GRETEL: Lightweight Fault Localization for OpenStack

Ayush Goel*
IBM Research

Sukrit Kalra*
IBM Research

Mohan Dhawan
IBM Research

ABSTRACT

Like any other distributed system, cloud management stacks such as OpenStack, are susceptible to faults whose root cause is often hard to diagnose and may take hours or days to fix. We present GRETEL, a system that leverages non-intrusive system monitoring, to expedite root cause analysis of both operational and performance faults manifesting in OpenStack operations. GRETEL uses unique operational fingerprints to quickly identify faulty operations at runtime. GRETEL is accurate in its diagnosis, and achieves >98% precision in identifying the faulty operation with very few false positives even under conditions of stress. GRETEL is lightweight and orders of magnitude faster than prior work, sustaining a throughput of ~77 Mbps.

Keywords

Fault localization; Network monitoring; OpenStack

1. INTRODUCTION

Cloud management stacks (or CMSes) like Apache's CloudStack [1], VMware's vSphere [19] and OpenStack [9] are complex distributed systems. Current operational and performance fault localization is both heavyweight and time consuming, even for skilled developers/operators. Recent faults [4,5,10–12,18] in Rackspace's OpenStack based cloud offering took hours and days to fix. In this paper, we focus on the problem of fast and lightweight fault localization and expedited root cause analysis in OpenStack.

Software and human issues are one of the major reasons for faults arising in cloud systems [31]. Most of the prior art [20, 21, 25–29, 33, 34, 37, 38, 41–45, 47] focuses either on active software instrumentation, passive monitoring or log analysis to detect and diagnose problems. All these techniques offer low-level diagnosis, typically reporting errors in specific API executions or listing dependencies on possible softwares or operations. However, all these solutions are often slow in reporting the fault and are not real time. For example, log analysis requires collation of event logs and subsequent parsing to reveal the exceptions or errors occurring at runtime; all of which takes significant time. Further, state of the art passive monitoring systems report an operational latency of the order of several seconds [43]. Lastly, the above systems mostly target operational faults, i.e., faults in API executions, and do not consider diagnosis of performance issues.

We present the design and implementation of GRETEL—a system for fast and lightweight fault localization of both operational and performance issues in OpenStack. GRETEL monitors relevant network communication and distributed system state, and systematically analyzes the information for fault diagnosis. Specifically, GRETEL builds a precise sequence of APIs, which identify an operation (called the operational fingerprint), of possible administrative tasks in OpenStack a priori, and leverages it to detect operations with faults. It then combines this knowledge of the faulty operations with additional passively collected system state to expedite fault analysis at runtime. GRETEL is novel because it (a) detects root cause of failures (both operational and performance) without requiring any expensive system modification, (b) identifies the high-level administrative operation responsible for the fault, and (c) requires no change even on introduction of new system components.

GRETEL builds upon the key observation that the various OpenStack components interact using a finite set of API interfaces. Note that additions or updates to these APIs is very infrequent, even across different OpenStack versions. Thus, for all practical purposes, there exist a finite set of high-level administrative operations. Furthermore, this set of operations is effectively captured in the OpenStack integration tests, which exercise several real world and relevant combinations of OpenStack APIs. GRETEL leverages these integration tests to create, a priori, a fingerprint for all these OpenStack operations, which build a sequence of REST and RPC invocations for each high-level administrative task. Thus, faults in OpenStack API invocations not covered by the operational fingerprints will not be diagnosed.

GRETEL localizes operational faults by analyzing relevant

*Both authors contributed equally.

CoNEXT '16, December 12-15, 2016, Irvine, CA, USA
 ISBN 978-1-4503-4292-6/16/12…$15.00
DOI: http://dx.doi.org/10.1145/2999572.2999600

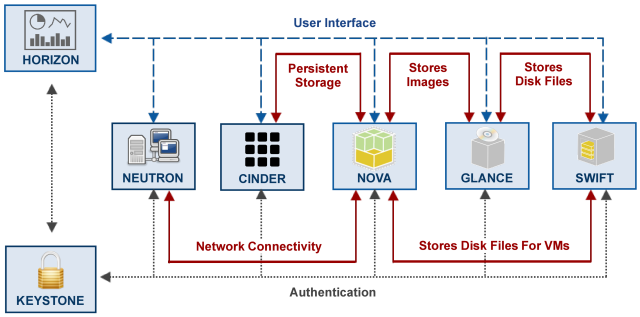


Figure 1: OpenStack architecture.

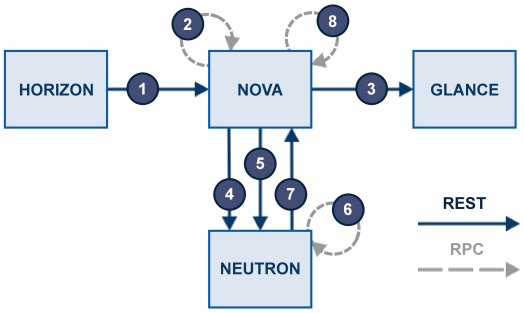


Figure 2: Workflow for launching a new instance.

OpenStack REST and RPC messages for API return errors and anomalous API latencies. To do so, GRETEL leverages the set of operational fingerprints to determine the high-level administrative task and subsequently, the OpenStack component that caused the fault. GRETEL then analyzes each network interaction amongst the various components decorated with operational and performance metadata, like per-API latency, availability of external dependencies, and resource utilization, for anomalies to expedite root cause analysis of the fault. Note that abnormal resource utilization that does not affect API latencies is not identified as a fault.

We have built a prototype of GRETEL for OpenStack LIBERTY. GRETEL's use of well defined patterns to detect API error codes for operational faults, and passive monitoring to detect API latencies, makes it compatible with all OpenStack versions, and potentially other cloud orchestration platforms as well. We evaluated GRETEL on a physical setup with 3 compute nodes, using OpenStack's Tempest integration test suite [15] with over 1600 real world scenarios, and exercised 470 faulty high-level administrative tasks in our setup. We further stress tested GRETEL and observed that for a fault frequency of 1 in $2K$ messages, it can process about 50K REST/RPC events per second of control traffic, or ~77 Mbps, which is orders of magnitude higher than prior work [43].

This paper makes the following contributions:

(**1**) We present GRETEL—a system that analyzes relevant network messages and distributed system state to localize faults and expedite root cause analysis in OpenStack operations, without requiring any system modifications. GRETEL also determines the high-level administrative task in progress that potentially led to the fault.

(**2**) We list scenarios (§ 3 and § 7) where existing tools are slow or mislead even skilled developers and operators.

(**3**) We provide a practical design for GRETEL (§ 4 and § 5), which leverages precise operational fingerprints to identify the faulty operation even at high throughputs.

(**4**) We apply GRETEL on OpenStack (§ 6), and evaluate it (§ 7) to demonstrate its accuracy, precision and speed, even under conditions of stress.

2. BACKGROUND

OpenStack is a state of the art cloud management stack, written in over 2.5 million lines of Python. It is a complex distributed system, where each high-level administrative task involves several cross component interactions.

Fig. 1 shows a conceptual architecture of a basic OpenStack deployment, where each of these services run on different nodes with distinct IP addresses, and provide a command-line interface (CLI) to enable access to their APIs via REST calls. **Horizon** is OpenStack's web-based dashboard, and provides an administrative interface to other services. **Keystone** is OpenStack's identity service, which provides authentication for all other OpenStack services. **Neutron** provides virtual networks as a service between devices managed by other OpenStack services. It allows users to create their own networks and then link them to the devices of their choice. **Cinder** is OpenStack's block storage service that provides persistent storage for VMs hosted on the cloud. **Nova** provides a controller for orchestrating cloud computing tasks for OpenStack, and supports a variety of virtualization technologies, including KVM, Xen, etc. **Glance** provides image services for OpenStack, such as a catalog and a repository for VM images using REST calls. **Swift** is an object/blob store, which enables creation, modification, and retrieval of objects via REST APIs.

COMMUNICATION. OpenStack mandates that all inter-service communication, such as those between Nova and Neutron happen via well defined REST calls. Thus, each OpenStack component has a corresponding HTTP client that enables access to its REST APIs. In contrast, all intra-service communication takes place exclusively through RPCs. Since OpenStack services could be distributed across several nodes, all RPC messages are routed through the RabbitMQ broker. For example, communication between the Nova controller and Nova agents on the compute nodes happens via RPCs channeled through the RabbitMQ broker.

