vector Sequences
Runtime Logs

```
00000004:00080000:0.0:1607450691.76
5123:0:3263:0:(osp_object.c:1517:os
p_create()) lustre-OST0002-osc-
MDT0000: Wrote last used FID:
[0x100020000:0x316f:0x0], index 2:
0

00002000:00080000:0.0:1607317382.38
9082:0:6295:0:(ofd_dev.c:1752:ofd_c
reate_hdl()) lustre-OST0002:
reserve 8 objects in group 0x0 at
10242
...
```

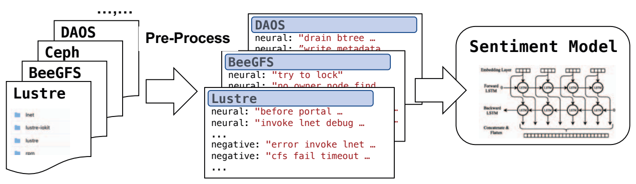
Fig. 4: The overall architecture of Drill

and its surrounding code. Here, the top log statement uses *error* log level (CERROR) to report if the previous operation ostid_set_id fails (checking rc); the bottom one uses info *log* level (LCONSOLE_INFO) to report a similarly wrong rc issue while transferring data between LNET network interfaces. Although these two log statements are in different log levels, they actually both report system anomalies in Lustre. This clearly shows that the context of log statement (i.e., after 'if (rc)') could be a strong feature to indicate the abnormal states. On the other hand, developers may periodically print logs for recording system status. Such logs are more likely related to normal system status. Such actions, showing benign condition, manifest in the context code as a log statement in a *for-loop* or *while-loop*. Hence, the context of a log statement in the source code could be a strong feature in understanding the nature of the relevant logs, which motivates the design of Drill.

III. Drill Design and Implementation

In this section, we describe the design and implementation details of Drill. We begin by showing its overall workflow in Fig. 4. There are two fundamental components in Drill: *Log Statement Vector Representation* and *Runtime Log Training and Testing*.

First, the *Log Statement Vector Representation* component takes charge of generating vector representation of logs. As described earlier, Drill analyzes the source code to build the vector representations. More specifically, it leverages the *Sentiment Feature Builder* component (❶) to build sentiment features, and leverages *Context Feature Builder* component (❷) to build context features. More details about these two components will be explained in the later subsections. These two sets of features will be combined together to form a complete vector representation. We store the representation in the *Feature Vector Database*, so that we can quickly query the database to assign vectors for runtime logs. The *Feature Vector Database* is a key-value store. During the log statement vector generating phase, each log statement in the source code



```
DAOS
neural: "drain btree …
neural: "write metadata
BeeGFS
neural: "try to lock"
neural: "no owner node find
Lustre
neural: "before portal …
neural: "invoke lnet debug …
...
negative: "error invoke lnet …
negative: "cfs fail timeout …
...
```

Fig. 5: The workflow for sentiment model training

is processed. Its generated vector representation is stored in the database using its string-based log statement as the key. Later, when runtime logs arrive (in *Runtime Log Training and Testing* phase), we first use Drain [23] to pre-process them to retrieve their string-based log templates. We then use each log template to match the closet log statement key in the database to retrieve the vector representation. In this way, querying the database should always return a vector representation.

Second, the *Runtime Log Training and Testing* component works on actual runtime logs for both training and testing. When a runtime log arrives, Drill queries the *Feature Vector Database* for its vector representation as described earlier. For a sequence of log events, we will build a sequence of vectors, which are later fit into the same bidirectional LSTM (BiLSTM) model for both training and testing (❸). The training is done based on a set of labeled historical logs. The testing is done by applying the trained model upon current runtime logs, which may include log entries that were never seen in the training dataset. More details regarding the bidirectional LSTM models will be discussed in Section III-C.

A. Sentiment Feature Builder

In the Drill workflow, building a high quality storage system-specific language model to generate the sentiment feature is the key to obtaining accurate vector representations for the logs. Building such a model is the core task of the Sentiment Feature Builder.

One key challenge when training the sentiment model for log statements is obtaining the labeled training data. Given a log statement, we do not know the common 'tone' or 'intention' shared by the community. To address this issue, we directly use the log level of each log statement as a natural label to train the model. For example, CERROR indicates abnormal and CDEBUG indicates normal. These labels are easy to obtain from the source code, addressing the training data issue. However, intuitively there is an accuracy problem as log levels are not considered to be accurate as developers may not always be consistent in using the correct log level. For instance, BeeGFS [32] may use Log_ERR to log an event which in fact is not related to any anomaly [31].

