Sieve
Actionable Insights from Monitored Metrics in Distributed Systems
https://sieve-microservices.github.io/

Jörg Thalheim, Pramod Bhatotia
University of Edinburgh

Antonio Rodrigues
CMU

Bimal Viswanath
University of Chicago

Istemi Ekin Akkus, Ruichuan Chen
NOKIA Bell Labs

Lei Jiao
University of Oregon

Christof Fetzer
TU Dresden
Monitoring of distributed systems

- Most distributed systems are constantly monitored
  - Amazon CloudWatch, Azure Monitor, and Google StackDriver

- Goals of monitoring
  - **Efficiency** (Resource management, Autoscaling)
  - **Dependability** (detect and fix failures)
  - etc.
Challenges

Distributed systems are complex

- Uber: 500+ services
- LendingClub: from 5 services in 2013 to 139 services in 2015

To monitor and understand them is difficult

- Netflix: 2,000,000 metrics
- OpenStack: 17,608 metrics

Metrics * machines * services → **Information overflow**
Problem statement

How to derive actionable insights from monitored metrics in distributed systems?

Previous work
- Limited to message-level happens-before relationships
- Requires application-specific instrumentation

Design goals
- Utilize existing monitoring infrastructure without modifying the application
- Make it general
Contributions

1. Sieve: a general framework to derive actionable insights from monitored metrics

2. Applied Sieve to two case studies:
   ○ Root cause analysis in OpenStack
   ○ Autoscaling in ShareLaTex
Key ideas of Sieve

Complex distributed systems
- Several services
- Each service exporting several metrics

1. Metric reduction engine:
   - Filter metrics per service that contains redundant information

2. Metric dependency extractor
   - Infer predictive-causal relationships between applications
Outline

✓ Introduction
  ● Design
  ● Evaluation
  ● Case studies
Sieve overview

1. Load the application
2. Reduce metrics
3. Identify relationships
#1: Load generator

- **Purpose:**
  - Generate load with known random distribution to derive metrics
  - Derive a call graph for inferring communication b/w services

- **Characteristics of load generator:**
  - Runs in offline mode
  - Application-specific

- **Our case studies:**
  - **OpenStack:** Used the shipped load generator Rally
  - **ShareLaTex:** Self written, simulates virtual users
#1: Derive metrics

- **System Metrics**
  - CPU usage
  - Disk I/O
  - Cache misses

- **Application Metrics**
  - Timing of db queries
  - Business metrics
  - HTTP request counts
  …
Sieve overview

1. Load the application
2. Reduce metrics
3. Identify relationships
#2: Reduce metrics

A single service

N metrics

K metric clusters

(K<<N)

Cluster of metrics that are highly correlated
#2: Time series clustering

**Solution:** K-Shape time-series clustering [SIGMOD’15]
- **Unsupervised** algorithm
- **Robust** to distortion
- **Scales** linearly

**Caveat:** Preprocessing is necessary
- Filter metrics with constant values or low variance/frequency
- Normalize units: \( \text{bytes/s, MB, s} \rightarrow \text{Zscore} \)
Sieve overview

1. Load the application
2. Reduce metrics
3. Identify relationships
#3: Identify relationships

Service A

Service B

B has a dependency with A
#3: Granger causality

- **Statistical property:**
  
  "X granger-cause Y"

  ≡ X provides statistically significant information about the future of Y

- **Methodology:** Create a linear regression model (OLS)

  - Y = a*X(t-1) + b * X(t-2) ...

- Null hypothesis test using F-Statistics
#3: Call graph

Points to service from consumer of the service

Record communication patterns by logging network related syscalls
#3: Dependency graph

Metric pairs where granger causality was found
Outline

✓ Introduction
✓ Design
● Evaluation
● Case studies
  ○ Root cause analysis
  ○ Autoscaling
Evaluation: Microbenchmarks

What is the resulting improvement in monitoring overhead?
(more results in the paper)

Experimental setup:
- ShareLaTex application
- 10 node cluster
Reduction in monitoring overheads

Sieve reduces monitoring overheads up to 90%
Case study #1: Root Cause Analysis (RCA)

Before
no anomaly

After
with anomaly

Output
root cause analysis ranking

<table>
<thead>
<tr>
<th>Rank</th>
<th>Service</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>A</td>
<td>of cluster 2</td>
</tr>
<tr>
<td>2nd</td>
<td>C</td>
<td>of cluster 4</td>
</tr>
<tr>
<td>3rd</td>
<td>B</td>
<td>of cluster 1</td>
</tr>
</tbody>
</table>

service
cluster

missing edge: hint on root cause
Result: RCA

- **Symptom:** When launching VMs, they go into failed state despite the availability of compute nodes.
- **Root cause:** Crash in Neutron service (provides network)
- **Results:**