DEPENDENCIES. All data processed or operated upon by OpenStack services is stored and managed by MySQL. OpenStack has communication dependencies on Python-based HTTP clients for REST calls, and Python's `oslo` messaging module for RPC invocations using RabbitMQ. Lastly, OpenStack relies upon virtualization solutions like KVM and Xen, and mandates that all nodes be NTP synchronized. If any of these dependencies are not satisfied, administrative operations in OpenStack would experience either operational or performance faults, or both.

2.1 Example: Launch a new VM

We now briefly describe the sequence of interactions amongst the various OpenStack component services to launch a new VM instance from the dashboard. Fig. 2 shows

a schematic workflow for the same. For the sake of brevity, we omit all invocations to Keystone for authentication.
(**1**) When an administrator initiates the launch of a VM from the dashboard, Horizon issues an HTTP `POST` to Nova (using the `novaclient`) to create a VM for the specified tenant.
(**2**) At this instant, the control migrates from the Nova controller to the `nova-compute` service on the compute node, where it initiates RPCs to build the instance.
(**3**) Nova then leverages the `glanceclient` to issue an HTTP `GET` request to Glance to fetch the desired VM image and initiates the boot process.
(**4**) Concurrently, Nova issues a series of `GET` requests to the Neutron controller node (leveraging the `neutronclient`) to determine the existing network, port and security group bindings for the specified tenant.
(**5**) Nova temporarily halts the boot process, and invokes Neutron APIs requesting it to create and attach a new port for the VM instance. Nova sends the desired VM and network identifiers within the `POST` request body to Neutron.
(**6**) Neutron immediately responds with the newly created port identifier for the VM. Subsequently, Nova makes the request to attach the port to the VM and blocks the boot process while waiting for a callback from Neutron, indicating that it has plumbed in the virtual interface.
(**7**) Upon completing port attachment to the VM, Neutron sends out a `POST` to Nova using the `novaclient`.
(**8**) Nova boots the VM when all events have been received.

3. MOTIVATION

Faults may arise in OpenStack operations due to incorrect component configuration, third party dependencies, request/response latencies, resource exhaustion, etc. In this paper, we consider a *fault* as a manifestation of deviant system behavior, whether operational or performance or both. Operational faults include API error responses, while performance faults include abnormal API latencies that impact currently executing OpenStack operations. Operational faults are detected using lightweight regular expression checks, while inordinate API latencies are detected using online anomaly detection tools. Note that we do not consider operational or performance issues arising in VM instances booted atop OpenStack, as faults.

FAULT LOCALIZATION. OpenStack operations involve complex interactions amongst its distributed components. While API errors or anomalous performance metadata are indicative of some fault, they can often be misleading [43]. Further, in several scenarios, these fault notifications may even be absent altogether, forcing developers/operators to mine the system logs to determine the actual source of the problem. Moreover, the actual root cause may be further upstream than the component exhibiting the fault.

ROOT CAUSE ANALYSIS USING METADATA. While it is imperative to detect what operation resulted in failure, mere fault isolation does not reveal the exact cause of the problem. Moreover, debugging the component involved requires extensive domain knowledge. However, we observe that faults afflicting OpenStack operations, typically perturb some state on the physical node, atop which the OpenStack component is running. These perturbations in system state can often provide evidence of the root cause of failure.

Root cause analysis either through log analysis or software instrumentation [22, 27, 29, 33, 34] often requires complete knowledge of the entire system. Moreover, its success depends on the intrusiveness of the implementation. Effectiveness of log analysis is limited by the verbosity and precision of the log level, while software instrumentation often requires source level changes, with subsequent compilation and deployment.

3.1 Representative Scenarios

We now illustrate the effectiveness of GRETEL's approach with the help of three representative scenarios. We show how GRETEL's root cause analysis using metadata, assists developers and operators. We also argue why state of the art log analysis and passive monitoring tools, such as HANSEL [43], do not suffice for root cause analysis in production environments on the scenarios discussed.

3.1.1 VM create

In a particular VM create scenario, with no compute nodes available, we observed that Horizon first schedules the VM for creation and then abruptly reflects a "No valid host was found" error on the dashboard.

DIAGNOSIS WITH EXISTING TOOLS. Analysis of the Nova logs with log level set to `ERROR` reveals no errors. However, setting the log level to `WARNING` simply reflects the same error as on the dashboard. We next used HANSEL and observed that it identifies a chain of messages that led to the error [43]. Specifically, HANSEL traces that the `POST` request from Horizon to Nova succeeds, but the subsequent `GET` to determine the status of the VM fails with the error mentioned above. However, HANSEL's diagnosis stops at that stage and does not reveal the true reason for the error.

GRETEL'S DIAGNOSIS. GRETEL leverages its set of operational fingerprints to detect that the failed operation corresponded to a VM creation. HANSEL, in contrast, does not indicate the operation; it merely detects the sequence of API invocations. GRETEL, then systematically leverages the available distributed state (as will be discussed in § 5.4) to indicate that the `nova-compute` service on all the compute hosts is down, which is the true reason for the error.

3.1.2 API bottlenecks

When creating several VM instances in parallel, we observed significant delays in the completion of the operation. The operation eventually succeeds but is delayed, and thus constitutes a performance fault.

DIAGNOSIS WITH EXISTING TOOLS. Log analysis yields no information even with `TRACE` level logging enabled across all OpenStack services. Since the operation succeeds and does not raise any operational error, HANSEL is not even invoked. Thus, neither of the above approaches indicate any performance issue with the operation.

GRETEL'S DIAGNOSIS. Latency anomalies are detected for two Neutron RPCs—`get_devices_details_list`

and `security_group_info_for_devices`—and thus reports a performance fault. It then leverages its operational fingerprints to identify the operation as a VM instance creation. Lastly, GRETEL checks the resource utilization on every node in the deployment and confirms anomalous CPU usage on the Neutron server, thereby isolating the performance bottleneck in the operation.

3.1.3 *Multiple parallel operations*

Production deployment of OpenStack may have several similar operations executing in parallel. For example, a client could invoke multiple VM creation operations. Even if one of those create operation fails, the administrator must accurately pinpoint the offending operation.

DIAGNOSIS WITH EXISTING TOOLS. Log analysis could identify the failed operation, only if log levels are set to `WARNING` or below. Recall that log level set to `ERROR` may not capture all failed operations. Furthermore, log analysis is slower than other runtime solutions. In contrast, parallel operations present the worst case scenario for HANSEL, which would end up reporting every operation (as per its operation stitching algorithm). Furthermore, HANSEL keeps a buffer window of 30s to avoid out-of-order or delayed messages, and also stitches operations on receipt of every message, both of which significantly increase the stitching and reporting time for faulty operations.

GRETEL'S DIAGNOSIS. GRETEL's operational fingerprints can quickly determine the failed operation, in spite of the several parallel invocations. Since GRETEL invokes operation detection only upon faults, it is not affected by any number of successful parallel operations.

4. FAULT DETERMINATION

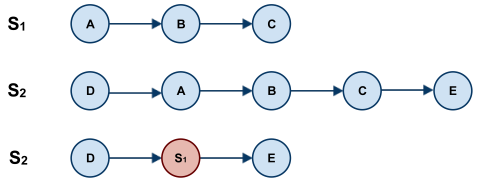
Precise root cause analysis of an OpenStack operation requires that given a (sub)set of API invocations and a fault, we must accurately identify the high-level administrative task responsible for the fault.

MODEL. We model OpenStack as a closed system, where all operations would continue to completion without any operational or performance issues, unless the system is acted upon by external factors, like software and resource dependencies. Software dependencies involve the set of third-party software binaries that are required for correct execution of the OpenStack operation. Resource dependencies include the system-level resources that could potentially affect execution of the operation, such as CPU load, free memory, network throughput, etc. In such a scenario, any operational and/or performance issue can be attributed to external factors.

Given the above operating model, root cause analysis can be expedited by systematic analysis of external factors. Thus, the fault determination problem reduces to correctly determining the OpenStack operation responsible for the fault. To do so, GRETEL leverages operational fingerprints available a priori, and determines the closest match to a given (sub)set of API invocations containing the fault.

COMPOSITE OPERATIONS. We consider each OpenStack operation as a temporally related sequence of REST and RPC API invocations. Like other distributed systems, OpenStack pieces together several basic REST and RPC invocations to build more complex administrative tasks. In a typical OpenStack deployment, all such complex, administrative tasks originate at the dashboard or CLI, and use relevant REST directives to initiate it. These starting REST directives then initiate further sequences of REST and RPC invocations to complete the operation.