To address this accuracy issue, we propose to train the sentiment language model using source code of multiple open-source distributed storage systems. Doing so allows us to capture the developers' general consensus in the community and to avoid bias from a particular developer or software implementation. Additionally, it allows us to train a generic model in the system logs domain, which may be applied to

TABLE I: Training dataset for sentiment model

	Log Level		Log Mechanism
	Debug	Error	
OrangeFS [33]	1058	1202	`gossip_debug`, `gossip_err`,...
Ceph [34]	15459	2726	`dout`, `derr`,...
DAOS [35]	1549	3444	`D_DEBUG`, `D_ERROR`,...
GlusterFS [36]	2460	5260	`gf_msg`, `gf_log`,...

a wide variety of storage systems. Here, we selected a set of widely used open source distributed storage systems to train the model, which are listed in Table I. We briefly list the logging mechanisms used by each system and the total number of training samples collected from each system. The `Debug` logs are labeled as neutral and `Error` logs as negative. Note that, the similar idea was cursorily examined in our previous work SentiLog [31]. The major difference here is Drill not only trains the sentiment model but also extracts the context-based features and combines both features to form more accurate representations of logs. In this way, the representations that Drill builds can be used in downstream models to achieve better anomaly detection for sequences of logs, while SentiLog only predicts the abnormality of every single log entry via the trained sentiment model.

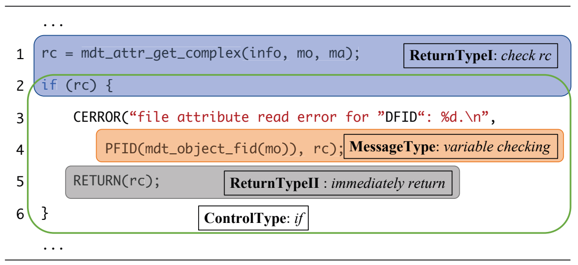
The actual training workflow is shown in Fig. 5. During training, the pre-processed logging statements from various storage systems are fed into the model trainer. We use the same bidirectional LSTM (BiLSTM) to conduct sentiment analysis. The network contains two layers, 100 neurons in each layer, and in total has 789K parameters. The model is trained in batch size 64 and using Adam as optimizer with learning rate 0.01. Since the BiLSTM network takes word vectors as the inputs, we tokenize each single word of the logging statements using the pre-trained GLoVe Embedding [28].

Drill pre-processes the raw logging statement before sending it to the BiLSTM network for training because the log statements contain information which is useless for sentiment analysis. For example, a logging statement from Lustre may look like this: ‘`Error %d invoking LNET debug log upcall %s %s;`’. The format strings (‘%s’) are clearly not useful as they will be later substituted by the actual strings. In NLP, text pre-processing is a typical step to obtain consistent training results. Drill follows similar pre-processing on the log statements [37], which includes the following steps: 1) lowercasing all the texts; 2) stemming words to their root form (e.g., invoking → invoke); 3) removing the stopwords (e.g., 'this','that','and','a','we'); 4) normalizing a text into a standard form; 5) removing noises such as the format strings (e.g., %) and punctuation. The ultimate goal of these steps is to bring the log statements closer to natural language, in order to reduce the training time and maximize the accuracy.

As opposed to training a sentiment language model, it is also theoretically possible to train a domain-specific language model or fine-tune general language model on domain-specific data. However, to obtain a decent rendition of such a model typically requires a much larger volume of training data due to the increased complexity of the task. Given the small volume of log statements in source code, it is unrealistic to train this kind of model. Instead, the simpler sentiment analysis task does not require the same volume of data, which properly fits in our scenarios. In fact, we evaluated the use of more general language models and our sentiment model in Drill, and the results show that sentiment model does in fact have better performance. More details and results can be found in §IV-B.

B. Context Feature Builder

The Context Feature Builder takes the source code of the target storage system and the log statement information as input. Its goal is to output the context feature of the log statement. The context of a log statements can be described as the surrounding code of the log statement, which provides information about the execution context of the log statement. As Figure 6 shows, we consider the context by systematically checking the code from before, after, within and around the log statement.



```
...
rc = mdt_attr_get_complex(info, mo, ma);
if (rc) {
    CERROR(“file attribute read error for ”DFID“: %d.\n”,
        PFID(mdt_object_fid(mo)), rc);
  RETURN(rc);
}
...
```

ReturnTypeI: check rc
MessageType: variable checking
ReturnTypeII : immediately return
ControlType: if

Fig. 6: Example Lustre code snippet showing log statement context.

To retrieve the contexts of log statements, based on their location, Drill generates an Intermediate Representation (IR) [38] of the relevant source code first and applies classic static analysis on the IR to extract the context features. We summarize the key steps of the context feature builder in Algorithm 1 and describe it in more detail below.

First, similar to [39], based on the IR, we create a control flow graph (CFG) of only the calling functions that contain log statements. Here, each node represents a basic block (i.e., a continuous sequence of non-branch statements [38]) and the edges represent possible transfers of control between basic blocks. Next, based on the CFG, we iterate through each instruction (I) in each basic block (BB) of the calling function (F) to identify the log statement as well as its context features. We consider four types of features among them. Since Drill features may be combinations of classic static analysis features, such as sequences of certain statements before or after the log statements, we define four new terms to refer to these features and explain how to retrieve them using standard terms below.

- **ControlType** indicates the type of flow-of-control statements (e.g. *if*, *while*) surrounding the log statement. It helps measure the structural similarity among log statements in Drill. The function `CheckControlType` in Alg. 1 (Line

10-19) calculates this feature. The implementation is fairly straightforward in static program analysis. For example, we check the conditional jump to the basic block to determine whether it `is_conditional_Block`.

