CloudSeer: Workflow Monitoring of Cloud Infrastructures via Interleaved Logs

Xiao Yu
North Carolina State University
xyu10@ncsu.edu

Pallavi Joshi Jianwu Xu
NEC Laboratories America
pallavijoshi84@gmail.com
jianwu@nec-labs.com

Guoliang Jin
North Carolina State University
guoliang_jin@ncsu.edu

Hui Zhang Guofei Jiang
NEC Laboratories America
{huizhang, gfj}@nec-labs.com

Abstract

Cloud infrastructures provide a rich set of management tasks that operate computing, storage, and networking resources in the cloud. Monitoring the executions of these tasks is crucial for cloud providers to promptly find and understand problems that compromise cloud availability. However, such monitoring is challenging because there are multiple distributed service components involved in the executions.

`CloudSeer` enables effective workflow monitoring. It takes a lightweight non-intrusive approach that purely works on interleaved logs widely existing in cloud infrastructures. `CloudSeer` first builds an automaton for the workflow of each management task based on normal executions, and then it checks log messages against a set of automata for workflow divergences in a streaming manner. Divergences found during the checking process indicate potential execution problems, which may or may not be accompanied by error log messages. For each potential problem, `CloudSeer` outputs necessary context information including the affected task automaton and related log messages hinting where the problem occurs to help further diagnosis. Our experiments on OpenStack, a popular open-source cloud infrastructure, show that `CloudSeer`'s efficiency and problem-detection capability are suitable for online monitoring.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids, Distributed debugging, Monitors; D.4.5 [*Operating Systems*]: Reliability; D.4.8 [*Operating Systems*]: Performance

Keywords Cloud Infrastructures; Distributed Systems; Log Analysis; Workflow Monitoring

ASPLOS '16 April 2–6, 2016, Atlanta, Georgia, USA

DOI: http://dx.doi.org/10.1145/2872362.2872407

1. Introduction

With the fast growth of the global cloud-computing market, many dedicated software infrastructures have emerged to provide convenient access to cloud-based computing, storage, and networking resources. For example, Amazon Elastic Compute Cloud (EC2) [3] and Microsoft Azure [10] are two widely used public cloud infrastructures that enable users to easily create computing platforms in the cloud with multiple servers and configurable resources. In the open-source community, OpenStack [11] has been gaining popularity steadily in recent years [1].

Cloud infrastructures provide access to cloud resources via a series of management tasks, e.g., tasks enabling users to spawn virtual machines (VMs), stop VMs, and delete VMs. To execute a task, cloud infrastructures coordinate multiple internal service processes distributed on different server nodes, each of which takes part in the whole execution (e.g., a scheduler to assign VMs to different nodes and hypervisors on assigned nodes to boot up VMs). The distributed and multi-process nature of task executions introduces extra complexity and non-determinism, which can sometimes result in subtle problems.

From the perspective of cloud providers, monitoring is an important way to help the identification and diagnosis of execution problems that compromise cloud availability. The provider-side administrators expect to know through monitoring if each task execution completes successfully and in time. When execution problems occur, administrators also expect enough information from monitoring that can guide the diagnosis. However, cloud infrastructures usually do not expose enough execution details. De facto practices include checking the outcomes of test requests [8] and monitoring running statuses of service processes and resource usages to detect abnormalities. Unfortunately, test requests may fail to exercise and expose problems in specific problematic execution paths, and resource usages may not reveal problems that do not cause usage irregularities (e.g., expected inter-

service messages not sent or received). As a result, revealing internal execution steps across service processes for direct monitoring in the workflow level is challenging.

Monitoring logs is another common practice to find and diagnose problems. For example, Amazon CloudWatch [2] provides a log-streaming interface for real-time log monitoring on VM instances. There are also open-source projects, such as Logstash [9], to support a similar feature for centralizing distributed logs from multiple server nodes to a single location. However, it is tedious and labor-intensive to manually monitor and understand numerous distributed and interleaved log messages by human effort. In a real-world banking system, that some of the authors had experiences with, there were 200 full-time operators dedicated to 24x7 log monitoring with 67 screens for 190 subsystems.

To facilitate monitoring through logs, we present a non-intrusive lightweight approach for *workflow monitoring*. Our prototype for OpenStack, `CloudSeer`, purely works on logs that are readily available in existing cloud infrastructures. `CloudSeer` is backed by the observations that logging is a general practice for complex software systems and logs implicitly carry workflow information. In these logs, not only error messages, but also sequence-based information, can help monitor and discover execution problems. To enable log-based workflow monitoring, `CloudSeer` addresses the following key challenges.

Working with interleaved log sequences. When a cloud infrastructure is executing multiple tasks in parallel, log messages from different executions are usually interleaved in log files. Even in a single execution, asynchronous operations among different services can interleave log messages in different ways. To recover workflow information, it is necessary, but not straightforward, to identify the correspondence between log messages and task executions. Some existing work [18] assumes that messages have unique request and thread identifiers, which can distinguish messages from different task executions. However, such an assumption does not always hold. While working with OpenStack, we found that there could be multiple non-unique identifiers representing different entities (e.g., VMs, physical machines, and networks) in different log messages. Only the combinations of some identifiers may identify sequences of log messages that belong to different task executions.

Working in an online scenario. The sooner a problem is noticed, the quicker can it be fixed and the more can its effects be mitigated. Existing work [21, 24, 29] on detecting problems using data mining and clustering on logs can only operate in an offline manner, since they require task or program executions to finish so that logs are available in their entirety beforehand. Such a design delays problem detection and is more problematic if the executions may not finish at all. In the monitoring scenario, the detection of execution problems should be quick without the need to wait for the tasks to finish.

Detecting problems without explicit error messages. Execution problems can manifest in different forms. Some of them may not be easily noticeable, e.g., performance degradation and silent failures that do not come with explicit error messages. From the logging practices in OpenStack [11], we found that failures were not always accompanied with indicative error messages. A recent study [31] on logging practice also reports that over half of the studied failures were not properly logged. Thus, only looking for error messages might not reveal subtle execution problems.

Providing workflow information. As a monitoring approach, just reporting a failure is not enough for administrators to diagnose the failure and take further actions. The context of a failure, particularly, workflow [14, 23], is useful for guiding administrators to narrow down and isolate possible causes. A monitoring approach should leverage log messages to include such information in failure reports.

`CloudSeer` addresses these challenges by combining an offline modeling stage (Section 3) and an online checking stage (Section 4). In the offline modeling stage, `CloudSeer` builds an automaton for each task via the logs generated by multiple correct executions. The result automaton depicts the task workflow, in which state transitions capture temporal dependencies between log messages. We introduce this automaton to serve two roles. `CloudSeer` uses it as a specification to identify log sequences in the online checking stage. The automaton is also a bookkeeper to keep track of the execution progress or context on-the-fly by maintaining automaton states. The bookkeeping enables the automaton to be used as an informative output that provides context information when `CloudSeer` detects a potential problem.

In the online checking stage, `CloudSeer` works on a log stream that consists of interleaved log messages from distributed services. It uses the pre-built automata to identify individual log sequences out of interleaved ones, and check them for divergences that may indicate execution problems.

We define two divergence criteria for two symptoms of execution problems: (1) error messages and (2) expected log messages not appearing within a time interval. While the first case is straightforward as it represents failures that are properly caught and handled by a system, the second case indicates unhandled silent failures that are not accompanied by error messages (e.g., when a key service stops responding) or performance degradation. In this paper, we do not distinguish between silent failures and performance degradation. Our goal is to show that `CloudSeer` is capable of detecting different kinds of execution problems, rather than directly identifying the root causes of the detected problems.