Operations from a particular OpenStack component, say Nova, may perform similar functionality, like launching or deleting a VM instance. All such operations leverage similar set of APIs. If each API is denoted by a literal, then OpenStack operations can be represented by a context free grammar (CFG). Let S_1 be the operation to snapshot a VM instance, and S_2 be the operation to create a volume. Since VM snapshotting subsumes volume creation, the two operations could be represented as shown below:



We note that S_2 is completely subsumed by S_1. Hence, their CFG representation will have a common operation preceded or succeeded (or both) by terminals corresponding to additional operations, i.e., $S_2 \rightarrow DS_1E$.

CHALLENGE. Determining the single operation responsible for the fault, however, gets exacerbated in OpenStack primarily because several REST and RPC APIs may be used across multiple high-level administrative tasks. Occurrence of simultaneous OpenStack operations makes it challenging to segregate them.

5. GRETEL

GRETEL leverages two key observations to enable fast and precise root cause analysis in OpenStack operations.

(**1**) There exist a finite set of OpenStack APIs (both REST and RPC), and subsequently, only a finite set of high-level administrative tasks are possible.

(**2**) Since OpenStack is a closed system, faults afflicting its operations must be a result of some perturbations of the system state on the physical node, atop which the OpenStack component is running.

GRETEL leverages the first observation to build a precise fingerprint of each OpenStack operation. It uses the second observation to systematically combine (in real time) any API faults (both operational and performance) along with per component OpenStack activity, operational fingerprints, and fine-grained metadata about per node resource utilization, for fault localization and subsequent root cause analysis.

Fig. 3 shows a schematic architecture for GRETEL, which consists of a distributed setup of monitoring agents and a central analyzer service. GRETEL analyzes network messages flowing across different components in OpenStack to (a) list constituent APIs per operation, (b) gather per

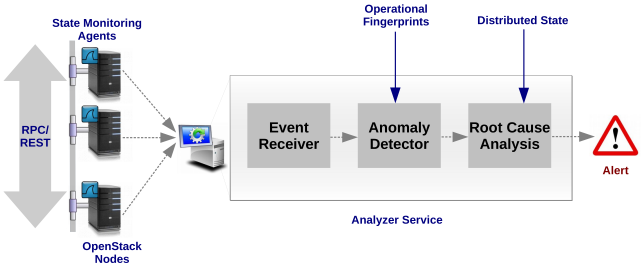


Figure 3: Schematic architecture of GRETEL.

GET_OPERATIONAL_FINGERPRINT($\mathbb{T}$)
Input: $\mathbb{T}$: List of traces for an operation.
Output: $\mathbb{R}$: Regular expression representation of the operation.

$\mathbb{T}$ = SORT_BY_TRACE_LENGTH($\mathbb{T}$);
$\mathbb{F}$ = FILTER_NOISE($\mathbb{T}$[0]);

foreach ($\tau \in \mathbb{T}$) **do**
  τ = FILTER_NOISE(τ);
  $\mathbb{F}$ = GET_LONGEST_COMMON_SUBSEQUENCE($\mathbb{F}$, τ);
end

$\mathbb{R}$ = "";
// Iterate over all the APIs in the fingerprint.
foreach ($f \in \mathbb{F}$) **do**
  if (GET_API(f) $\in$ {POST, PUT, DELETE}) **then**
    $\mathbb{R}$ = $\mathbb{R}$ + GET_SYMBOL(f);
  else
    $\mathbb{R}$ = $\mathbb{R}$ + GET_SYMBOL(f) + "*";
  end
end
return $\mathbb{R}$;

Algorithm 1: GRETEL's fingerprint generation.

API latency, and (c) determine API return errors, if any (for root cause analysis). Since, GRETEL sits transparently underneath the existing deployment, it does not affect the *safety* or *liveness* of the distributed system.

Several third-party dependencies, like `libvirt` and Python-REST clients are loaded dynamically when required by an operation. Thus, GRETEL leverages administrative help to list each of the dynamically loaded software dependencies touched upon, when executing the operation under consideration. This involves listing out the services installed during the installation of OpenStack. In the absence of such a list, GRETEL may be unable to pinpoint the root cause of faults arising from these services. Other software dependencies for OpenStack, like NTP, RabbitMQ and MySQL, are typically standard across all operations and components. GRETEL also works seamlessly with third-party vendor specific plugins as long as an agent for these plugins is provided in OpenStack.

FINGERPRINTING OPERATIONS. To fingerprint operations, GRETEL executes OpenStack in a controlled setting to avoid interference and determine the list of all APIs and set of software dependencies per operation.

GRETEL analyzes relevant OpenStack network packets, i.e., traffic corresponding to REST and RPC communication, to determine the most precise sequence of APIs that identify an operation (also known as the fingerprint).

Routine OpenStack operations typically involve several messages, both REST and RPC, that do not contribute in any meaningful way to segregate user-level operations at run time. These messages include heartbeat and status update RPCs, common REST invocations involving Keystone, and repeat occurrences of idempotent REST actions for a specific URI. GRETEL identifies such messages and removes them from the operational fingerprint to improve its quality. GRETEL further prunes the fingerprint by re-executing each operation several times and considering only those common APIs as the operational fingerprint that occur in each of the successful iterations. This mechanism removes any inadvertently captured transient REST or RPC invocations. If the re-execution of an operation leads to a different set of API invocations, GRETEL only considers the common set of APIs as a fingerprint of the operation. As will be discussed later in § 5.3.1, GRETEL prioritizes state change operations to improve the precision of the fingerprints being generated. Algorithm 1 briefly lists these steps.

5.1 Distributed State Monitoring

GRETEL uses monitoring agents at each OpenStack node in the deployment to monitor (a) relevant OpenStack REST and RPC communication, (b) resource utilization, and (c) health of OpenStack dependencies at each node. The network monitoring agents determine the observed network latency between request and response for each network message. The resource monitoring agents periodically poll the host nodes for CPU, memory, network throughput, storage, and disk read/write behavior. Additionally, GRETEL maintains watchers on third-party software dependencies.

5.2 Event Receiver

The event receiver receives relevant OpenStack traffic from every node, and forwards them to an anomaly detector to detect operational and performance faults in operations. Complex OpenStack tasks are essentially a combination of simpler operations consisting of REST and RPC messages, which must occur in a specified order to complete the task. Thus, while messages may be interleaved across different operations or delayed, they will not occur out of order for a successful operation. Further, TCP connections from network monitoring agents to GRETEL ensure that order of messages along a TCP stream is preserved.

5.3 Anomaly Detector

The anomaly detector has two main tasks. First, it analyzes each message to determine an anomaly. Second, based on that anomaly, it determines the operation which potentially contains the offending message.

DETECTING FAULTS. An API error return value signifies an operational error, while an inordinate API latency indicates a performance issue. Detecting operational faults is straightforward for RESTs, where errors are indicated in the HTTP response header. For RPCs, however, diagnosing faulty messages requires domain-specific knowledge of OpenStack so that error patterns are accurately identified in the message body. To ensure that anomaly detection for operational faults remains lightweight, GRETEL does not parse the JSON formatted message body and simply uses regular expressions to identify error codes in the message.

API latencies are computed based on timestamps

associated with each message, and GRETEL leverages available online outlier detection tools to detect performance faults. REST latencies are computed by pairing request and response messages based on TCP connection metadata, like IP and port, while RPC latencies are computed using IP and message identifier that is unique to each pair. With such lightweight message analysis, anomaly detection proceeds at high throughput rates.

5.3.1 Operation detection

GRETEL maintains a sliding window of size α to isolate the possible faulty OpenStack operation. We define α as

$$\alpha = 2 * max\{FP_{max}, P_{rate} \times t\}$$

FP_{max} is the size of the largest fingerprint across all OpenStack operations, while P_{rate} is the rate of the incoming message stream in packets per second and t is the time in seconds. The choice of a small time interval t is shadowed by FP_{max}, whereas a bigger value of t ensures that the sliding window is big enough to determine the largest operation given a high P_{rate}. On detection of an anomaly, GRETEL slides the window ahead by $\alpha/2$ messages and waits for the event receiver to fill the remainder $\alpha/2$ of the window. This mechanism helps GRETEL to have a snapshot of both the past and the future of the faulty message, and effectively determine the operations in progress at the time of the fault.

Once the sliding window is full, GRETEL spawns a new thread to detect the faulty operations in progress. To do so, GRETEL first determines the set of operations that include the offending message in their API sequences and then uses regular expressions for these operations (derived from the CFGs as per § 4) to detect a match in the snapshot.