- **MessageType** indicates the type of the message included in the log statement, which may indicate the purpose of the log. According to our observation, if the log statement only contains string literals (e.g., `CERROR("libcfs ioctl: user buffer too small for ioctl\n");`), it implies the generated log are the same, which typically serves the purpose of monitoring the current system status. If the log statement further includes non-string variables (e.g., `CDEBUG ("... %d\n", lov_connects)`), it often serves the purpose of checking certain variables. These different purposes lead to different features in Drill. The `CheckMessageType` in Alg. 1 shows how we calculate this feature. Specifically, we infer the message type based on the number of operands in the log statement: a statement with only one operand implies a string-only message, while additional operands in the IR implies that additional variables in the message. In this way, we can determine if a log statement is only for status monitoring or for variable checking.
- **ReturnTypeI** and **ReturnTypeII** catpure the potential relationship between the log statement and the return statements around it. For example, a log statement may be relevant to an *if* statement which checks the return value of another function, we call this a *log–after–return* type (ReturnTypeI). If a log statement is immediately followed by a return statement, we call it a *return–after–log* type (ReturnTypeII). Since the return statements typically imply major transitions in the program flow, their relative positions to log statements help capture the characteristics of the logs. To check these two return types, we leverage the log statement's *basic block* and *next instruction* as shown in Alg. 1 (Line 25-33). Specifically, for ReturnTypeI, we check if there is a conditional jump to the current BB that contains the log statement. If yes, we then extract the *condition* and inspect if it is a return value from another function, which can be done by back-tracing the condition of the last instruction and checking if it is a return value of a *call instruction* to another function. Similarly, for ReturnTypeII, we extract the next instruction of the log statement within the current BB and check if it is a return instruction, which may imply whether there is an immediate return after the log statement.

Algorithm 1: `Context Feature Extraction`

```
   Input: $IR$, $F$
   Output: Context Features
1  Function ExtractContextFeature(IR, $F$)
2      CreateCFG($IR$, $F$)
3      foreach $BB \in F$ do
4          foreach $I \in BB$ do
5              if log_statement
                 ($CERROR \,\|\, CDEBUG$) then
6                  CheckControlType($BB$)
7                  CheckMessageType($I$)
8                  CheckReturnTypeI($BB$)
9                  CheckReturnTypeII($I$)
10 Function CheckControlType($BB$)
11     // implemented by checking conditional jump
12     if $is\_conditional\_Block(BB)$ then
13         control_type ← "if"
14     else
15         // detecting cycle using depth-first-search
16         if $is\_loop\_Block(BB)$ then
17             control_type ← "loop"
18         else
19             control_type ← "null"
20 Function CheckMessageType($I$)
21     if $num\_of\_operands(I) > 1$ then
22         message_type ← "variable_check"
23     else
24         message_type ← "status_monitor"
25 Function CheckReturnTypeI($BB$)
26     if $is\_conditional\_Block(BB)$ then
27         condition ← jump_condition
28         // back-trace the condition
29         if $is\_function\_return\_value$(condition)
             then
30             log-after-return ← "yes"
31 Function CheckReturnTypeII($I$)
32     if $is\_return\_statement$(next_instruction(I)) then
33         return-after-log ← "yes"
```

C. Anomaly Detection Model

After mapping runtime logs to feature vectors, we use the grouped, sequences of relevant logs and their labels to train the anomaly detection model. Here, grouping runtime logs may vary across different systems. For instance, in many distributed file systems, some global IDs may exist to denote a sequence of relevant operations, which can be leveraged to build sessions. For instance, Hadoop HDFS has *block_id* [1], Apache HTTP server has *cache_key* [40], and Hadoop MapReduce has *task_id* [41]. Alternatively, other systems, such as Lustre and BeeGFS, do not have these global Ids. For these systems, we still build sessions for training, except each session now only contains one log entry. Note that we do not construct the sessions by grouping logs within a designated time window mostly because the size of the time window greatly impacts the results. Also, due to the irregularity of log generation, a fixed time window may contain thousands of logs or only several logs, making the training extremely difficult. To ensure reliable results, we will check each log entry in systems that do not have reference IDs to build meaningful sessions.

Drill uses a BiLSTM model which takes sequences as inputs and outputs a probability of abnormality. Each time, the input is a vector representation of the runtime log. Each vector is recurrently forwarded to the BiLSTM Model. The output of each LSTM Cell will be concatenated and flattened, then passed to a dense network, turning into a two-dimensional vector as the logit. Finally, a softmax layer is applied to calculate the probability of normal and abnormal. If the probability of abnormality is higher than 0.5, the model will report there is an anomaly in the session. The BiLSTM network contains two layers, 20 neurons in each layer, and has a total of 17K parameters. The model is trained using a batch size of 64 sessions, along with the Adam optimizer and a learning rate of 0.01. Once trained, the detection model is used to take runtime logs and make predictions regarding the presence of anomalies in the system. For unseen logs, the feature vector database will still provide vectors, which will be fed to the BiLSTM model to make a decision. Note that Drill provides a general interface for sequence-based classification models (e.g., GRU [42], Transformer [43]). In this prototype, we chose BiLSTM mainly for two reasons. First, BiLSTM stacks two layers of LSTM with one layer in a forward pass and the other in a backward pass, capturing more sequential information of logs in bi-direction. Second, it has been used in state-of-the-art anomaly detection research, which performs well in real-world systems [44]. We take on the comparisons of different models as one of our future investigations.