To ensure efficiency for monitoring via interleaved logs, we design effective heuristics that leverage non-unique message identifiers to group related log messages into growing sequences on-the-fly. Based on the heuristics, `CloudSeer` associates a limited number of automaton instances with each sequence and checks if the sequence followed by a

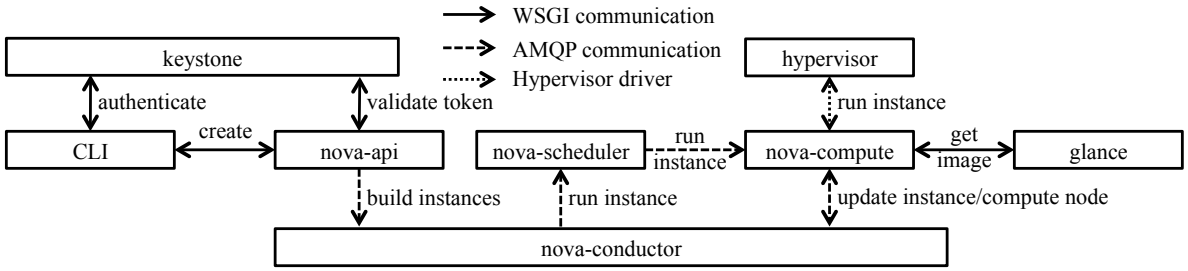

Figure 1. Inter-service interactions when booting a VM

new message still conforms with its associated automaton instances. As a result, `CloudSeer` is able to avoid the brute-force checking on each message against all automaton instances, thus significantly reducing performance overhead.

During the checking process, `CloudSeer` marks and reports automaton instances that satisfy the divergence criteria. The reported instances provide workflow information about the task and the step within the task where the problem is occurring. Administrators can then use the provided information to understand and mitigate ongoing problems.

We evaluate `CloudSeer` by a series of experiments on primitive VM tasks of OpenStack (Section 5). The experiments evaluate how accurate and efficient `CloudSeer` checks interleaved log sequences and its capability of detecting some injected common problems. The results show that the checking accuracy of `CloudSeer` is at least 92.08% on interleaved logs, with a satisfactory efficiency (an average of 2.36s/1k-messages in our test bed), making it suitable for the monitoring scenario. The results further show `CloudSeer`'s problem-detection capability with a precision of 83.08% and a recall of 90.00% on the injected problems.

2. Overview

We start with a general description about a representative cloud infrastructure, OpenStack, and a common set of logging characteristics that exist in similar systems. Then, we use examples based on OpenStack to briefly describe how `CloudSeer` monitors task executions through logs.

2.1 Cloud-Infrastructure Example: OpenStack

OpenStack is an open-source cloud-infrastructure platform. It consists of multiple service components that manage computing, storage, and networking resources in the cloud. For example, the component `nova` provides a set of tasks to manage VMs. Cloud users can issue task commands, e.g., "`nova boot`" and "`nova stop`" from the command-line interface (CLI) to boot up and stop VMs.

The execution of a task involves coordination between multiple service components. Figure 1 depicts how service components interact during the booting of a VM. When a user executes a task with the "`nova boot`" command, a request is sent to `nova-api`, which is a web service built using Web Server Gateway Interface (WSGI). The authentication service, `keystone`, authenticates the incoming request. If authenticated, the scheduling service, `nova-scheduler`, selects a compute server on which to boot up a new VM. The compute service, `nova-compute`, running on the compute server, obtains the appropriate VM image from `glance`, which provides VM images. The VM hypervisor then takes over the request from `nova-compute` to launch the VM. The proxy service, `nova-conductor`, records states of VMs in the database and manages remote procedure calls (RPCs) across service boundaries using Advanced Message Queuing Protocol (AMQP).

2.2 Logging Characteristics

We are aware of two logging characteristics that make the log-based workflow monitoring not straightforward. The two characteristics are the manifestation of a set of underlying system behaviors, which are common in similar systems and not specific to OpenStack examples.

First, besides the message interleaving caused by simultaneous task executions, log messages from the same execution might also be interleaved. This is caused by asynchronous operations (e.g., AMQP communications in OpenStack), which create multiple execution paths through different service processes. As a result, log messages from different paths could be interleaved without a fixed ordering, making task workflows more complicated to monitor.

Second, there is often no single and unique identifier associated with log messages related to a task execution. In cloud infrastructures, each service manages its own resources (e.g., users, VMs, and images are managed by different services) and logs them with identifiers based on resource types. Those identifiers may not be propagated across different services during a task execution, because different services do not share the knowledge on different resources. For instance, a hypervisor knows which VM is running, but it has no knowledge about the user whom the VM belongs to. Consequently, different log messages may not share common identifiers.

We use an example in OpenStack to further illustrate the two characteristics. Figure 2 shows simplified log messages for two executions of the "`nova boot`" task. To ease the explanation, we simplify messages by substituting timestamps with numbers at the beginning of the messages, remov-

```
(1) api accepted IP1
(2) api accepted IP2
(3) api [UUID1] IP1 "POST /UUID2/servers"
(4) api [UUID3] IP2 "POST /UUID4/servers"
(5) scheduler [UUID1] Scheduling instance UUID5.
(6) scheduler [UUID3] Scheduling instance UUID6.
(7) api [UUID3] IP2 "GET /UUID4/servers/UUID6"
(8) compute [UUID1] Starting instance UUID5.
(9) api [UUID1] IP1 "GET /UUID2/servers/UUID5"
(10) compute [UUID3] Starting instance UUID6.
(11) compute Instance UUID5 spawned successfully.
(12) compute Instance UUID6 spawned successfully.
```

Figure 2. Simplified Log Sequences for Booting VMs in OpenStack

ing logging levels, replacing identifiers (IPs, URLs, UUIDs, etc.) with simple text (e.g., replace the actual IP address with just *IP1*), and shortening text in the messages (e.g., `api` for `nova-api`). The identifiers in log messages indicate different resources, e.g., the identifiers in square brackets indicate users who initiate the tasks, the identifiers preceded by the text "`Instance`" indicate VMs, and the identifiers in the message bodies may indicate tenants or VMs.

In Figure 2, there are two interleaved log sequences, $1 \rightarrow 3 \rightarrow 5 \rightarrow 8 \rightarrow 9 \rightarrow 11$ and $2 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 10 \rightarrow 12$. Messages $8 \rightarrow 9$ in the first sequence and $7 \rightarrow 10$ in the second one show two different orderings for one pair of log messages. This is because the service `nova-scheduler`, which generates messages 5 and 6, uses asynchronous AMQP communications to schedule VM instances *UUID5* and *UUID6* to compute nodes. Such asynchronicity causes `nova-api` (generating messages 7 and 9) and `nova-compute` (generating messages 8 and 10) to execute their parts of the task in parallel, resulting in two possible orderings.

It is also noticeable how diversified the log identifiers are. For instance, messages 1 and 5 from the same sequence do not have any common identifiers. On the other hand, many a time some messages can connect other seemingly disconnected ones. For instance, message 3 shares *IP1* and *UUID1* with messages 1 and 5, respectively. It can effectively establish the relation between messages 1 and 5. This observation suggests that shared identifiers, regardless of how far they are propagated across services, can create a chain that transitively connects messages together. This observation thus enables a heuristic that separates interleaved log sequences.

2.3 Demonstration of CloudSeer

We show the basic idea of `CloudSeer` using the example in Figure 2. We assume that we have already built a task automaton using logs collected from multiple normal executions of booting VMs. Figure 3 shows the automaton for the "`nova boot`" task, and it is simplified accordingly as what we have done for the log sequence in Figure 2. Section 3 describes the modeling step in detail.

The Task Automaton. The automaton in Figure 3 represents the booting-VM workflow by encoding the dependencies among log messages. The automaton takes message templates as inputs. The message templates are log messages without variable parts. Determining the variable parts requires certain domain knowledge. We consider IP addresses, UUIDs, and numbers as the variable parts in log messages.