If at least one such match is detected, the anomaly detector forwards the (a) set of operations that matched the snapshot, (b) set of messages (both REST and RPC) corresponding to these matched operations, and (c) source and destination IPs for all the error messages, to the root cause analysis engine.

PROBLEM. The sliding window may, however, include APIs from multiple different operations, where every API can potentially be part of several other administrative tasks. Thus leveraging the entire sliding window to narrow down the faulty operation may yield poor results.

SOLUTION. GRETEL, therefore, uses a context buffer atop this sliding window that dynamically adjusts itself to determine the minimum context required for identifying the OpenStack task and prune away any noise that could affect precise operation determination. Specifically, GRETEL starts with a small fixed size window $\beta = c_1\alpha$ and determines the precision θ of matching with the operational fingerprint, where c_1 is empirically determined.

We define GRETEL's precision θ of detecting unique operations per fault as:

$$\theta = (N - n)/(N - 1)$$

where N is the total number of operational fingerprints, and n is the actual count reported by GRETEL's operation detection mechanism. A precision of one, i.e., $\theta = 1$, indicates that the fault has been correctly narrowed down to a single operation. A value of $n > 1$ degrades the precision, with $\theta = 0$ for $n = N$.

```
GET_FAILED_OPERATIONS(𝔹, 𝔸)
Input: 𝔹: List of packets inside the context buffer.
Input: 𝔸: The offending API.
Output: 𝔽: The set of failed operations.

// Get the operations that contain the failed operation
𝕆 = GET_POSSIBLE_OFFENDING_OPERATIONS(𝔸);
𝕆 = TRUNCATE_OPERATION_FINGERPRINTS(𝕆, 𝔸);

ℙ = "";
// Build the pattern to match the fingerprints against
foreach (b ∈ 𝔹) do
  | ℙ = ℙ + GET_SYMBOL(b);
end

𝔽 = {};
foreach (o ∈ 𝕆) do
  | if (REGEX_MATCH(o, ℙ)) then
  |   | 𝔽 = 𝔽 ∪ o;
  | end
end
return 𝔽;

TRUNCATE_OPERATION_FINGERPRINTS(𝕆, 𝔸)
Input: 𝕆: Set of operations to truncate.
Input: 𝔸: The offending API.
Output: 𝕋: The given set of operations with truncated fingerprints.

𝕋 = {};
foreach (o ∈ 𝕆) do
  | i = FIND_LAST_OCCURENCE(o, 𝔸);
  | 𝕋 = 𝕋 ∪ o[0 : i];
end
return 𝕋;
```

Algorithm 2: GRETEL's anomaly detection.

At each iteration, GRETEL increments the context buffer β by $\delta = c_2\alpha$ messages on either side of the fault, where c_2 is empirically determined. GRETEL stops these iterations as soon as the precision θ drops. This also ensures that a false negative is not returned unless one of the APIs characterizing the operation is not contained within the sliding window. GRETEL starts with a small value of β so as to minimize the number of operations matched and then rapidly increases the buffer size by δ on each side of the fault to make sure that the whole sliding window α is covered as fast as possible.

IMPROVING PRECISION. Errors manifesting in RPC invocations are typically communicated back to the dashboard or CLI via REST calls. To avoid processing duplicate snapshots, GRETEL initiates the snapshot mechanism only when it detects an error in REST messages. However, all REST and RPC errors present in the snapshot are together analyzed to detect the root cause of the fault.

In case of performance faults, the operation will proceed to completion, and the snapshot is likely to match APIs even beyond the point of the observed fault. GRETEL, therefore, makes use of the entire context buffer to detect a match with the operational fingerprints. However, in case of operational errors the entire sequence of operations may not complete (due to the failed operation), and the snapshot may not match any regular expression. Thus, GRETEL truncates the regular expressions corresponding to operations that contain the offending API till its last occurrence in the fingerprint. The intuition is that the snapshot would match with one or more of these truncated regular expressions, which identifies

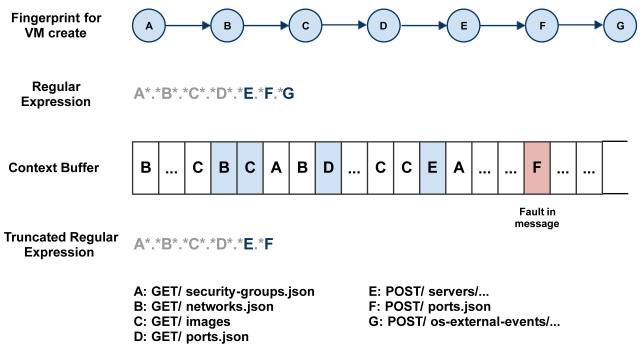


Figure 4: GRETEL's operation detection mechanism. We omit RPCs for brevity. State change APIs are marked in dark.

the potential faulty operations executing in the system. Algorithm 2 briefly lists these steps.

GRETEL also prioritizes matching of literals in the regular expression corresponding to state change operations, such as REST APIs with `POST`, `PUT` and `DELETE` modifiers and RPCs. Thus if the fault is in a state change operation, which is more likely in a dynamic system, then GRETEL generates regular expressions that are simpler and, hence, faster to match.

OpenStack is in the process of introducing a correlation identifier [8] to tie together requests and responses from different services pertaining to a single operation. When fully implemented, GRETEL can exploit these correlation identifiers to increase its precision by reducing the number of packets against which a fingerprint is matched.

EXAMPLE. The operational fingerprint for the VM create operation involves 7 REST and 3 RPC invocations. Fig. 4 lists this fingerprint, but omits the RPCs for brevity. If Nova's `POST ports.json` call to Neutron (node F) for creating and attaching a port to the VM instance fails, i.e., step 6 in § 2.1, GRETEL then creates a snapshot of the messages in its context buffer (which sits atop the sliding window) and initiates the operation detection procedure. Specifically, GRETEL converts the regular expressions corresponding to the operational fingerprints and the snapshot into strings, where each symbol represents a REST or RPC API, and isolates all fingerprints that contain the symbol F. It then truncates these regular expressions till the last seen occurrence of the symbol F. GRETEL uses these truncated regular expressions to match against the snapshot.

Due to several concurrent operations in the system, it is possible that some of the symbols even in the truncated regular expression are not captured in the snapshot. Thus, GRETEL further relaxes the notion of a fingerprint match, such that a regular expression matches the snapshot if the sequence of symbols corresponding to the state change operations, i.e., RPCs and `POST`, `PUT` and `DELETE` REST calls, is preserved. Fig. 4 presents the mechanism for the detection of a faulty VM create operation. Note that even though symbol A is missing from the context buffer, the truncated regular expression still matches as it preserves the order of E and F, corresponding to `POST servers` and `POST ports.json` invocations from Nova to Neutron.

GRETEL limits itself to analyzing request and response headers alone, and does not parse the JSON formatted payload for extracting rich operation metadata. As a result, it may miss out on matching certain operations that change their fingerprints according to the data in the payload. While GRETEL should leverage payload for more precise operation tagging, we leave this improvement for future work.

```
GET_ROOT_CAUSE(𝕆, 𝔸)
Input: 𝕆: Operation under consideration.
Input: 𝔸: Set of all dependencies.
Output: ℝ: Root cause.
Initialize: ℝ = {}
ϕ = Get_List_Of_Error_Messages_For_Operation(𝕆);
ℕ = Get_List_Of_Nodes_For_Operation(𝕆);
[nodes] = Get_Error_Nodes(ℕ, ϕ);
ℝ += Find_Root_Cause([nodes], 𝔸);
if (True == Is_Empty(ℝ)) then
  [remaining_nodes] = Get_Remaining_Nodes_In_Operation(ℕ,
[nodes]);
  ℝ += Find_Root_Cause([remaining_nodes], 𝔸);
return ℝ;

FIND_ROOT_CAUSE(ℕ, 𝔸)
Input: ℕ: List of nodes.
Input: 𝔸: Set of all dependencies.
Output: faulty: Return value indicating root cause.
Initialize: faulty: {}.
// Determine anomalies in metadata
foreach (ψ ∈ ℕ) do
    𝕄 = 𝔸['resource'][ψ];
    𝕊 = 𝔸['software'][ψ];
    foreach (ρ ∈ 𝕄) do
        if (True == Is_Anomalous(ψ, ρ)) then
            faulty += ρ;
    end
    foreach (η ∈ 𝕊) do
        if (True == Is_S/W_Dependency(𝕊, ψ, η) then
            sw = Get_Offending_S/W(𝕊, ψ, η);
            faulty += sw;
    end
end
return faulty;
```

Algorithm 3: GRETEL's root cause analysis.