IV. Evaluation

A. Datasets and Evaluation Setup

1) Datasets: To evaluate Drill, we conducted evaluations on two different distributed storage systems: Apache HDFS and Lustre. Here, HDFS is written in Java and Lustre is written in C, which demonstrates the generality of Drill. We collected datasets from both systems to conduct the evaluation. Details about these datasets are discussed below.

TABLE II: Description of datasets

Datasets	# of log entries	# of sessions	# of log index	# of anomalies
HDFS	11,175,629	575,061	29	16,838
HDFS-Upcoming	104,634	4841	35	2277
Lustre	157,874	157,874	73	7,401

Table II shows all the datasets used in these evaluations. First, the *HDFS* dataset is a publicly available set of runtime logs collected from a 200-node cluster running Hadoop 0.17 [17]. There are $\sim$11 million logs in total, which form 575,061 pre-built sessions in the dataset. Among them, 2.9% are anomalies which were labeled by domain experts. There are 29 unique log templates detected among these 11 million runtime logs.

To evaluate how anomaly detection tools will perform in a real-world setting, facing continuously arriving new and probably unseen logs, we further generated the *HDFS-Upcoming* dataset. This dataset was collected using a 4-node CloudLab [45] cluster running the same Hadoop 0.17 version. We ran the built-in benchmark application (i.e. TestDFSIO), which continuously generated logs (some are new compared with the original *HDFS* dataset). We manually labeled each session as normal or abnormal following the same protocol discussed in the paper that originally introduced HDFS dataset [17].

We also generated the *Lustre* dataset to evaluate Drill performance on different storage systems. We generated this set of Lustre logs by running the IO500 [46] benchmark on a Lustre cluster built in CloudLab. To accurately label the logs, we leveraged an open-source fault injection tool called PFault [3] which injects faults into Lustre and recorded the generated logs. We considered logs generated before the injected faults as normal. For logs generated after the fault injections, the domain experts labeled each log entry depending on whether or not it was relevant to the injected faults. Specifically, our labeling criteria leveraged standard Linux error numbers (or equivalent customized error numbers), as Lustre utilizes these extensively while logging. Logs with a standard or equivalent error number were considered to be abnormal. In addition, we pruned any potential noise by examining the log descriptions further. For example, logs related to transient network issues are exempted from the abnormal logs as such transient issues are common in pre-fault logs as well. In total, the numbers of normal and abnormal log entries are 150,473 and 7,401, respectively. Since it is hard to build sessions in Lustre logs, we treat each log as a session. Note that, for both the HDFS and Lustre log datasets, 1% of logs are chosen as the training data and the remaining 99% serve as testing data. We used this specific training and testing data partition for a fair comparison with previous work [14] which used the same data partitioning.

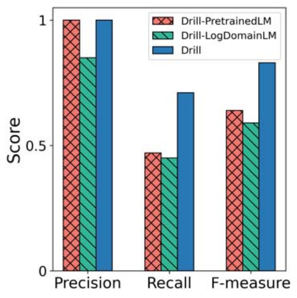
2) Performance Metrics: To compare the performance of different anomaly detection methods, we use four metrics: *Accuracy, Precision, Recall, F-measure* defined as follows:

- Accuracy: measures the percentage of correct predictions (both normal and abnormal) over all predictions.
- Precision: measures the percentage of the reported anomalies which are actually anomalies.
- Recall: measures the percentage of total actual anomalies which are reported.
- F-measure: the harmonic mean of Precision and Recall, which often indicates the quality of the model.

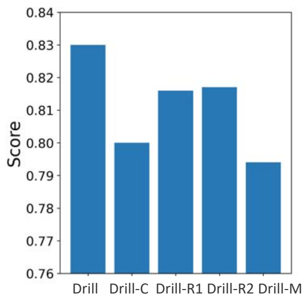
B. Impacts of Sentiment Language Model

As discussed earlier, we argue that re-training or fine-tuning complicated language models to work on storage system logs is inefficient and unnecessary. Drill uses multiple open-source storage systems' source code to fine-tune a sentiment language model using limited training data. To show that this design decision makes sense, we compared different ways of leveraging the language models while keeping other components of Drill unchanged. The results are reported in Figure 7(a). Here, *Drill-PretrainedLM* indicates the direct use of the pre-trained language model from natural language to encode the logs; *Drill-LogDomainLM* uses the source code of multiple storage systems to fine-tune an existing language model (i.e. GLoVe) and uses that to encode the log; *Drill* is our sentiment model-based design.