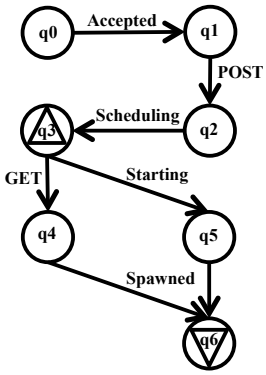

Figure 3. The Automaton for Booting a VM

To model interleaved messages caused by asynchronous operations, we introduce two special types of states in addition to the conventional automaton states: the *fork* states ($q3$ in Figure 3) and the *join* states ($q6$). A *fork* state has multiple outbound transitions for log messages that do not have dependencies with each other but share the same precedent, and each outbound transition transits to a new state from the fork state (e.g., $q4$ and $q5$ are forked from $q3$ after consuming the messages "`GET`" and "`Starting`"). A *join* state has multiple inbound transitions for a log message that must always be followed by some other messages, and all inbound transitions transit to the same join state (e.g., $q4$ and $q5$ join to $q6$ after consuming the message "`Spawned`").

Checking Interleaved Sequences. `CloudSeer` takes one log message at a time. It checks if the message belongs to a growing log sequence and if the growing sequence with the new message still conforms with its corresponding automaton. This checking process continues until a message completes a sequence or satisfies divergence criteria.

To leverage multiple identifiers to determine the association of a message with a developing log sequence, `CloudSeer` uses a data structure, *identifier set*, as the signature of each log sequence. An *identifier set* stores identifiers appearing in all messages of a sequence.

Table 1 shows the process of `CloudSeer` checking log messages. The column "Identifier Set" presents the identifier set after processing the corresponding message shown in the column "Message." Due to space limit, we use the symbol $UUID1/2$ to represent $UUID1$ and $UUID2$ in the column "Identifier Set." The column "Instance Transition" shows the state transitions of automaton instances. The label before each identifier set and state transition indicates the log sequence under checking.

`CloudSeer` considers messages 1 and 2 to be the beginning of two new sequences because their identifier sets do

Message	Identifier Set	Instance Transition
(1) `Accepted`	1: $\{IP1\}$	1: $\{q0\} \rightarrow \{q1\}$
(2) `Accepted`	2: $\{IP2\}$	2: $\{q0\} \rightarrow \{q1\}$
(3) `POST`	1: $\{IP1, UUID1/2\}$	1: $\{q1\} \rightarrow \{q2\}$
(4) `POST`	2: $\{IP2, UUID3/4\}$	2: $\{q1\} \rightarrow \{q2\}$
(5) `Scheduling`	1: $\{IP1, UUID1/2/5\}$	1: $\{q2\} \rightarrow \{q3\}$
(6) `Scheduling`	2: $\{IP2, UUID3/4/6\}$	2: $\{q2\} \rightarrow \{q3\}$
(7) `GET`	2: $\{IP2, UUID3/4/6\}$	2: $\{q3\} \rightarrow \{q3, q4\}$
(8) `Starting`	1: $\{IP1, UUID1/2/5\}$	1: $\{q3\} \rightarrow \{q3, q5\}$
(9) `GET`	1: $\{IP1, UUID1/2/5\}$	1: $\{q3, q5\} \rightarrow \{q4, q5\}$
(10) `Starting`	2: $\{IP2, UUID3/4/6\}$	2: $\{q3, q4\} \rightarrow \{q4, q5\}$
(11) `Spawned`	1: $\{IP1, UUID1/2/5\}$	1: $\{q4, q5\} \rightarrow \{q6\}$
(12) `Spawned`	2: $\{IP2, UUID3/4/6\}$	2: $\{q4, q5\} \rightarrow \{q6\}$

Table 1. Demonstration of Checking Process

not have any common element. Therefore, `CloudSeer` creates a new identifier set and a new automaton instance for each of these two messages.

The identifiers in message 3 have a common element with the identifier set of the first sequence ($\{IP1\}$) only. Therefore, `CloudSeer` associates the message with the sequence "1." It then tests if the message causes the corresponding automaton instance to make a state transition. As the transition happens, `CloudSeer` extends the identifier set of the sequence with the UUIDs in the message. When message 4 arrives, `CloudSeer` associates the message with sequence "2" since the IP address $IP2$ appears in the current identifier set for that sequence. For the rest of the messages, `CloudSeer` repeats this checking process until the two automaton instances accept their log sequences.

Interpreting Results. When an incoming message, or lack thereof, satisfies divergence criteria, `CloudSeer` is able to report the diverged automaton instance indicating a problematic execution. For investigation, the diverged instance provides useful information that includes the ongoing task with its normal workflow, and the execution point where the problem occurs in the form of the current automaton state(s) and all log messages consumed so far.

For example, in Figure 2, if a problem occurs in the compute node assigned to run the VM $UUID5$, the message 8 or 11 may not appear. Such a case is detectable by a "timeout" criterion that specifies a time period in which an automaton instance must consume a message. `CloudSeer` would then report an instance with the states $\{q3, q4\}$ or $\{q4, q5\}$, depending on if the message 8 has appeared. Then, a knowledgeable administrator may quickly narrow down the problem to immediately before booting the VM (if $\{q3, q4\}$, network could be the main cause) or during the booting process (if $\{q4, q5\}$, I/O or hypervisor could be the main cause).

3. Modeling Task-Based Log Sequences

The offline modeling process creates automata from logs generated by correct task executions. `CloudSeer` uses the output automata to check interleaved logs in the online scenario. For each task to model, the modeling process takes *log sequences* from multiple executions of the task. The term log sequence denotes all log messages ordered by their timestamps (or in the order they arrive) from a single execution. Since asynchronous operations among services may cause some messages to be interleaved in different ways, using multiple log sequences of a task helps the modeling process reconstruct *temporal dependencies*, which describe the order relation of log messages to represent the task workflow.

The modeling process consists of three steps: (1) preprocessing log sequences to extract key log messages for the task workflow; (2) mining temporal dependencies in the preprocessed log sequences; and (3) constructing automata from the mined temporal dependencies.

3.1 Preprocessing

The preprocessing step takes log sequences as input. A log sequence LS is a vector of log messages $\langle m_1, m_2, ..., m_n \rangle$. Each log message m_i in LS has two attributes: a template t representing the constant text of m_i, and a value set S_v containing values in the variable parts of m_i. There are different ways to determine the template t and the value set S_v of a log message m, and we consider numbers, IP addresses, and UUIDs as variable parts, where UUID is in the form of a well-formatted string. We use regular expressions to match and extract them to create $m.S_v$, and $m.t$ is naturally the part of m after the extraction.

For a set of log sequences S_{LS} collected from multiple executions of a task, the preprocessing step keeps key log messages in each log sequence of S_{LS} and removes all other messages which are considered less relevant to the task workflow. To determine if a log message m is a key message, this step counts the appearance of $m.t$ in each log sequence of S_{LS}. The message m is a key message if $m.t$ appears the same number of times in every sequence.

The preprocessing provides two benefits. First, it removes irregular log messages from loops, conditional choices, and periodical background tasks. Such messages are irrelevant to the task workflow and would introduce excessive possibilities when `CloudSeer` checks the interleaved log sequences. We consider that keeping only key messages is sufficient and efficient for the monitoring purpose. Second, the result sequences bring out in-sequence message interleaving caused by asynchronous operations. Capturing and modeling such interleaving is important for recovering the task workflow.