5.4 Root Cause Analysis

GRETEL leverages (a) the error metadata forwarded by the anomaly detector, and (b) the knowledge of the distributed state obtained from polling individual nodes, within the duration of events captured in the context buffer (as passed on by the anomaly detector) to detect the root cause.

Algorithm 3 briefly lists the steps. Specifically, GRETEL analyzes all operations containing the offending error message (as forwarded by the anomaly detector), and uses the operational fingerprints to determine the set of nodes in the OpenStack deployment that correspond to the particular high-level administrative task. GRETEL focuses on the source and destination nodes (as passed on by the anomaly detector for the offending message), and determines anomalous resource or software usage that could potentially affect the operation. In case no anomalous values are detected on the nodes analyzed, GRETEL expands its search to other possible nodes that participate in the operation (as per the fingerprint). This is imperative since the root cause of the error in the operations may manifest upstream from the actual node where the fault arose. We

further describe, in detail, the utility of this approach in § 7.2.

IMPROVING PRECISION. OpenStack deployments typically install each of its component services on separate nodes for scalability and fault isolation. GRETEL leverages this observation to further improve the accuracy of its root cause detection. This distinction enables GRETEL to accurately track API level performance metadata per node, for every operation. Note that even if the component services share IP addresses, GRETEL would only require the various OpenStack HTTP clients to report the source service as an HTTP request header to identify the start of an operation. No further changes would be needed in the core deployment. By annotating each packet with the service name, GRETEL remains unaffected by hardware failures or upgrades that may typically change low level information about the deployment. In other words, this small change will make GRETEL oblivious to the underlying deployment.

6. IMPLEMENTATION

We have implemented a prototype of GRETEL for OpenStack LIBERTY based on the design described in § 4 and § 5. We built GRETEL's distributed state monitoring infrastructure using Bro [2, 39] and `collectd` [7]. We augmented Bro with a custom protocol parser for the RabbitMQ messaging protocol (60 LOC in C++), and leveraged Python-bindings for Broccoli [3] (Bro's communication library) to send events from the OpenStack nodes to GRETEL's analyzer service, written in ~1600 LOC in Python. We additionally installed and configured `collectd` on all OpenStack nodes to capture system metrics and send them to the analyzer service. We now briefly describe a few salient features of our implementation.

(**1**) **System state monitoring**. GRETEL performs a comprehensive check about the status of the dependencies and the resources utilized. Specifically, GRETEL analyzes `collectd` generated snapshots of the system resource usage. Additionally, GRETEL also has watchers to detect TCP-level reachability to MySQL, RabbitMQ and NTP servers, and other nodes in the deployment.

(**2**) **Anomaly detection**. GRETEL detects operational faults by analyzing the error packets, which involves lightweight regular expression checks over the message payload. However, to determine performance anomalies, GRETEL leverages Python-bindings for the R `tsoutliers` package [16]. Specifically, GRETEL uses the LS (Level Shift) mode in the `tsoutliers` to detect the outliers in the continuous stream of API latencies and resource utilization received at the analyzer. The LS mode ensures that GRETEL adapts to the underlying system changes and does not report many false alarms. Note that outlier detection in GRETEL is pluggable and administrators can leverage any sophisticated detection mechanism for the same.

OPTIMIZATIONS. GRETEL detects currently executing OpenStack operations using regular expressions. Since the number of unique OpenStack APIs is 643, we use Unicode encoding to assign a symbol to each API. However, matching hundreds of regular expressions against a Unicode-encoded message snapshot is slow. Thus, GRETEL removes symbols corresponding to RPC messages to speed up operation detection. The intuition here is that an RPC error message must also be captured in the REST message(s) to the dashboard or CLI. To further speed up operation detection, GRETEL offloads all regular expression matching to a Perl process.

Lastly, GRETEL leverages a dual buffer to receive and process the incoming REST and RPC messages. It speeds up the snapshotting process using a combination of two pointers in the dual buffer separated by α messages, where α is the size of the sliding window. Whenever an error is encountered in the message stream, GRETEL freezes the messages between these two pointers to create a snapshot.

7. EVALUATION

We now present an evaluation of GRETEL. In § 7.1, we characterize OpenStack operations and build their fingerprints, which can be used to identify operations at high throughput rates. In § 7.2, we present case studies highlighting GRETEL's utility in root cause analysis for OpenStack operations. In § 7.3, we use Tempest to evaluate GRETEL for its precision in uniquely identifying the possible operations for every fault introduced because there exists no publicly available trace, representative of real OpenStack workloads. Lastly, in § 7.4, we measure GRETEL's network and system overhead under conditions of stress.

EXPERIMENTAL SETUP. Our physical testbed consists of 7 servers (including 3 compute nodes) connected to 14 switches (IBM RackSwitch G8264) arranged in a three-tiered design with 8 edge, 4 aggregate, and 2 core switches. All servers are IBM x3650 M3 machines having 2 Intel Xeon x5675 CPUs with 6 cores each (12 cores in total) at 3.07 GHz, and 128 GB of RAM, running 64-bit Ubuntu v14.04. We installed OpenStack LIBERTY with each component on a different server. The inter-component OpenStack traffic and Bro-to-analyzer service communication was isolated to avoid any performance penalties. `collectd` frequency to poll for resource utilization was set to 1s.

EMPIRICAL DETERMINATION OF THRESHOLDS. We empirically determine the values of the thresholds used by the anomaly detector. We use Bro to determine the value of P_{rate}, which is the only dynamic parameter affecting the value of α and in turn β. As will be shown later in § 7.1, the maximum fingerprint size FP_{max} was 384 across all OpenStack operations. P_{rate} for our deployment with 400 concurrent operations was observed to be ~150 pps and we chose the value of parameter t as 1s. Thus, we set the sliding window size α to be 768 (recall § 5.3.1). We empirically determined c_1 as 0.1 and c_2 as 0.04 and thus, the start value of the context buffer β was set at 80, while δ was set at 30.

7.1 OpenStack Characterization

We characterize OpenStack by developing fingerprints for all possible operations. Specifically, we leveraged OpenStack's Tempest integration test suite [15], which includes an exhaustive set of tests representative of actual

Category	Tests	Unique APIs		Events		Avg. Fingerprint	
		RPC	REST	RPC	REST	w/ RPC	w/o RPC
Compute	517	61	195	77.2K	87.8K	100	56
Image	55	10	38	0.9K	4.8K	18	15
Network	251	24	70	20.2K	18.5K	31	16
Storage	84	11	40	3.5K	6.2K	17	15
Misc.	293	11	20	9.1K	14.1K	16	11
Total	1200	-	-	110.9K	131.4K	-	-

Table 1: Characterization of the Tempest test suite.

administrative tasks in OpenStack. We executed each Tempest test (applicable for our setup) in isolation and monitored the sequence of REST and RPC APIs reported by GRETEL, along with the software dependencies required at every node in the OpenStack deployment (per § 5.1).

WHY TEMPEST? Tempest has an extensive battery of tests for OpenStack API validation, real-world scenarios testing the integration between different OpenStack services, and other specific tests useful in validating an OpenStack deployment. Further, Tempest is designed to run against any OpenStack deployment, irrespective of the scale, the environment setting or the underlying services that the deployment uses. Since Tempest can run on large deployments, it provisions running its test cases in parallel to stress the OpenStack deployment close to the degree that it would endure during peak usage cycles.

All tests that Tempest runs and validates are tasks that a tenant or administrator can execute in a real-world setting via OpenStack APIs. Further, Tempest's functional and integration tests are complex operations, derived from actual, real-world use cases occurring daily in fully operational OpenStack deployments. Typically, these scenarios involve a series of steps for testing OpenStack functionality spanning single or multiple nodes, where complicated state requiring multiple services is set up, exercised, and torn down.

GRETEL's fingerprint generation is an offline process since these fingerprints are independent of the scale of the deployment. Hence, GRETEL does not require learning atop production environments and can be trained on test environments. Unlike operational anomalies which leverage these fingerprints generated offline, performance anomaly detection is an online process and makes use of mechanisms described earlier in § 5.3 and § 6.

TEST SUITE CHARACTERIZATION. The latest Tempest test suite has 1645 tests of which 1200 executed successfully on our setup. The rest were not applicable for our setup and thus skipped by the test harness. We further classified these tests based on the nature of the operations. All tests that include creation, migration, etc., of instances, were classified under Compute. Tests that operate upon network configuration, ports, routers, etc., were classified under Network, while those involving VM images were categorized under Image. Tests operating upon VM storage were grouped under Storage. All other tests, such as those involving management tasks, like querying for key pairs, availability zones, etc., were grouped as Miscellaneous.