It can be seen that although *Drill-PretrainedLM* achieves a precision of 1.0, it has considerably worse recall (i.e. 0.47) and hence worse F-measure (i.e. 0.64) compared to *Drill*. We consider two major reasons contributing to such a result: a)



(a) Sentiment



(b) Context

Fig. 7: (a) Performance comparison among Drill using pre-trained language model, Drill using fine-tuned storage system-specific language model, and Drill using our sentiment language model on HDFS dataset; (b) The impact of context features on HDFS dataset. *Drill-C, Drill-R1, Drill-R2, and Drill-M* mean *Drill* without different context features, such as *ControlType, ReturnTypeI, ReturnTypeII, and MessageType*.

the pre-trained language model used in *Drill-PretrainedLM* is not effective for log analysis; b) Compared to the embedding of raw content, sentiment is more effective. As for *Drill-LogDomainLM*, it shows even worse performance than Drill-PretrainedLM, which obviously means it is worse than *Drill*. Note that, even though this is a domain-specific fine-tuned model, we believe the poor result can be attributed to the volume of domain-specific data, which is not enough to fine-tune a workable language model, resulting in one which is not as effective as the sentiment model.

C. *Impacts of Context Features*

One of the new designs of Drill is its integration of both sentiment-based and context-based features. In this section, we further explore how the context features contribute to the performance of Drill. More specifically, we present our evaluation of the effect of each context feature introduced in Drill, such as *ControlType* and *MessageType*. In this evaluation, we used the evaluation results of Drill on the HDFS dataset as the baseline. Then we conducted multiple experiments with one feature deliberately disabled each time. The results are reported in Figure 7(b). Here, we only report the results of F-measure due to limited space. On one hand, F-measure is a reliable metric to indicate the quality of a model. Additionally, the respective precision values of different evaluations showed little variation, hence the recall shared the same trend as the F-measure. From these results, we can see that disabling any one feature leads to a tangible reduction in performance. Especially when disabling *MessageType*, the F-measure was reduced from 0.83 to 0.78. These results prove that the context-based features are necessary and contribute to Drill's overall performance significantly.

D. *Drill on Logs of Different Systems*

In this evaluation, we compared Drill with several index-based and content-based anomaly detection solutions, including two traditional machine learning models, (i.e., Decision Tree and SVM [20], [12], [13]), two deep learning, (i.e., DeepLog [14] and LogAnomaly [15]), and a state-of-the-art solution NeuralLog [27] in statistical analysis.

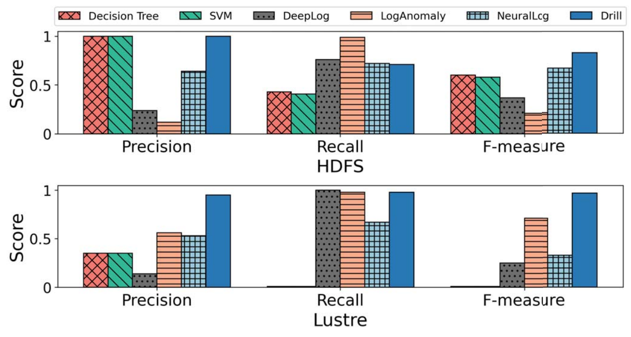


Fig. 8: Performance comparison on HDFS and Lustre datasets.

The experimental results on the HDFS dataset are reported in Figure 8. Among the three reported metrics, we again focus on the *F-measure* metric since it is the harmonic mean of *precision* and *recall* and often indicates the overall quality of the model more accurately. From the results, we observe that Drill achieves the best performance among all the six approaches (i.e., 0.83) on F-measure, presenting its effectiveness. The F-measure of the Decision Tree and SVM are relatively close, with 0.60 and 0.58 respectively. It is noticeable that even though the overall performance of DeepLog and LogAnomaly is not good, they have very high recall, which means that they report more true anomalies. However, they suffer from low precision, which means that only a small number of reported anomalies are true anomalies. Such a high false positive rate is not surprising as both DeepLog and LogAnomaly simply consider all unseen log indices to be anomalies due to their unexpected variation in log patterns when compared to the seen logs. When these unseen logs are actually normal, both of them will have a high false positive ratio. Oppositely, the Decision Tree and SVM methods have high precision (1.00) but low recall (0.43 and 0.41). This means that the anomalies they reported are most likely true anomalies but only cover a very small sets of the total anomalies. This represents another extreme case for handling unseen logs: simply consider them as normal. Both of these approaches let the anomalies in unseen logs slip through, which may put the systems which produce these logs at risk. NeuralLog has a good overall performance due to its improved log vectorization technique. However, its lack of context-based features makes it less suitable to handle some anomalies which have no obvious alarm in their log content but can be inferred through consideration of code context.

The experimental results of the Lustre dataset are reported in Figure 8. Drill still achieves the best performance when compared with other baselines (i.e., 0.97 on F-measure). Deeplog and LogAnomaly suffer from a similar issue on the Lustre dataset as they did on HDFS: they achieve high Recall but low Precision, leading to low F-measure. The

TABLE III: Performance of log-anomaly detection methods for realistic streaming analytics Case 1 and Case 2.