---

**Diagram:**

1. **Nova API** (1st rank)
   - Metric added: `nova_instances_in_state_ERROR`
   - Metric discarded: `nova_instances_in_state_ACTIVE`

2. **Neutron Server** (3rd rank)
   - New edge: `neutron_ports_in_status_DOWN`
   - Metric remains in `neutron_ports_in_status_ACTIVE`

---

23
Case study #2: Autoscaling

Most influential metric: http-requests_Project_id_GET

Relations w/ http-requests_Project_id_GET_mean

Other relations
Result: Autoscaling

- **Application**: ShareLaTex
- **Workload**: World cup ‘98 traces
- **Baseline**: Default autoscaling rule w/o application knowledge
- **Setup**: 12 t2.large VM-Instances on Amazon EC2
Result: Autoscaling

- Higher CPU utilisation
- Less SLA violations & scaling actions
Summary

Sieve is a general framework for distributed systems:

- To derive actionable insights from monitored metrics
- Efficient and robust way to reduce the complexity of monitoring

Sieve applied to two case studies:

- Root cause analysis in OpenStack
- Autoscaling for ShareLaTex

Thanks!

Source code: https://sieve-microservices.github.io/
Sieve overview

1. Load the application
   - Excite components to produce metrics (for Step 2)
   - Produce call graph among components (for Step 3)

2. Reduce metrics
   Analyze each component's metrics and filter redundancies (for Step 3)

3. Identify relationships
   Use important metrics from Step 2 and call graph from step 1 to produce relations
Case study #1: RCA in OpenStack

Methodology:

1. Pick anomalies from OpenStack’s bugtracker with known root causes
2. Run Openstack on both faulty and healthy versions, and run load generator Rally
3. Generate ranked list of possible root causes, and compare it with known root cause
Case study #1: RCA in OpenStack

Methodology:

1. Pick anomalies from OpenStack’s bugtracker with known root causes
2. Run Openstack on both pre- and post-commit versions, and run load generator Rally
3. Generate ranked list of possible root causes, and compare it with known root cause
Evaluation methodology: RCA

- **OpenStack**, a cloud management software
  - 47 components (total)
  - 17,608 metrics

**Methodology:**

1. Pick anomalies from OpenStack’s bugtracker with known root causes
2. Run Openstack on both **pre-** and **post-commit** versions, and run load generator Rally
3. Generate ranked list of possible root causes, and compare it with known root cause
[Step #1] Callgraph design

- sysdig.ko
- Getting the callgraph
- Syscalls trace
- Client
- Service

recvfrom fd=3(192.168.8.17:52252 -> 192.168.8.36:http) size=2048
Case Study: Autoscaling engine with Kapacitor

```plaintext
var cpu_percentile = stream
  |from()
    .measurement('docker_cpu')
    .where(lambda: "cont_image" =~ /sharelatex-web/)
  |window()
    .period(10s)
    .every(1s)
  |percentile('usage_percent', 95.0)
  |log()

var scale_out = cpu_percentile
@scale()
  .simulate(FALSE)
    .id('1s33') // web service id
  .when('percentile > 90')
  .by('current + 2')
  .min_instances(1)
  .max_instances(6)
  .cooldown('10s')
```
Workload: Request rate for Worldcup 98
Q3: Consistency across workloads

- Pairwise comparison cluster assignment of different workloads
- **AMI:**
  - adjusted mutual information
  - entropy measure how different two cluster assignments are
  - Higher is better (best at 1.0)

→ Clusters are consistent: Most services are in range of 0.5 to 0.9

- Other results in the thesis:
  - Graphical and semantical evaluation of cluster
[Step #2] Detect and eliminate monotonic counters
Callgraph: overhead

![Chart showing overhead comparison for native, sysdig type, and tcpdump methods. The chart indicates that sysdig type has the highest overhead, followed by native, and tcpdump has the least.]
Case study: Autoscaling

Application (ShareLatex) → Monitoring → Scaling

Telegraf → InfluxDB → Kapacitor

UDF: kapacitor-scaling
[Step #1] Load the application: Framework

Application

Monitoring

- Metric collector
- Application aware
- Input protocols: Statsd

Telegraf

- Time-series database
- Horizontal scaleable
- SQL

InfluxDB
[Step #2] Reducing metrics: K-Shape example
Example: Clusters of chat component

Metrics of cluster 1
- **Memory**: pgfault, pgpgin, total_pgfault, ...