We executed each of the applicable 1200 Tempest tests in isolation using GRETEL, and report our findings for each

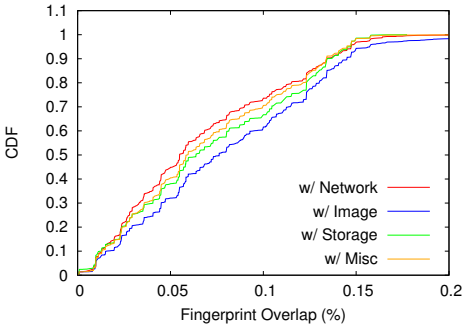


Figure 5: CDF for Compute operations.

category as described above in Table 1. TESTS indicates the count of tests executed per category. UNIQUE APIS indicate the number of unique RESTs and RPCs observed across all tests in the corresponding category. The REST and RPC events column indicate the network messages processed by GRETEL. These messages also include periodic updates, heartbeats, etc., which GRETEL prunes. The last column indicates the average fingerprint size for each category, both with and without RPCs.

We observe some overlap across operations within the same category. This overlap stems from the fact that operations belonging to the same category tend to leverage a small set of APIs provided by a single OpenStack service. However, GRETEL's fingerprints are substantially unique across operations of different categories. We select 70 representative Compute operations and plot their overlap across all other categories in Fig. 5. We note that ~90% of the representative Compute operations have <15% overlap across all categories.

LIMITATION. While Tempest aims to be exhaustive in its API coverage, we note that it is not sufficient to fingerprint 'all' possible OpenStack operations. Specifically, OpenStack components expose a total of 643 public APIs through their REST clients and CLIs. However, our characterization reveals that Tempest tests utilizes only a subset of these APIs across all its operations. Therefore, our characterization must be augmented to cover scenarios not included in the Tempest test suite.

7.2 Accuracy

We now present representative scenarios highlighting GRETEL's utility in root cause analysis of OpenStack operations. While the scenarios presented here are far from being exhaustive, they are indicative of (a) typical OpenStack operations and the nature of cross-component communications involved, and (b) GRETEL's effectiveness in isolating and identifying the root cause of faults.

7.2.1 Failed image uploads

We observed that uploading new VM images failed with Horizon showing a "Unable to create new image" error. Analysis of Glance logs revealed no entries. However, GRETEL discovered a REST 413 "Request Entity Too Large" error issued from Glance to Horizon for the `PUT` method of the `v2/images/<UUID>/file` API and

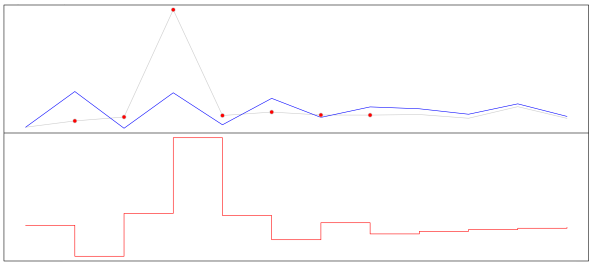


Figure 6: Anomalous latency for Neutron's `GET /ports.json`.

narrowed down the number of possible operations to just one, which was the Image upload operation. Subsequently, GRETEL's root cause analysis engine detected a resource issue on the Glance server, specifically low free disk space. After clearing up space and restarting the Glance service, subsequent image upload operations succeeded.

7.2.2 *Neutron API latency increase*

VM instance creation is one of the most common and significant operation in OpenStack, and any increase in its latency will have an adverse effect on overall system performance. During a run of 400 concurrent operations, we observed an increase in the latency of Neutron's `v2.0/ports.json` API as shown in Fig. 6. The original and adjusted time series are plotted in gray and blue respectively, while the level shift corresponding to the anomalous behavior is marked in red.

GRETEL was able to recognize this anomalous API latency increase and initiated the root cause analysis, which focuses on both Neutron and Nova controller nodes to determine the reason for the performance fault. GRETEL detects no issues with Nova, but observes a surge in the CPU utilization on the Neutron server during the execution of the operation. GRETEL attributed the increased latency to the high CPU usage. Other Neutron APIs like `v2.0/quotas/{ID}` and `v2.0/networks.json` that are part of the same VM create operation also experienced similar API latency increase.

7.2.3 *Linux bridge agent failure*

Neutron's layer 2 Linux bridge agent plugin configures a Linux bridge to realize the various Neutron abstractions. This plugin is deployed on all compute nodes and in event of its failure, the VM creation operation fails since it is not able to provide a network to the newly created instance. When launching an instance from the dashboard, we observed that Horizon reported a "No valid host was found. There are not enough hosts available." error. However, a subsequent listing of Nova's services showed that there was a compute node enabled and up and running. There were no errors on the Nova compute host, while the controller's logs showed the same error as the one on the Horizon dashboard.

GRETEL characterized this operation as a failed VM create operation and started the root cause analysis, which found no resource related anomalies on any of the servers involved in the operation. Subsequently, GRETEL checked for failures in software dependencies and found that the `neutron-plugin-linuxbridge-agent` on the compute host had crashed and reported that as a possible reason for the failure. Manual debugging revealed service misconfiguration, and fixing it completed the operation.

7.2.4 *NTP failure*

OpenStack requires the presence of NTP agents on each server that is part of the deployment to properly synchronize services amongst nodes. While doing a `cinder list` on the controller node, we noticed that the operation failed with the error "Unable to establish connection to Keystone". Log analysis found no error logs on the Keystone service and Cinder logs showed a "Timeout is too large" error; none of which were useful in identifying the root cause.

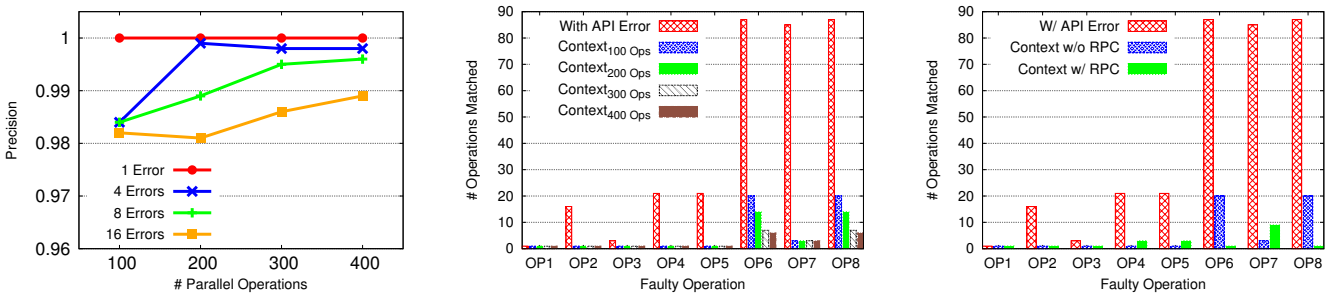
GRETEL identified a 401 Unauthorized error being relayed from Keystone to Cinder. Our root cause analysis engine checked the node hosting Keystone for both resource anomalies and software issues and found none. The engine then checked for resource related issues on the Cinder node and found no anomalies, but checking the software dependencies listed out the stopped NTP agent as a possible cause of the issue, and upon restarting the NTP agent on the host, the cinder client started working.

7.3 Precision

PARALLEL WORKLOAD. We evaluate GRETEL's precision in a controlled setting with several parallel operations. We randomly select non-faulty Tempest tests proportional to their distribution in the test suite, and execute them concurrently with a specified number of faulty test cases. These faulty operations included erroneous APIs only from the Compute and Network category, which form over 80% of all REST API invocations in the test suite (per Table 1). We ignore all performance faults for these tests, and evaluate them separately. Since each test spans several distinct operations, this setup is representative of real world, parallel operations, and is particularly challenging for our characterization that considered only individual operations executing in isolation.

We measure GRETEL's precision with varying number of parallel tests from 100 to 400 in increments of 100. For each scenario, we varied the number of operational faults injected (1, 4, 8 and 16) and plot the precision in Fig. 7a. Each fault additionally invokes operation detection against entire set of the 1200 fingerprints (corresponding to all OpenStack operations) in our characterization set.

In all scenarios, GRETEL reports a precision of >98%. We also observe that as we increase the load and the faults injected, GRETEL's precision increases only marginally. We attribute this small increase in precision to the availability of more operational context, which increases due to increase in size of the context buffer (β). Note that an increase in the number of parallel operations increases the number of interleavings between subsequent calls of the same operation. Thus, GRETEL increases the size of the context buffer (β), which forces a more precise match with the truncated regular expressions for the operational fingerprints (per § 5.3.1).