Case1	Hist.(4k)+Upcom.(1k)		Hist.(3k)+Upcom.(2k)		Hist.(2k)+Upcom.(3k)		Hist.(1k)+Upcom.(4k)		Upcom.(5k)	
	Accuracy	F-measure	Accuracy	F-measure	Accuracy	F-measure	Accuracy	F-measure	Accuracy	F-measure
Decision Tree	0.90	0.67	0.80	0.67	0.70	0.67	0.60	0.67	0.88	0.89
SVM	0.90	0.01	0.80	0.01	0.70	0.01	0.60	0.01	0.53	0.01
DeepLog	0.89	0.66	0.79	0.66	0.70	0.66	0.60	0.67	0.54	0.66
LogAnomaly	0.88	0.55	0.76	0.56	0.64	0.57	0.53	0.56	0.40	0.55
NeuralLog	0.97	0.74	0.96	0.65	0.96	0.60	0.96	0.56	0.95	0.49
Drill	0.99	**0.95**	0.99	**0.97**	0.99	**0.98**	0.99	**0.99**	0.95	**0.95**

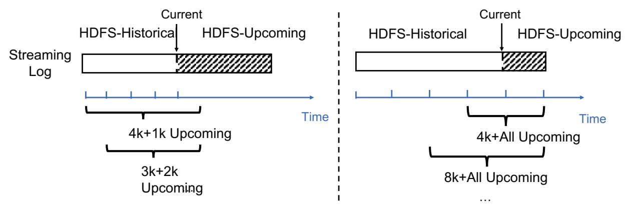
Case2	Upcom.(5k)		Hist.(4k)+Upcom.(5k)		Hist.(8k)+Upcom.(5k)		Hist.(12k)+Upcom.(5k)		Hist.(16k)+Upcom.(5k)	
	Accuracy	F-measure	Accuracy	F-measure	Accuracy	F-measure	Accuracy	F-measure	Accuracy	F-measure
Decision Tree	0.88	0.89	0.76	0.68	0.84	0.68	0.88	0.68	0.90	0.68
SVM	0.53	0.01	0.74	0.01	0.82	0.01	0.87	0.01	0.89	0.01
DeepLog	0.54	0.66	0.75	0.66	0.82	0.65	0.86	0.65	0.89	0.65
LogAnomaly	0.40	0.55	0.67	0.55	0.77	0.55	0.83	0.55	0.86	0.55
NeuralLog	0.95	0.49	0.97	0.72	0.97	0.77	0.97	0.77	0.97	0.79
Drill	0.95	**0.95**	0.97	**0.95**	0.97	**0.92**	0.95	**0.80**	0.97	**0.89**

reason is the same as before: DeepLog and LogAnomaly assume the unseen log indices are anomalies. Decision Tree and SVM perform much worse on the Lustre dataset, with precision of 0.35, recall of 0.01, and F-measure of 0.01. Compared to the performance on the HDFS dataset, apart from the low recall, the precision of both approaches decreased significantly. This means that they report many false positives in the Lustre dataset. Conservatively predicting the unseen logs as normal can give good precision, but if this leads to the misclassification of some normal unseen logs as anomalies it will incur low precision. Interestingly, NeuralLog has worse performance on the Lustre dataset. We consider this is due to the dissimilarity between the Lustre logs and natural language which may confuse the language model which NeuralLog relies on.

E. Drill on Logs in Streaming Analytic Patterns

In this section, we show how Drill and other anomaly detection solutions will perform when they are applied to a more realistic streaming scenario. Specifically, we used the training data in the original HDFS log dataset to train Drill and other systems. Then, we fixed the model and used it to monitor the system where the new HDFS-Upcoming log entries will continuously arrive. The HDFS-Upcoming dataset includes approximately 5K logs in total with 12 new unseen log indices. The HDFS-Historical dataset is the original HDFS testing data. The goal of this evaluation is to compare the effectiveness of different solutions when they were trained using limited historical log data and then applied to process the real-world stream of new logs online. Although computational cost could be an important factor in this online setting, it is not a bottleneck for any of the evaluated solutions. For instance, Drill needs only 0.25 milliseconds to process a new session. Other solutions run at a similar speed.

We simulate two separate cases for handling the streaming new logs. Both cases are shown in Figure 9. In the first case, we assume the anomaly detection tools will fix the size of a monitoring window, and slide it towards up-to-date logs (the HDFS-Upcoming log dataset) gradually. With each measurement, we move forward by approximately 1K new



(a) Case 1 (b) Case 2

Fig. 9: Streaming of logs in realistic log anomaly detection.

log entries and re-evaluate the performance. Note that there are roughly 5K log entries in HDFS-Upcoming dataset so, after sliding four times, the model will only see the HDFS-Upcoming dataset. In the second case, we assume that the anomaly detection tools will keep searching back for more logs to analyze after up-to-date logs arrive. Specifically, we grow the window size from 5K to 21K logs as Figure 9(b) shows.

The evaluation results are reported in Table III. In case 1, we have two major observations. First, Drill has the best performance with regard to both *Accuracy* and *F-measure* among all the six methods for every sliding monitoring window. Second, for *Accuracy*, the other methods have a decreasing trend as more unseen logs arrive, while Drill consistently maintains a stable performance. These results show that Drill remains stable while handling streaming logs. In case 2, the two metrics have a similar trend to case 1. For example, the accuracy of other methods decreases as the size of the monitoring window decreases, while the accuracy of Drill remains the highest throughout the experiment and does not seem to be influenced by changes in the size of the window. Combining the results of case 1 and case 2, we conclude that Drill is the preferred method to be used in a realistic streaming analytic scenario.