3.2 Mining Temporal Dependencies

Given a set of preprocessed log sequences S_{LS}, *temporal dependencies* describe the order relation in which collaborative services generate log messages. We define the following two types of temporal dependencies over log templates $\{t_1, t_2, ..., t_n\}$ shared by sequences in S_{LS}: (1) a strong dependency $t_i \xrightarrow{strong} t_j$ iff t_j is always next to t_i for each of their occurrences in S_{LS}; (2) a weak dependency $t_i \xrightarrow{weak} t_j$ iff t_i always happens before t_j in S_{LS}, but it is not necessarily followed by t_j immediately. Any pair of t_i and t_j

not having the two dependencies is considered to not have any specific order. For example, in Figure 2, the messages "accepted" and "POST" have a strong dependency, since both the occurrences $1 \rightarrow 3$ and $2 \rightarrow 4$ follow one specific order. On the other hand, the message "Scheduling" (5 and 6) always leads "Starting" and "GET" ($5 \rightarrow 8 \rightarrow 9$ and $6 \rightarrow 7 \rightarrow 10$), but the latter two messages do not follow a specific order. Therefore, the messages "Starting" and "GET" have weak dependencies with "Scheduling." The weak dependencies imply that the execution becomes asynchronous after the message "Scheduling" and before "Starting" and "GET."

To mine temporal dependencies, this step consists of three parts. It first enumerates and collects all pairs of log templates for each sequence in S_{LS}. For example, given a sequence $\langle a, b, c\rangle$, this step enumerates pairs of $(a, b), (a, c), (b, c)$. Then it classifies all collected pairs into strong and weak temporal dependencies based on their definition. Finally, it performs a reduction of transitive relations on the classified temporal dependencies. The transitive relations are the result of enumerating all message pairs. In particular, for any dependencies $a \rightarrow b$ and $b \rightarrow c$ before the reduction, there must be a dependency $a \rightarrow c$. The reduction removes such dependencies as $a \rightarrow c$ to keep the result dependencies minimal.

3.3 Constructing Automata

This step constructs a *task automaton* that encodes the mined temporal dependencies. A task automaton is a septuple $(Q, \Sigma, \Delta, q_0, F, Q_f, Q_j)$ where: (1) Q is a set of states of a task execution; (2) Σ is a set of log message templates; (3) Δ is a transition function $Q \times \Sigma \rightarrow \mathcal{P}(Q)$ based on the temporal dependencies; (4) q_0 is an initial state; (5) F is a set of final states; (6) $Q_f \subset Q$ contains fork states, which lead transitions to multiple other states by messages not dependent on each other but having weak temporal dependencies with the previous message; (7) $Q_j \subset Q$ contains join states, to which multiple states converge on a message having weak temporal dependencies with multiple previous messages.

The construction of a task automaton is as follows. All log templates in the temporal dependencies constitute the input set Σ. Each input in Σ is assigned a state, indicating the state after taking the input. Then, the assigned states with the mined dependency pairs constitute transitions in Δ. To determine the first transition from the initial state q_0, this step looks up an input that does not depend on any others. To determine the final states F, the fork states Q_f, or the join states Q_j, this step looks for the states without any outbound transitions, have multiple outbound transitions, or have multiple inbound transitions in Δ. For each fork state, this step adds a self-loop to the transition function to allow state forking, as shown in the rows 7 and 8 of Table 1. In addition, we limit a fork state to take self-transitions at most the number of its outbound transitions, so that the automaton only processes a finite length of log sequence.

Algorithm 1: Checking Individual Sequences

Input: a log message m; an automaton group G, empty by default
Output: the updated automaton group G after its automaton instances consume the message m
Global: a set of task automata M for different tasks

```
1 G' ← CopyInstances(G);
2 if G' is empty then
3 |   G' ← InitializeInstances(M);
4 G ← {};
5 foreach a in G' do
6 |   if TryInputMessage(a, m.t) is true then
7 |   |   G ← G ∪ {a};
8 return G;
```

4. Checking Interleaved Log Sequences

In the online checking stage, CloudSeer uses task automata from the previous offline modeling stage to check interleaved log sequences. The input and output of this checking stage are as follows. CloudSeer takes one log message at a time from a log stream. The input message belongs to one of some interleaved log sequences being generated. Based on the task automata, CloudSeer outputs (1) an accepting automaton instance, if the last message completes a correct log sequence; (2) an erroneous automaton instance, if the last message meets problem-detection criteria; or (3) no output, indicating more log messages are expected.

We start from the basic case for individual log sequences assuming tasks are executed one by one. Then we relax this assumption and present the full checking algorithm. Finally, we discuss our problem detection criteria.

Basic case: individual sequences. CloudSeer needs to choose a correct automaton from multiple task automata to check incoming log messages belonging to the current log sequence. Since CloudSeer takes one log message at a time, it does not know the right choice of automaton for sure until an automaton accepts the last message, completing the current log sequence.

Algorithm 1 shows the checking algorithm for individual sequences. Its main idea is using an *automaton group* G to keep track of all possible automata for the sequence being checked. G initially contains instances created from a global set of task automata M (Lines 2 to 3). Each automaton instance a in G represents one possible task, and it maintains its own state transitions based on input messages.

For each log message m with the current automaton group G, the algorithm finds the automaton instances that make state transitions with m (Lines 5 to 7, determined by the function *TryInputMessage* that lets an instance decide whether it can consume m), and it passes them to the next invocation by returning the updated G. A final state of any instance in the result G indicates the completion of checking a log sequence. On the other hand, an empty G indicates that none of the automaton instances can consume m.

Checking interleaved sequences. We extend the checking algorithm shown in Algorithm 1 in two ways. First, the

Algorithm 2: Checking Interleaved Sequences

Input: a log message m
Output: an accepting or erroneous automaton instance, if the message m completes a correct log sequence, or meets an error criterion; no output otherwise
Global: a set of identifier sets $\mathcal{I}$; a set of automaton groups $\mathcal{G}$; a relation set $R : \mathcal{I} \times \mathcal{G}$

```
1  S_id ←GetIdentifierValues(m.S_v);
2  I_max ←ComputeMaxIdentifierSets(S_id, I);
3  G_max ←GetAutomatonGroups(R, I_max);
4  G_after ← {};
5  foreach G in G_max do
6  |   G' ← Apply Algorithm 1 with m and G;
7  |   if G' is not empty then
8  |   |   G_after ← G_after ∪ {(G, G')};
9  if |G_after| = 1 then
10 |   G, G' ←Single(G_after);
11 |   ID ←GetAssociatedIdentifierSet(R, G);
12 |   R ← R \ {(ID, G)};
13 |   ID' ←ExpandOrCreateIdentifierSet(ID, m.S_v);
14 |   R ← R ∪ {(ID', G')};
15 else if |G_after| > 1 then
16 |   NID ←CreateIdentifierSet(m.S_v);
17 |   G' ← {};
18 |   foreach (G, G') in G_after do
19 |   |   ID ←GetAssociatedIdentifierSet(R, G);
20 |   |   NID ←ExpandOrCreateIdentifierSet(NID, ID);
21 |   |   G' ← G' ∪ {G'};
22 |   R ← R ∪ [{NID} × G'];
23 else
24 |   Divergence Recovery and Problem Detection
25 a, I, G, R ← PruneIfAcceptingOrErroneous(I, G, R, G_after);
26 return a or None;
```

new algorithm keeps multiple automaton groups simultaneously to track interleaved sequences, where each automaton group tracks the log messages from one task execution. Second, to avoid exhaustively trying every incoming message on all automaton groups, the algorithm associates an *identifier set* with each automaton group, which helps determine the proper automaton group to use on-the-fly.

An identifier set is a signature for an automaton group, and it is generated based on the log sequence that has been checked by the automaton group. During the growth of a sequence, its corresponding identifier set incrementally stores identifiers appearing in log messages of the sequence. Identifier sets serve as the key to determine whether an incoming message belongs to a sequence. This is based on an observation that a log message is likely to belong to a sequence with whose identifier set the message shares the greatest number of common identifiers. Since each automaton group is associated with an identifier set, the checking algorithm can find the most likely automaton group(s) to consume an input message without trying all the groups.