(a) Precision for parallel workload. **(b)** Operations matched w/ increasing context. **(c)** Operations matched w/ and w/o RPCs.

Figure 7: GRETEL's precision.

(**1**) **False positives:** A false positive for GRETEL occurs when it cannot narrow down the sequence of messages in its context buffer to just a single OpenStack operation. This is possible since majority of OpenStack operations involve cross-service interactions (e.g VM create, VM migrate etc.). Further, as discussed in § 7.1, GRETEL's fingerprints are substantially unique for composite operations and hence, a larger snapshot results in lower number of false positives.
Fig. 7b plots the variation in operations matched when executing 100, 200, 300 and 400 concurrent operations along with 8 faulty operations. "With API error" indicates the number of operations matched based on just the REST error API, without using the snapshot from the context buffer. We note that as the parallelism grows, GRETEL's precision improves marginally. We attribute this improvement primarily to an increase in the context buffer, whose size increases with increased parallelism in the system to improve precision (per § 5.3.1).
(**2**) **Pruning RPCs in fingerprint**: As a performance optimization, GRETEL prunes the operational fingerprint to match upon only the set of REST APIs in the operation (recall § 6). We executed 100 tests concurrently along with 8 injected faulty operations. Fig. 7c plots the number of operations matched with the fingerprint with and without RPCs included. 'With API error' indicates the count of operations matched on just the REST error API, without using the snapshot from the context buffer. We observe that the use of RPCs only marginally improves precision for some scenarios, which is possible due to a richer context available for matching with the operational fingerprints.
(**3**) **Multiple parallel faults**: We measure the number of operations matched when executing 16 parallel instances of the same faulty operation, along with varying number of concurrently executing tests from 100 to 400 in increments of 100. Fig. 8a plots the results. We observe that the average number of operations with which the fault matches decreases steadily as the concurrency increases. This is primarily due to the increase in the context buffer, where a larger context buffer size helps to match the operational fingerprint and subsequently improve precision (per § 5.3.1).
(**4**) **Performance faults**: We determine the performance faults reported for anomalous API latencies observed for Glance's `v2/<ID>/images` REST API when executing 200 Tempest operations concurrently, which took ~20 mins to complete. The `v2/<ID>/images` API is one of the most frequently invoked APIs by OpenStack services to retrieve metadata for the specified image ID. We used `tc` [13] to inject a 50 ms latency in all communication to/from the Glance server for 10 mins, starting at the 5 min mark.
Fig. 8b plots the original (in gray) and adjusted time series (in blue), and the level shift (in red) corresponding to the anomalous behavior. We observe that GRETEL raises 18 alarms during the 10 min duration and is also corroborated by the level shifts reported during the corresponding time period. Note that the adaptive nature of LS raises alarms only when there is a sudden spike in the time series values. LS does not raise alerts even if latency variations are smaller than the initial observed spike.

7.4 Performance

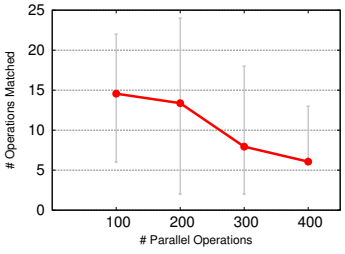
7.4.1 *Throughput*

Since the 1200 tests chosen from the Tempest test suite may not stress our deployment, we perform a synthetic evaluation of GRETEL's fault testing mechanism at scale using a large number of concurrent operations. We measure the steady throughput achieved by GRETEL for fault frequencies of 1 fault per 100, 500, 1K, 1.5K and 2K messages. We use tcpreplay [14] to generate RPC events at varying rates from the Bro agents, and plot the results in Fig. 8c. We observe that for 1 fault per 100 messages, GRETEL reports a throughput of ~7.5 Mbps. On reducing the error rate to 1 fault per 1K messages, GRETEL processes messages at near line rates. Thus, GRETEL's snapshotting mechanism is not a bottleneck.

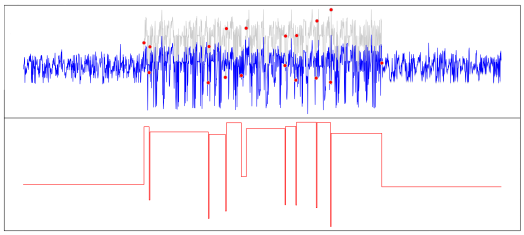
We limit the maximum packets per second (pps) rate for tcpreplay to 50K, since higher pps results in several packet retries, which affect throughput calculations. Further, GRETEL took a maximum duration of <2 seconds to report the faults even with 400 concurrent operations. In contrast, HANSEL [43] achieves a peak throughput of only 1.6K REST messages per second, and introduces a latency of 30 seconds to account for delayed or out-of-order messages.

7.4.2 *System Overhead*

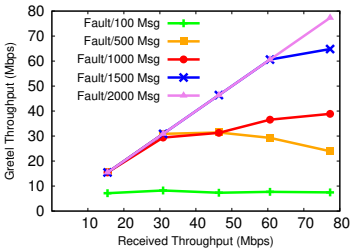
We ran 100 Tempest tests in parallel, which took ~6 mins



(a) Operations matched w/ 16 identical concurrent operations.



(b) Performance faults observed w/ parallel workload.



(c) GRETEL's throughput.

Figure 8: GRETEL's evaluation.

to complete, and monitored all nodes for CPU and memory consumption by the analyzer and the Bro agents. Note, we disabled our watchers for passive state monitoring for this experiment. The analyzer reported peak CPU usage of ~4.26% and memory consumption of ~123 MB. We observed that peak CPU usage for Bro-based monitoring agents was <12.38%, while their memory usage was ~1 GB.

8. LIMITATIONS

(**1**) GRETEL's accuracy is contingent upon the message context available in the sliding window used to determine the faulty operation. A small snapshot may include only a partial fingerprint resulting in no operation match.
(**2**) GRETEL requires the presence of a REST or RPC error message for detection of faults, and may miss out on faulty operations that do not generate any tangible operational or performance errors, like a stuck VM create
(**3**) GRETEL leverages regular expression checks to detect operational faults. Faults that do not match any regular expression may be potentially missed. For performance faults, GRETEL relies upon its watchers. Thus, transient changes resulting in performance issues, may be missed if not polled at a finer granularity.
(**4**) GRETEL's use of fingerprints for all OpenStack operation governs the precision of operation detection. Thus, more number of test scenarios would improve the fingerprint. In its present form, GRETEL may miss on operations that are not captured in the Tempest test suite, i.e., it is predicated on the completeness of the test suite.
(**5**) GRETEL cannot detect root cause of errors that manifest due to interfering operations, such as those that have causal dependencies upon each other. This limitation is, however, present across most prior art, including log analysis systems.
(**6**) GRETEL does not handle asynchronous calls that occur in the middle of an operation and lead to a branched fingerprint. Currently, GRETEL's re-execution of operations removes truly asynchronous APIs from the fingerprint.
(**7**) Enhancements to OpenStack or its APIs require building additional fingerprints for the newly introduced operations.
(**8**) GRETEL currently requires manual input to determine software dependencies for each operation, and the accuracy of its root cause analysis is contingent upon the correctness of the characterization provided.

9. RELATED WORK

9.1 Software instrumentation

Causal dependencies can be derived by comprehensively instrumenting all software, including any middleware, for communication, scheduling, and synchronization to record interaction amongst the various different components in the distributed system. Such systems [21, 25–29, 33, 37, 41, 42, 44, 45, 47] often integrate end-to-end tracing of all component interaction within the distributed system operations themselves.

X-trace [29], BorderPatrol [37], Pip [41], Webmon [30], Pinpoint [27], vPath [45] and Ju *et al.* [33], actively instrument the network stream along the software stack to uniquely tag all network operations. Popular industry implementations, like Zipkin [17] Dapper [44], and Htrace [6], also take a similar approach towards end-to-end tracing. In contrast, X-ray [21] instruments application binaries and uses dynamic information flow tracking to estimate the root cause of performance issues.

While being useful, the above systems have implementation and deployment limitations. Not only do they require exhaustive knowledge of the entire system, their success primarily depends on the intrusiveness of the implementation. Further, whole system instrumentation to track faults, requires modification of the entire source code, recompilation and subsequent deployment, which may not be possible in some deployments. Thus, GRETEL, unlike prior work, does not rely on any instrumentation. Instead, GRETEL employs a combination of precise operational fingerprints of OpenStack operations and passive monitoring of resources and dependencies to identify root cause of operational and performance issues.