Metrics of cluster 2
- **HTTP**: http-room_project_id_messages_POST
- **Database**: mongo-messages_insert, mongo-rooms_query_project_id
- **Network**: rx_bytes, tx_bytes, ...
- **CPU**: usage_in_kernelmode, usage_in_usermode, ...
Sieve - A system overview

1. Loading the Application
   - Load
   - Application
   - Metrics timeseries

2. Reducing Metrics
   - N Metrics
   - k Cluster
   - \( k \ll N \)

3. Identifying Relationships Across Services
   - Service_1
   - Service_2
   - Service_3
   - metric relation
Q1: Reduction of metrics

On average Sieve reduces metrics by 92%
Key ideas

The underlying intuition behind Sieve is two-fold: Firstly, in the metric dimension, some metrics of a component may behave with similar patterns as other metrics of that component. Secondly, in the component dimension, there are dependencies between components. As a result, monitoring all metrics of all components at runtime may be unnecessary and inefficient (as components are not independent).
[Step #2] Reducing metrics: Preprocessing

1. Filter metrics with constant values or low variance/frequency
2. Normalize units:
   - \( \text{bytes/s, MB, s} \rightarrow \text{Zscore} \)
   - \( \text{Zscore}(s) = (x-\mu)/\sigma \)
   - \( \mu \) .. mean; \( \sigma \) .. standard deviation
3. Detect and derive monotonic counters
Solution: K-Shape [Sigmod2015]

- **Unsupervised** time-series cluster algorithm
- Robust to distortion in
  - Phase
  - Amplitude
  - And time (or time lags)
- Scales **linearly** well with the number of metrics
Case study: RCA details

1. Extract new and discarded metrics
   - New: A: , C: 
   - Discarded: A: , B: , C: 

2. Rank components by novelty
   - Novelty: A: 2, B: 2, C: 0, 0, 0

3. Calculate cluster novelty and similarity
   - Similarity: \( \frac{1}{2} \) A: 1, 1, 0, 0, 0

4. Filter edges by novelty and similarity
   - New edge: A \( \rightarrow \) C
   - Discarded: B \( \rightarrow \) C
   - Unchanged: A \( \rightarrow \) B

5. Final rankings
   - C
   - A
   - B

Legend:
- Component
- Cluster
- Metric
Result: RCA

- **Symptom:** When launching VMs, they go into failed state despite the availability of compute nodes. *(More bugs in the paper)*
- **Root cause:** Crash in Neutron service (provides network)
- **Results:**
Result: RCA

- **Symptom:** When launching VMs, they go into failed state despite the availability of compute nodes. *(More bugs in the paper)*
- **Root cause:** Crash in Neutron service (provides network)
- **Results:**

<table>
<thead>
<tr>
<th>Component</th>
<th># filtered metrics</th>
<th>Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nova API</td>
<td>29 / 59 (-.51%)</td>
<td>1</td>
</tr>
<tr>
<td>Nova libvirt</td>
<td>21 / 39 (-.46%)</td>
<td>2</td>
</tr>
<tr>
<td><strong>Neutron Server</strong></td>
<td><strong>12 / 42 (-.71%)</strong></td>
<td><strong>3</strong></td>
</tr>
<tr>
<td>RabbitMQ</td>
<td>11 / 57 (-.81%)</td>
<td>4</td>
</tr>
<tr>
<td>Neutron L3 agent</td>
<td>7 / 39 (-.82%)</td>
<td>5</td>
</tr>
</tbody>
</table>
Evaluation results: RCA anomaly #1

- **Symptom:** Error message ‘No valid host was found. There are not enough hosts available.’ when launching VM, despite the availability of compute nodes.

- **Root cause:** Crash in Neutron component (#1533942 in Launchpad)

- **Results:**

<table>
<thead>
<tr>
<th>Component</th>
<th># filtered metrics</th>
<th>Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nova API</td>
<td>29 / 59 (-.51%)</td>
<td>1</td>
</tr>
<tr>
<td>Nova libvirt</td>
<td>21 / 39 (-.46%)</td>
<td>2</td>
</tr>
<tr>
<td>Neutron Server</td>
<td>12 / 42 (-.71%)</td>
<td>3</td>
</tr>
<tr>
<td>RabbitMQ</td>
<td>11 / 57 (-.81%)</td>
<td>4</td>
</tr>
<tr>
<td>Neutron L3 agent</td>
<td>7 / 39 (-.82%)</td>
<td>5</td>
</tr>
</tbody>
</table>
Outline

✓ Introduction
✓ Design
● Evaluation
● Case studies
  a. Root cause analysis
  b. Autoscaling