Algorithm 2 shows the detail of checking interleaved log sequences. The algorithm first gets the automaton group(s) associated with the identifier set(s) having the greatest number of common identifiers with the incoming message m (Lines 1 to 3). Then the algorithm uses Algorithm 1 with the selected automaton groups to check m (Lines 4 to 8). The checking may result in three cases: (1) only one automaton group left (the branch from Lines 9 to 14); (2) multiple automaton groups taking m (the branch from Lines 15 to 22); or (3) none of the selected automaton groups to take m (the branch from Lines 23 to 24). After handling the three cases, the algorithm looks for and returns any automaton instance that is in the accepting state, or identifies a potential problem. The algorithm also cleans up related identifier sets, automaton groups, and their relations (Lines 25 to 26).

We describe the handling of the three cases below:

Case (1): decisive checking. When there is only one decisive automaton group, which shares the greatest number of common identifiers with the incoming message and can consume the message, the algorithm applies the steps from Lines 10 to 14 to update the associated identifier set and the relation between the set and the group. In particular, Line 13 expands the set ID by including new identifiers in the message m, i.e., $ID \cup m.S_v$. If the set is associated with multiple automaton groups besides the group taking the message m, the algorithm creates a new identifier set from the original one, and it then expands the new set with new identifiers in m, i.e., two sets ID and $ID \cup m.S_v$. Upon the acceptance of an automaton instance, Line 12 removes the old relation between the original automaton group and its associated identifier set, and Line 14 adds the new relation.

Case (2): brute force and heuristics. There can be multiple automaton groups from Lines 4 to 8. These automaton groups all share the same greatest number of identifiers with the incoming message and can consume the message, but they may have different identifier sets. The algorithm cannot decisively determine an automaton group among all the chosen ones, so it has to keep all of them before and after taking a message to explore different possibilities. For example, suppose there are two automaton groups G_1 and G_2 with identifier sets ID_1 and ID_2, both of which share the most identifiers with an incoming message m. G_1 and G_2 make their own state transitions to G_1' and G_2' by taking the message m. The algorithm keeps (ID_1, G_1) and (ID_2, G_2), and it adds $(ID_1 \cup ID_2 \cup m.S_v, G_1')$ and $(ID_1 \cup ID_2 \cup m.S_v, G_2')$ to the relation set R. When the message after m comes in, the extended identifier set $(ID_1 \cup ID_2 \cup m.S_v)$ gives G_1' and G_2' an equal chance to check the message. As a result, the algorithm tracks both possibilities: G_1' and G_2, or G_1 and G_2'. Upon the acceptance of an automaton instance, the algorithm removes all automaton groups, identifier sets, and relations that are related to the other possibilities.

To reduce the number of possible automaton groups to track, we introduce two heuristics to deal with the two causes of having multiple automaton groups. First, if two identifier sets share the same number of identifiers with an incoming message, the function *ComputeMaxIdentifierSets* also computes the numbers of identifiers that are different between these sets and the message, and it selects the set with the least difference. Second, if there are multiple automaton

groups associated with the same identifier set, the function *GetAutomatonGroups* randomly selects one group when it finds multiple equivalent groups. Two automaton groups are equivalent if they contain automaton instances of the same kind, and the instances of the same kind are in the same state.

Case (3): divergence recovery. Divergences during the checking on the message m result in an empty set returned from Algorithm 1. Such divergences may not indicate execution problems under the following four causes. We provide a recovering heuristic for each of these causes, and these causes are prioritized in the ascending order based on our knowledge of their recovering cost and side effect: (a) m is not in the Σ of any automaton; (b) m starts a new log sequence, so there is no corresponding automaton group; (c) the chosen identifier set is likely not correct, so the automaton group is unable to take m; and (d) m does not arrive in the order defined by the modeled temporal dependencies.

`CloudSeer` applies the recovering heuristics one by one until it recovers. For the cause (a), the algorithm allows unrecognizable messages to pass through. Such messages are usually not of interest to the checking process. If an error message appears, the error-message criterion described below can capture it. For the cause (b), Algorithm 1 is applied with an empty automaton group to create new automaton instances for tracking a new sequence.

If the first two heuristics cannot resolve the divergence, it is likely that the algorithm has chosen a wrong automaton group (the cause (c)). Such situation can sometimes happen if two interleaved log sequences share some identifiers, e.g., the same user boots two VMs at the same time. The algorithm tries another automaton group that has less common identifiers than the best match to take the message.

The unexpected message reordering may fail all other heuristics (the cause (d)). In this situation, a message B arrives earlier than another message A, which should not happen according to the dependency $A \rightarrow B$ from the modeling stage. There are two possible reasons for such reordering: $A \rightarrow B$ is a false dependency or a message-delivery delay causes A to arrive late. We consider that it is unnecessary to determine the particular reason in `CloudSeer`'s usage scenario, so we treat all such reordering as the result of false dependencies encoded in task automata and remove such dependencies from the automaton instances.

Figure 4 shows an example of the removal step. The graph on the left side shows an automaton with a false temporal dependency $B \rightarrow C$. To remove $B \rightarrow C$ from the automaton and make it accept a new sequence $ACBD$, the algorithm first removes the state transition from $q2$ to $q3$, and it adds two new transitions $q1 \rightarrow q3$ and $q2 \rightarrow q4$ (shown by dashed arrows in the right-side graph of Figure 4). The state $q1$ thus becomes a fork state, and $q4$ becomes a join state. These changes reflect two weakened dependencies $A \rightarrow C$ and $B \rightarrow D$ as the results of removing $B \rightarrow C$.

Problem Detection. We implement two simple criteria in `CloudSeer` to detect common execution problems. The two

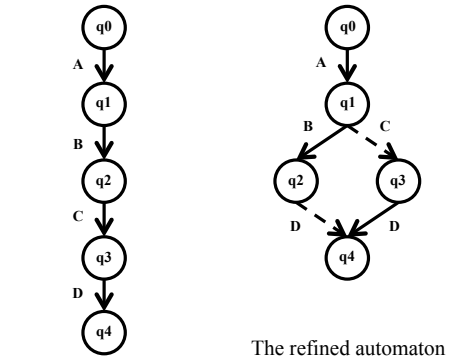

Figure 4. Removing Dependencies

criteria are based on two common manifestations of execution problems: (1) the presence of error messages, indicating task failures; (2) the absence or delay of messages, indicating task failures or performance degradation.

Error-message criterion is in effect when an error message (determined by its logging level) arrives, and the checking algorithm in Algorithm 2 results in a state of divergence, as shown in Line 24. This criterion then applies the functions *ComputeMaxIdentifierSets* and *GetAutomatonGroups* to identify and output the most likely automaton group that may be associated with the error message.

Timeout criterion marks any automaton groups that do not take any messages within a specified time period. The marked automaton groups become the outputs of Algorithm 2. The timeout value may vary in different systems and configurations. Determining such values is not in the scope of this paper, and we leave it for future work.

5. Evaluation

We present a series of systematic experiments conducted on OpenStack to evaluate the following aspects of `CloudSeer`.

Checking Accuracy. The accuracy that `CloudSeer` correctly checks interleaved log sequences is fundamental to `CloudSeer`'s problem detection. We measure the accuracy based on logs collected from correct task executions. `CloudSeer` is supposed to recognize and accept all log sequences in the correct logs. Therefore, any nonacceptance would indicate inaccuracies.

Efficiency. To show `CloudSeer`'s capability of working in the monitoring scenario, we measure the throughput of `CloudSeer` consuming log messages and present how the identifier-based heuristic affects the checking efficiency.

Capability of Problem Detection. With typical execution problems that are injected to certain weak points in the OpenStack system, we show how well `CloudSeer` detects them and provides automaton instances as useful information related to the detected problems.