9.2 Passive monitoring

GRETEL, like prior work [20, 34, 38, 43], utilizes passive monitoring of the deployment (without any knowledge of node internals or message semantics) to infer causal sequences of actions leading to faults. However, unlike prior work, GRETEL employs a learning phase to build a precise operational fingerprint of OpenStack operations, and combines it with statistical inferences from system metrics to determine the root cause of faults.

Sherlock [22], SCORE [36], NetMedic [34], dFault [40],

FChain [38] and HANSEL [43] exploit dependency graphs representing the complex chain of dependencies that exists in services, analyze symptoms from the network, and output a candidate set of root causes for the faults. Even though GRETEL requires a training phase to build its operational fingerprints, it operates at high throughput rates and reports no lag or loss of packets (per § 7.4.1).

COMPARISON WITH HANSEL. GRETEL is similar in spirit to HANSEL [43]. However, there exists significant differences between the two systems, as described below:

(**1**) GRETEL detects both operational and performance anomalies and provides a potential root cause of such errors. In contrast, HANSEL merely detects operational faults without providing a possible reason for the error.

(**2**) GRETEL classifies a faulty operation as a high level administrative task. In contrast, HANSEL merely provides the administrator with a low-level sequence of operations for each task that failed.

(**3**) GRETEL processes messages as they arrive and requires no buffering. Thus, it provides a more scalable design as compared to HANSEL, which employs time buckets to avoid any dropped packets.

(**4**) GRETEL triggers its error reporting only when it detects a fault. HANSEL, in contrast, leverages a heavy duty stitching logic that is triggered on every message, which coupled with the time buckets, increases its error reporting latency to almost 30s for zero dropped packets.

(**5**) HANSEL analyzes the request and response payloads to extract meaningful identifiers. However, common identifiers, like tenant ID, etc., may cause a faulty operation to link with several successful operations. In contrast, GRETEL is precise and detects only the faulty operation.

(**6**) GRETEL may fail to report the faulty operation if the sequence of packets fails to match a fingerprint. On the other hand, HANSEL does not suffer from this limitation and will always a report a chain of events leading to the fault.

(**7**) GRETEL does not handle failures in asynchronous calls in OpenStack operations. In contrast, HANSEL can still provide a chain of events leading to the fault.

9.3 Log analysis

Tracing systems [24, 35, 46, 48, 49] leverage log analysis to determine causal relationships among events. Such tools, however, are limited by the verbosity of the logs, and employ probabilistic data mining approaches.

Magpie [23] and ETE [32] rely on the distributed system's event semantics, and use temporal join-schemas on custom log messages. However, effectiveness of log analysis is limited by the comprehensiveness of the system logs and their debug level. Moreover, production systems typically have very lightweight logging enabled. Thus, GRETEL does not depend on log analysis, but can augment its root cause analysis using logs, if available.

Log analysis approaches induce a delay since determination of causal relations takes time until all the logs are collected and analyzed. In contrast, GRETEL can operate at high network speeds.

10. CONCLUSION

We present GRETEL, a fast fault detection and diagnosis system for OpenStack, which leverages non-intrusive system monitoring to systematically combine different system states and identify root cause of operational and performance faults in OpenStack. GRETEL uses unique operation fingerprints to quickly identify faulty operations at runtime. GRETEL is lightweight and precise even under stress.

11. ACKNOWLEDGEMENT

We thank our shepherd, Nedeljko Vasic, and Vijay Mann and the anonymous reviewers for their valuable comments.

12. REFERENCES

[1] Apache CloudStack. https://goo.gl/1S3K9W.
[2] Bro. https://www.bro.org/.
[3] Broccoli. https://goo.gl/4NUdFi.
[4] Cloud block storage issues. https://goo.gl/E6BPxG.
[5] Cloud servers issues. https://goo.gl/yg9gFB.
[6] Cloudera HTrace. https://goo.gl/Pz3lQu.
[7] collectd. https://collectd.org/.
[8] Correlation id in python-glanceclient. https://goo.gl/UyFDOr.
[9] OpenStack. https://www.openstack.org/.
[10] Rackspace Issue 1. https://goo.gl/2tdjHB.
[11] Rackspace Issue 2. https://goo.gl/CnqSTl.
[12] Rackspace Issue 3. https://goo.gl/JVVpX0.
[13] tc: Traffic Control in the Linux kernel. http://goo.gl/f8YDaH.
[14] Tcpreplay. http://tcpreplay.synfin.net/.
[15] Tempest. http://goo.gl/OZiXTV.
[16] tsoutliers. https://goo.gl/aVxsSJ.
[17] Twitter Zipkin. https://goo.gl/bHtUKc.
[18] VM build intermittent failure. https://goo.gl/s3VP3j.
[19] VMware vSphere. http://goo.gl/kNAR0f.
[20] M. K. Aguilera et al. Performance Debugging for Distributed Systems of Black Boxes. In *SOSP'13*.
[21] M. Attariyan et al. X-ray: Automating Root-cause Diagnosis of Performance Anomalies in Production Software. In *OSDI'12*.
[22] P. Bahl et al. Towards Highly Reliable Enterprise Network Services via Inference of Multi-level Dependencies. In *SIGCOMM'07*.
[23] P. Barham et al. Using Magpie for Request Extraction and Workload Modelling. In *OSDI'04*.
[24] L. Bitincka et al. Optimizing Data Analysis with a Semi-structured Time Series Database. In *SLAML'10*.
[25] A. Chanda et al. Whodunit: Transactional Profiling for Multi-tier Applications. In *SOSP'07*.
[26] M. Y. Chen et al. Path-based Failure and Evolution Management. In *NSDI'04*.
[27] M. Y. Chen et al. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *DSN'02*.
[28] R. Fonseca et al. Experiences with Tracing Causality in Networked Services. In *INM/WREN'10*.
[29] R. Fonseca et al. X-trace: A Pervasive Network Tracing Framework. In *NSDI'07*.
[30] T. Gschwind et al. Webmon: A Performance Profiler for Web Transactions. In *WECWIS'02*.
[31] H. S. Gunawi et al. What Bugs Live in the Cloud?: A Study of 3000+ Issues in Cloud Systems. In *SOCC'14*.
[32] J. L. Hellerstein et al. ETE: A Customizable Approach to Measuring End-to-end Response Times and their Components in Distributed Systems. In *ICDCS'99*.
[33] X. Ju et al. On Fault Resilience of OpenStack. In *SOCC'13*.
[34] S. Kandula et al. Detailed Diagnosis in Enterprise Networks. In *SIGCOMM'09*.
[35] S. P. Kavulya et al. Draco: Statistical Diagnosis of Chronic Problems in Distributed Systems. In *DSN'12*.
[36] R. R. Kompella et al. IP Fault Localization via Risk Modeling. In *NSDI'05*.
[37] E. Koskinen et al. BorderPatrol: Isolating Events for Black-box Tracing. In *Eurosys'08*.
[38] H. Nguyen et al. FChain: Toward Black-box Online Fault Localization for Cloud Systems. In *ICDCS'13*.
[39] V. Paxson. Bro: A System for Detecting Network Intruders in Real-time. In *USENIX Security'98*.
[40] P. Prakash et al. dFault: Fault Localization in Large-scale Peer-to-peer Systems. In *Middleware'10*.
[41] P. Reynolds et al. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI'06*.
[42] R. R. Sambasivan et al. Diagnosing Performance Changes by Comparing Request Flows. In *NSDI'11*.
[43] D. Sharma et al. HANSEL: Diagnosing Faults in OpenStack. In *CoNEXT'15*.
[44] B. H. Sigelman et al. Dapper: A Large-scale Distributed Systems Tracing Infrastructure. *Google Research*, 2010.
[45] B. C. Tak et al. vPath: Precise Discovery of Request Processing Paths from Black-Box Observations of Thread and Network Activities. In *ATC'09*.
[46] J. Tan et al. Visual, Log-based Causal Tracing for Performance Debugging of MapReduce Systems. In *ICDCS'10*.
[47] E. Thereska et al. Stardust: Tracking Activity in a Distributed Storage System. In *SIGMETRICS'06*.
[48] W. Xu et al. Detecting Large-scale System Problems by Mining Console Logs. In *SOSP'09*.
[49] D. Yuan et al. SherLog: Error Diagnosis by Connecting Clues from Run-time Logs. In *ASPLOS'10*.