F. Robustness of Drill on Partial Training Data

Finally, we evaluate the robustness of Drill. Specifically, we evaluated three settings on the HDFS dataset. In each setting, the training dataset contained only a portion of the total log indices while the testing dataset maintained all of the log

TABLE IV: The impact of different unseen log percentage on different solutions applied to HDFS dataset.

Settings	Full Training Logs			Most Training Logs			Half Training Logs		
	Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure
Decision Tree	0.89	1.0	0.94	1.0	0.45	0.62	1.0	0.43	0.60
SVM	0.98	0.95	0.97	1.0	0.42	0.60	1.0	0.41	0.58
DeepLog	0.82	0.94	0.87	0.17	0.98	0.29	0.24	0.76	0.37
LogAnomaly	0.95	0.90	0.93	0.12	0.99	0.21	0.17	0.99	0.29
NeuralLog	0.94	0.93	0.94	0.96	0.81	**0.87**	0.49	0.50	0.49
Drill	1.0	0.96	**0.98**	0.89	0.81	0.85	1.0	0.71	**0.83**

indices, so that a variable percentage of partial training data is used to evaluate the overall robustness of the models. The settings are listed below. 1) *Full Training Logs:* the original HDFS training dataset without removing any log indices . 2) *Most Training Logs:* based on *Full Training Logs*, we remove 6 log indices from the training data. 3) *Half Training Logs:* based on *Most Training Logs*, we further remove another 6 log indices from the training data.

For each setting, we generate the corresponding training data, train the four models, and compare them. The testing data is fixed as described in IV-A1, making all of the results comparable across different percentages of unseen log indices.

The results are reported in Table IV. It can be seen that for each percentage of unseen log indices, Drill has either the best or second best F-measure among all the approaches we considered. Additionally, as the percentage of unseen log indices increases, Drill has a stable overall performance while the performance of other approaches declines abruptly. The presented robustness of Drill is primarily due to the fact that it leverages the learned differences between seen and unseen logs using feature vectors which effectively encapsulate vital source code context and log content information.

V. Related Work

Log-based anomaly detection has been extensively studied recently. Generally, existing methods can be classified into three categories. The rule-based methods leverage expert-defined rules to assign unmatched log entries as anomalies [7], [8], [9], [11]. For example, LogLens [11] predefined a series of patterns that represent normal logs and considers logs that do not match such patterns as anomalies. Rule-based methods rely on expert knowledge and regularity in the logs, hence are limited in handling unseen logs.

Index-based methods [16], [17], [18], [19], [14] treat the runtime logs as independent entities encoded using index numbers. Once an index sequence is built, various methods ranging from statistical analysis [18] to deep learning models [14] can be applied to learn the patterns. DeepLog [14] utilizes an LSTM neural network to learn the pattern of seen normal log indices and treats unseen logs as anomalies. The index-based methods share this potential problem when applied to unseen logs. Drill is proposed to address such an issue. Specifically, Drill extracts sentiment-based and context-based features for log statements and constructs feature vectors to represent each runtime log, which allows the model to extend learned patterns to unseen logs.

Log content-based methods include static analysis [47] and natural language processing strategies [15], [44], [48], [31], [27], [49]. Among them, the NLP-based strategies offer advanced models to understand the log contents. For example, Meng et al. [15] extend DeepLog by considering the synonyms and antonyms in the log content. Le et al. [27] utilize BERT [26] to extract the semantic information of raw log content. Particularly, Zhang et al. [31] train sentiment analysis from the source code to directly predict the anomaly of each single runtime log. Compared with these studies, Drill takes advantage of both the sentiment model and context-based features from source code analysis to generate more accurate representations of logs, which leads to more robust results as our evaluation results have shown.

VI. Conclusion and Future Work

This paper presents Drill, a new log-based anomaly detection solution based on source code analysis. Drill introduces two new designs for generating more accurate and robust vector representations of logs: a storage system-specific sentiment language model and context-based feature extraction. Our evaluations show Drill outperforms state-of-the-art approaches on two representative large-scale storage systems, HDFS and Lustre. In the future, we plan to further investigate more features or even automated features. Additionally, we plan to apply more sophisticated language models, such as BERT [26] for sentiment analysis.

Acknowledgments

We sincerely thank the anonymous reviewers for their valuable feedback. This work was supported in part by NSF grants CCF-1910727, CNS-2008265, CCF-1910747, and CNS-1943204.

References

[1] "Apache HDFS," https://hadoop.apache.org, Accessed: 01/2023.

[2] "Lustre," https://www.lustre.org, Accessed: 01/2023.

[3] J. Cao, O. R. Gatla, M. Zheng, D. Dai, V. Eswarappa, Y. Mu, and Y. Chen, "Pfault: A general framework for analyzing the reliability of high-performance parallel file systems," in *Proceedings of the 2018 International Conference on Supercomputing (ICS'18)*, 2018.

[4] A. Das, F. Mueller, P. Hargrove, E. Roman, and S. Baden, "Doomsday: Predicting which node will fail when on supercomputers," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'18)*, 2018.

[5] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari, "Failures in large scale systems: long-term measurement, analysis, and implications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'17)*, 2017.