Task	Description	Msgs	Trans
Boot	Create a new VM.	23	34
Delete	Delete a VM.	9	9
Start	Start a stopped VM.	7	7
Stop	Stop a running VM.	6	6
Pause	Pause a running VM in memory.	7	7
Unpause	Bring back a paused VM.	7	7
Suspend	Suspend a running VM to disk.	6	6
Resume	Bring back a suspended VM.	7	7

Table 2. VM Tasks for Experiments

5.1 Test Bed

We deploy an OpenStack instance in a five-node cluster as the test bed for our experiments. The OpenStack version is *Havana*. In our deployment, there are one controller node, one network node running network-related services, and three compute nodes. We follow the installation guide of OpenStack [5] to deploy this instance.

We set the logging levels for all `nova` service components to *INFO*. Therefore, `nova-api`, `nova-scheduler`, and `nova-compute` generate *INFO*-level messages for VM-related tasks. Such a setting is typical for deployment, and our evaluation suggests that *INFO* level is sufficient to provide workflow information without imposing excessive burdens on servers.

We deploy our prototype of `CloudSeer` alongside Elasticsearch [7], which is a central log database, on the controller node. To centralize logs from OpenStack services, we deploy Logstash [9] on each server node. Logstash parses log messages generated by OpenStack services on each server node and sends the parsed messages to both `CloudSeer` and Elasticsearch on the controller node. While Logstash instances on multiple server nodes create a log stream, Elasticsearch stores all log messages in the stream. We use the log stream and stored log messages in different experiments.

5.2 Workload Generator

To evaluate `CloudSeer` with interleaved log sequences, we implement a workload generator that simulates multiple users submitting OpenStack tasks concurrently through the command-line interface. This generator can create various workloads varying in the number of concurrent users and the number of tasks submitted by each user.

The generator uses eight primitive and typical VM-related tasks to create workloads, and they are listed in Table 2. To obtain the automaton for each task, we keep running the task and applying the modeling algorithm on the output logs, until logs from any subsequent task executions do not change the result automaton. In this way, the result automaton is likely to be concise regarding key log messages and complete regarding relations between messages. The number of runs for modeling each task ranges from 200 to 800, depending on the extent of nondeterminism within task executions. In Table 2, the column "Msgs" shows the number of messages in the log sequence of each execution after preprocessing, and the column "Trans" shows the number of transitions in the corresponding automaton.

Grp.	Data Sets	Users	Single UID?	Total Tasks
1	$1-10$	2	N	1600
2	$11-20$	3	N	2400
3	$21-30$	4	N	3200
4	$31-40$	2	Y	1600
5	$41-50$	3	Y	2400
6	$51-60$	4	Y	3200

Table 3. Experiments for Accuracy and Efficiency

Injection Point	Type	Responsibilities
AMQP-Sender	Net	Request RPCs to services that control VMs, e.g., `nova-scheduler` and `nova-compute`.
AMQP-Receiver	Net	Receive and process RPC requests.
Image-Create	I/O	On each `nova-compute` server node, create a new VM image from an image template.
Image-Delete	I/O	When destroying a VM, delete all related files.
WSGI-Client	Net	Send HTTP-based requests to query image information stored by `glance` services.
WSGI-Server	Net	Process queries of image information.

Table 4. Execution Points for Failure Injection

For each simulated user, the generator randomly populates tasks by the following regular expression:

$$(Boot\,(StopStart \mid PauseUnpause \mid SuspendResume)*\,Delete)+$$

This expression produces multiple task groups, starting with a "boot" task and ending with a "delete" task. Within each task group, there can be multiple pairs of "stop/start," "pause/unpause," and "suspend/resume" tasks on the same VM. When booting a VM, we use CirrOS [6] as the guest system. To ensure the completion of each submitted task, we set each user to wait 15 seconds between two tasks.

5.3 Experiment Design

We design six groups of experiments (shown in Table 3) to evaluate checking accuracy and efficiency of `CloudSeer`. These experiments test two potential factors that may affect the results of accuracy and efficiency: the number of tasks being executed concurrently, and the diversity of identifiers appearing in logs. We control the first factor by setting various concurrent users in the workload generator (shown in the column "Users"). To control the second factor, we set the generator to simulate users with identical or different user identifier (shown in the column "Single UID?") when submitting tasks. We repeat the experiment of each combination of the two factors for 10 times to produce different combinations of interleaved log sequences (shown in the column "Data Sets"). Each simulated user in an experiment is set to submit 80 tasks, resulting in the total numbers shown in the column "Total Tasks."

The evaluation of checking accuracy and efficiency requires measuring multiple aspects of `CloudSeer`'s checking algorithm in a repeatable way. Therefore, we use the same set of stored log messages to feed `CloudSeer` multiple times to ensure consistent measurements.

To show the capability of problem detection, we randomly inject execution problems to the test bed to mimic

Grp.	Acc. Range	Median	% Interleaved ($\geq 2, 3, 4$)
1	$93.24\% - 100.0\%$	96.83%	48.50%
2	$96.82\% - 100.0\%$	98.09%	65.00%, 31.17%
3	$95.78\% - 98.72\%$	97.22%	74.34%, 48.63%, 22.84%
4	$96.15\% - 97.47\%$	97.47%	49.00%
5	$94.16\% - 99.37\%$	98.07%	64.67%, 32.71%
6	$92.08\% - 97.87\%$	96.51%	80.12%, 55.59%, 30.06%

Table 5. Experiment Results for Accuracy

Grp.	Ave. Msgs	Ave. Time	Ave. 1k	% Decisive
1	2164	3.91s	1.81s	83.13%
2	2990	6.25s	2.09s	80.76%
3	3849	8.98s	2.33s	78.18%
4	2158	4.32s	2.00s	80.12%
5	2989	7.37s	2.47s	75.48%
6	3820	11.57s	3.03s	71.43%

Table 6. Experiment Results for Efficiency

the real-world scenario, and let `CloudSeer` monitor the test bed for the injected problems. We configure the workload generator to use four users to submit tasks concurrently. In addition, we configure the timeout value of task automata to be 10 seconds based on the performance of our test bed.

We choose six execution points to inject execution problems and apply `CloudSeer` to detect them. Table 4 shows these injection points and their major responsibilities for tasks in the scope of our experiments. These injection points are potentially error-prone because of network and I/O, and they reflect general and typical failure patterns that have been well studied [17, 19, 20]. During the experiment, we enable these injection points one at a time and set every enabled injection point with a 25% chance to trigger an execution problem. The problem is chosen randomly by the injection point from: (a) delaying execution by a significant amount of time to simulate a performance problem, (b) aborting the current thread to simulate an unexpected exception, and (c) ignoring a network-based request or returning an incorrect file-system status, to simulate network or I/O specific problems. We use the workload generator to keep running tasks until each injection points triggers 10 execution problems. Then we examine whether the problems reported by `CloudSeer` match with the injected ones.

5.4 Checking Accuracy

Since there are nearly 15,000 log sequences from the experiment, it is impractical to manually determine whether `CloudSeer` accurately checks each of them with the correct automaton instance. Instead, we approximate the checking accuracy using the following formula:

$$Accuracy = 1 - \frac{\#\ of\ Not\text{-}Accepted\ Sequences}{\#\ of\ Interleaved\ Sequences}$$

We only use the number of interleaved log sequences from parallel task executions in the denominator. False temporal dependencies are the only cause of checking inaccuracies in sequential executions, because other checking heuristics are not in effect. As our task modeling is sufficient for sequential executions, we exclude them from the accuracy calculation for a close approximation. Since we cannot precisely control whether and how OpenStack executes parallel task requests, we estimate the number of interleaved sequences caused by parallel executions using the number of automaton instances simultaneously tracked by `CloudSeer`.

We determine the number of not-accepted sequences by counting automaton instances that are not in accepting states. We also compare the number of instances of each task with the number of tasks populated by the workload generator. If they do not match, we manually check related automaton instances to identify accepted instances that may happen to accept messages from multiple sequences, and we count them as not-accepted ones. However, we cannot identify the case where an accepted instance may happen to take messages from multiple sequences of the same kind of task.

Table 5 presents the experiment results. The column "Acc. Range" gives the accuracy ranges showing the worst and the best results in each experiment group, and the column "Median" gives the median values. The column "% Interleaved" shows the average percentages of interleaved sequences in each experiment group. For the groups 1 and 4, the percentages are of interleaved sequences from at least two executions being in parallel (shown by the first percentage in each row). For the groups having more than two users, we also show the percentages of interleaved sequences from at least three and four executions being in parallel (shown by the second and the third percentages in each row).

The results suggest that `CloudSeer` is accurate enough to check most interleaved log sequences. Among six groups of experiments, the worst accuracy is 92.08% in one experiment of the group 6, while two experiments in the groups 1 and 2 result in a perfect accuracy. Based on the data in Table 5, we find no substantial links between the accuracy and the number of paralleled task executions or the diversity of identifiers. On the other hand, there are about 271 sequences that are for sure not accurately checked by `CloudSeer`. They end up not being accepted or being accepted by wrong automaton instances. Such inaccuracies are caused by message-interleaving patterns that invalidate the heuristics on identifier sets and divergence recovery.

5.5 Efficiency

Table 6 presents the results of efficiency by measuring the checking time for each data set listed in Table 3. The column "Ave. Msgs" shows the average number of messages produced by each experiment group. The columns "Ave. Time" and "Ave. 1k" present the average checking time and the average time per 1000 messages in each experiment. The column "% Decisive" presents the percentage of decisive checking led by the identifier-based heuristic, i.e., case (1) in Algorithm 2, among all cases. These results reflect the throughput of the checking algorithm of `CloudSeer` (Algorithm 2), but not the overhead of Logstash delivering logs from each server node to the centralized location.

Injection Point	Tasks	D	A	S	Detected	F/P	F/N
AMQP-Sender	45	3	3	4	9	0	1
AMQP-Receiver	85	3	5	2	10	1	0
Image-Create	256	3	5	2	*10	3	1
Image-Delete	240	1	3	6	8	3	2
WSGI-Client	208	3	3	4	10	3	0
WSGI-Server	87	5	3	2	8	1	**2

*. `CloudSeer` caught an unplanned and unexpected performance problem.
**. One of the injected problems does not reflect on logs.

Table 7. Detection Results

Overall, the results suggest a satisfactory checking efficiency regarding the throughput and reveal two key observations that are related to the achieved efficiency. First, the identifier-based heuristic is effective in improving the overall checking efficiency. The higher the percentage of decisive checking is, the less the checking time is needed. In particular, comparing groups with similar numbers of log messages, e.g., groups 1 and 4, we conclude that the throughput is proportional to the percentage of decisive checking. Second, both the number of parallel executions and the diversity of identifiers can affect the efficiency. The diversity of identifiers directly affects the chance of applying the expensive brute-force and divergence-recovery algorithms, as the throughputs of groups 4, 5, 6 are lower than those of groups 1, 2, 3. Subtly, the number of parallel executions affects the number of candidate identifier sets, which further increases the chance of not having decisive checking in the first place.

5.6 Capability of Problem Detection

Table 7 presents the detection results. The column "Tasks" shows the number of tasks running for each injection point, as we keep running OpenStack tasks until each point triggers 10 problems. The execution problems come randomly from three types: column "D" for delaying execution, column "A" for aborting execution, and column "S" for network or I/O specific problems. The column "Detected" shows the number of true problems detected by `CloudSeer`. The columns "F/P" and "F/N" are false positives and false negatives.

Among all 60 injected problems, 17 of them have related error messages. The other 43 problems do not lead to any error messages. Within these 43 problems, 25 of them are execution failures, and the other 18 are performance problems that are not supposed to generate any error messages.

`CloudSeer` detects 16 problems by the error-message criterion and 38 problems by the timeout criterion. These results show a precision of 83.08% and a recall of 90.00%. To verify true positives and investigate causes of false positives and false negatives, we manually check relevant automaton instances produced by `CloudSeer` and the task-submission logs by the workload generator. We also check the injection history that records injection time and VM information.

We identify two major factors affecting the precision and recall of `CloudSeer` reporting execution failures. The checking accuracy is by nature a major reason, which causes both false positives and false negatives. In Table 7, eight

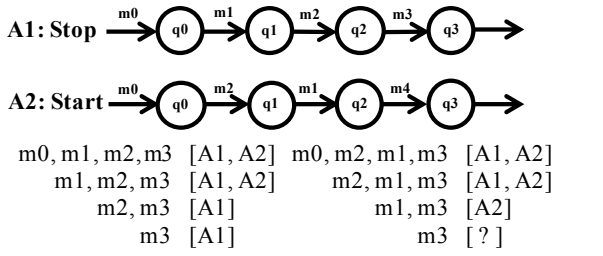

Figure 5. Normal and Reordering Cases

false positives and two false negatives are the results of inaccurate checking. We find that the major cause of such inaccuracies is message ordering missed by task automata due to insufficient modeling.

Figure 5 presents a simplified but representative case of message reordering. $A1$ and $A2$ represent automaton excerpts for the tasks "Stop" and "Start." Both automata can consume the messages m_1 and m_2, but in different orders. The concrete messages for m_1 and m_2 are "`Lifecycle event`" from a callback registered with the VM hypervisor and "`Instance destroyed`" from the `nova-compute` service. For a normal log sequence m_0, m_1, m_2, m_3 (bottom left of Figure 5), `CloudSeer` would choose $A1$ as the automaton for the sequence after checking messages followed by m_3. However, $m_1 \rightarrow m_2$ turns out to be a false dependency, where the two messages could be sometimes reordered when the server workload increases (bottom right of Figure 5). Coincidentally, it happens that $A2$ is able to recognize the reordered messages. Therefore, `CloudSeer` would keep $A2$ instead of $A1$ for the sequence without triggering divergence-recovery strategies. Consequently, $A2$ could trigger a timeout because the expected message m_4 never comes, leading to a false positive. Meanwhile, `CloudSeer` may find another automaton instance to consume the unhandled message m_3, leading to a potential false negative if the instance should have triggered a timeout. A straightforward mitigation of such reordering is to involve manual efforts in refining the task automata once false dependencies are identified during the checking process.

In addition, multiple error messages for a single execution problem lead to three false positives and three false negatives. Our error-message criterion assumes that each execution problem would be associated with only one error message. However, we find that occasionally different services may generate error messages at the same time for the same injected problem. In such cases, after associating an error message with an automaton instance, `CloudSeer` stops choosing that instance to check any other messages. Consequently, if identifier values in the extra error messages match with existing identifier sets, `CloudSeer` may find other automaton instances to associate with, which leads to false positives. Any falsely associated automaton instances would lose the capability of detecting problems of their own log sequences, so potentially lead to false negatives.

6. Related Work

In essence, we contribute an approach that monitors interleaved logs and reports potential execution problems in an automated lightweight manner. The approach uses the task automaton as the workflow abstraction to provide context information out of unstructured logs for further problem diagnosis. Based on our main contribution, we differentiate our approach from existing work as follows.

Mining workflow models from logs. Recent work [14, 15, 22, 23, 28] uses logs or traces to create workflow models for software testing, debugging, and the understanding of system behaviors. Notably, CSight [14] creates finite state machines by mining temporal invariants in logs of concurrent systems. Lou *et al.* [23] propose an intuitive automaton model and corresponding mining algorithms for reconstructing concurrent workflows from event traces.