[6] R. Han, O. R. Gatla, M. Zheng, J. Cao, D. Zhang, D. Dai, Y. Chen, and J. Cook, "A study of failure recovery and logging of high-performance parallel file systems," *ACM Transactions on Storage (TOS'21)*, 2021.

[7] M. Cinque, D. Cotroneo, and A. Pecchia, "Event logs for the analysis of software failures: A rule-based approach," *IEEE Transactions on Software Engineering (TSE'12)*, 2012.

[8] S. Roy, A. C. König, I. Dvorkin, and M. Kumar, "Perfaugur: Robust diagnostics for performance anomalies in cloud services," in *2015 IEEE 31st International Conference on Data Engineering (ICDE'15)*, 2015.

[9] A. Oprea, Z. Li, T.-F. Yen, S. H. Chin, and S. Alrwais, "Detection of early-stage enterprise infection by mining large-scale log data," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'15)*, 2015.

[10] K. Yamanishi and Y. Maruyama, "Dynamic syslog mining for network failure monitoring," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining (KDD'05)*, 2005.

[11] B. Debnath, M. Solaimani, M. A. G. Gulzar, N. Arora, C. Lumezanu, J. Xu, B. Zong, H. Zhang, G. Jiang, and L. Khan, "Loglens: A real-time log analysis system," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS'18)*, 2018.

[12] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer, "Failure diagnosis using decision trees," in *International Conference on Autonomic Computing, 2004. Proceedings. (ICAC'04)*, 2004.

[13] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo, "Failure prediction in ibm bluegene/l event logs," in *Seventh IEEE International Conference on Data Mining (ICDM'07)*, 2007.

[14] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*, 2017.

[15] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun *et al.*, "Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs." in *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI'19)*, 2019.

[16] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Online system problem detection by mining patterns of console logs," in *2009 Ninth IEEE International Conference on Data Mining (ICDM'09)*, 2009.

[17] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP'09)*, 2009.

[18] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, "Mining invariants from console logs for system problem detection." in *USENIX Annual Technical Conference (USENIX ATC'10)*, 2010.

[19] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C'16)*, 2016.

[20] S. He, J. Zhu, P. He, and M. R. Lyu, "Experience report: System log analysis for anomaly detection," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE'16)*, 2016.

[21] "Log4J," https://logging.apache.org/log4j/2.x/, Accessed: 01/2023.

[22] "SLF4J," http://www.slf4j.org, Accessed: 01/2023.

[23] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *2017 IEEE International Conference on Web Services (ICWS'17)*, 2017.

[24] M. Du and F. Li, "Spell: Streaming parsing of system event logs," in *2016 IEEE 16th International Conference on Data Mining (ICDM'16)*, 2016.

[25] N. Buzikashvili, "Sliding window technique for the web log analysis," in *Proceedings of the 16th international conference on World Wide Web (WWW'07)*, 2007.

[26] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[27] V.-H. Le and H. Zhang, "Log-based anomaly detection without log parsing," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE'21)*, 2021.

[28] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP'14)*, 2014.

[29] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing (TSP'97)*, 1997.

[30] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional lstm and other neural network architectures," *Neural networks*, 2005.

[31] D. Zhang, D. Dai, R. Han, and M. Zheng, "Sentilog: Anomaly detecting on parallel file systems via log-based sentiment analysis," in *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'21)*, 2021.

[32] "BeeGFS," http://beegfs.io, Accessed: 01/2023.

[33] "OrangeFS," http://www.orangefs.org, Accessed: 01/2023.

[34] "CephFS," https://ceph.io/ceph-storage/file-system/, Accessed: 02/2023.

[35] "Intel DAOS," https://daos-stack.github.io, Accessed: 01/2023.

[36] "GlusterFS," https://www.gluster.org, Accessed: 01/2023.

[37] T.-H. Chen, S. W. Thomas, and A. E. Hassan, "A survey on the use of topic models when mining software repositories," *Empirical Software Engineering (ESE'16)*, 2016.

[38] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, & tools*, 2007.

[39] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "DR. CHECKER: A soundy analysis for linux kernel drivers." in *26th USENIX Security Symposium (USENIX Security'17)*, 2017.

[40] "Apache Http Server," https://httpd.apache.org, Accessed: 01/2023.

[41] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM (CACM'08)*, 2008.

[42] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[43] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems (NeurIPS'17)*, 2017.

[44] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li *et al.*, "Robust log-based anomaly detection on unstable log data," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE'19)*, 2019.

[45] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb *et al.*, "The design and operation of CloudLab," in *2019 USENIX annual technical conference (USENIX ATC'19)*, 2019.

[46] "IO500," https://io500.org, Accessed: 01/2023.

[47] K. Rodrigues, Y. Luo, and D. Yuan, "CLP: Efficient and scalable search on compressed text logs," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*, 2021.

[48] S. Huang, Y. Liu, C. Fung, R. He, Y. Zhao, H. Yang, and Z. Luan, "Hitanomaly: Hierarchical transformers for anomaly detection in system log," *IEEE Transactions on Network and Service Management (TNSM'20)*, 2020.

[49] C. Egersdoerfer, D. Zhang, and D. Dai, "Clusterlog: Clustering logs for effective log-based anomaly detection," in *2022 IEEE/ACM 12th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS'22)*, 2022.