Compared to such work, `CloudSeer` specifically addresses the log-monitoring problem using workflow models. To make workflow models suitable for both monitoring and system-behavior understanding that helps problem diagnosis, we adapt the idea of mining workflow models by tailoring such models to be lightweight specifications (i.e., task automata). With the tailored models, `CloudSeer` can effectively and efficiently check interleaved log sequences for the purpose of monitoring; meanwhile, it can also output monitoring results represented in workflow models that are understandable by administrators. Furthermore, we add on-the-fly refinement to task automata in order to mitigate false dependencies from mining and message-delivery delays in the production environment.

Log analysis for distributed systems. Some log-analysis approaches apply data-mining and machine-learning techniques on logs of distributed systems [16, 18, 21, 24, 25, 29]. Some of these approaches also focus on workflow-based analysis. Fu *et al.* [18] proposes an approach that learns workflow models from unstructured logs and uses the models to detect performance anomalies in new input logs. The Mystery Machine [16] relies on a tremendous amount of log data to infer dependency models for critical-path analysis.

Compared to these approaches, `CloudSeer` addresses the following two key points. First, in the monitoring scenario that `CloudSeer` targets, the log-message set for analysis keeps growing and changing. Such dynamics make these techniques tend to be unstable and inefficient to produce results, because they rely on existing offline logs in which there are sufficient data points representing both normal and abnormal executions. On the other hand, `CloudSeer` separates the analysis into offline and online phases. The offline phase produces task automata as stable models for the ground truth of correct-execution logs. Then the online phase leverages the models to identify partial log sequences and look for potential problems along the growing of logs.

Second, some of the workflow-based approaches require a unique and global identifier for each request. As we discussed in Section 2, although the use of identifiers in logs is ubiquitous, there is usually no single and unique identifier that is propagated across all distributed components. In `CloudSeer`, we lift the requirement of such unique and global identifiers by introducing the identifier-based heuristics. The heuristics work on-the-fly to associate multiple identifiers that should belong to the same log sequence, but are used by different distributed components.

In addition, lprof [32] is a request-flow profiler for distributed systems. It reconstructs request flows from logs and assists the diagnosis of performance anomalies. lprof relies on static analysis on Java bytecode to identify logging information, such as request identifiers for attributing log messages to flows. To the contrary, `CloudSeer` is language-agnostic and complements lprof on systems where static analysis is tedious or difficult to implement.

Log-based failure diagnosis. Insight [26] and SherLog [30] use error logs and other artifacts, such as source code, to infer or reproduce failure paths. These failure-diagnosis techniques work on a single-thread or single-request basis. `CloudSeer` may be beneficial to these techniques when being applied to complex systems, because `CloudSeer` can provide abstracted log sequences including erroneous ones from parallel task executions.

Distributed tracing frameworks. There are frameworks targeting at tracing large-scale distributed systems [4, 12, 13, 27]. These frameworks require instrumentation in specific system components, e.g., network library, to generate traces for further analysis. Compared to these frameworks, `CloudSeer` purely works on logs that are readily available in existing running systems, so it does not require any special instrumentation. Since logging is a common practice in developing large-scale systems, `CloudSeer` is highly usable on existing production systems with general logs.

7. Conclusion

We present `CloudSeer`, a lightweight non-intrusive approach for log-based workflow monitoring in cloud infrastructures. `CloudSeer` effectively and efficiently checks interleaved log sequences for execution problems with or without error log messages. It outputs task-automaton instances representing possible erroneous sequences as hints for further diagnosis. Our experiments on OpenStack show that `CloudSeer`'s checking accuracy is high on interleaved logs, with a satisfactory efficiency for the online monitoring scenario. The experiment with injected execution problems further shows that `CloudSeer` is capable for online execution-problem detection.

Acknowledgments

We would like to thank anonymous reviewers for their thoughtful feedback that helps us improve this paper and Tao Wang for proofreading.

References

[1] 2013 Path to an OpenStack-Powered Cloud Survey Results Highlight Aggressive OpenStack Adoption Plans by Enterprises. http://www.redhat.com/en/about/press-releases/2013-path-to-an-openstack-powered-cloud-survey-results-highlight-aggressive-openstack-adoption-plans-by-enterprises.

[2] Amazon CloudWatch. https://aws.amazon.com/cloudwatch/.

[3] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/.

[4] Apache HTrace. http://htrace.incubator.apache.org/.

[5] Architecture. OpenStack Installation Guide, http://docs.openstack.org/havana/install-guide/install/apt/content/ch_overview.html.

[6] CirrOS: A Tiny Cloud Guest. https://launchpad.net/cirros.

[7] Elasticsearch. http://www.elasticsearch.org/overview/elasticsearch/.

[8] Logging and Monitoring. OpenStack Operations Guide, http://docs.openstack.org/openstack-ops/content/logging_monitoring.html.

[9] Logstash. http://www.elasticsearch.org/overview/logstash/.

[10] Microsoft Azure. http://azure.microsoft.com/en-us/.

[11] OpenStack. http://www.openstack.org/.

[12] Zipkin. http://zipkin.io/.

[13] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[14] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy. Inferring Models of Concurrent Systems from Logs of Their Behavior with CSight. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 468–479, New York, NY, USA, 2014. ACM.

[15] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, and T. E. Anderson. Mining Temporal Invariants from Partially Ordered Logs. *ACM SIGOPS Operating Systems Review*, 45(3):39–46, Jan. 2012.

[16] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 217–231, Berkeley, CA, USA, 2014. USENIX Association.

[17] T. Do, M. Hao, T. Leesatapornwongsa, T. Patana-anake, and H. S. Gunawi. Limplock: Understanding the Impact of Limpware on Scale-Out Cloud Systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 14:1–14:14, New York, NY, USA, 2013. ACM.

[18] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, pages 149–158, Washington, DC, USA, 2009. IEEE Computer Society.

[19] P. Joshi, H. S. Gunawi, and K. Sen. PREFAIL: A Programmable Tool for Multiple-Failure Injection. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 171–188, New York, NY, USA, 2011. ACM.

[20] X. Ju, L. Soares, K. G. Shin, K. D. Ryu, and D. Da Silva. On Fault Resilience of OpenStack. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 2:1–2:16, New York, NY, USA, 2013. ACM.

[21] K. Kc and X. Gu. ELT: Efficient Log-based Troubleshooting System for Cloud Computing Infrastructures. In *2011 30th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 11–20, Oct 2011.

[22] D. Lo, L. Mariani, and M. Pezzè. Automatic Steering of Behavioral Model Inference. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 345–354, New York, NY, USA, 2009. ACM.

[23] J.-G. Lou, Q. Fu, S. Yang, J. Li, and B. Wu. Mining Program Workflow from Interleaved Traces. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, pages 613–622, New York, NY, USA, 2010. ACM.

[24] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li. Mining Invariants from Console Logs for System Problem Detection. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 24–24, Berkeley, CA, USA, 2010. USENIX Association.

[25] K. Nagaraj, C. Killian, and J. Neville. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 26–26, Berkeley, CA, USA, 2012. USENIX Association.

[26] H. Nguyen, D. J. Dean, K. Kc, and X. Gu. Insight: In-situ Online Service Failure Path Inference in Production Computing Infrastructures. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 269–280, Berkeley, CA, USA, 2014. USENIX Association.

[27] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, Inc., 2010.

[28] N. Walkinshaw and K. Bogdanov. Inferring Finite-State Models with Temporal Constraints. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 248–257, Washington, DC, USA, 2008. IEEE Computer Society.

[29] 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, pages 117–132, New York, NY, USA, 2009. ACM.

[30] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: Error Diagnosis by Connecting Clues from Runtime Logs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 143–154, New York, NY, USA, 2010. ACM.

[31] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 293–306, Berkeley, CA, USA, 2012. USENIX Association.

[32] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm. lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 629–644, Berkeley, CA, USA, 2014. USENIX Association.