Pythia: A Just-in-Time Instrumentation Framework for Distributed Systems

Lily Sturmann☆△, Emre Ates†, Peter Portante△, Orran Krieger☆+△, Ayse Coskun†, Raja Sambasivan☆+ Mass. Open Cloud☆ / Boston University† / Red Hat△
Debugging: a perennial problem
Unique challenges in debugging distributed systems

Where is the problem? could be in ...

- One of many components
- One of several stack levels, ex:
  - VM vs. hypervisor
  - Application vs. kernel
- Inter-component interactions
Limitations of today’s debugging methods

Different problems benefit from different instrumentation points.

You can’t instrument everything: too much overhead, too much data.
Today’s painful debugging cycle

Problem found? Usually no.....

Gather data from current instrumentation

Currently, enabling the right instrumentation requires manual iterations of **guess and check**.
- This takes a lot of valuable **developer time**.
- It increases **downtime**, or the time the system is in a problem state.
- Both effects also additionally **cost money**.

Use data to guess where to add instrumentation
Perf. variation indicates where instrumentation is needed

Requests that are expected have **similar workflows** in the same system should have similar performance. High performance variation may indicate a problem.

**Expectation:**
Similar performance

**Reality:**
Varied performance

Slow performance can also be localized to find dominant contributors
The Pythia Just-in-Time Instrumentation Framework

- **Always-on framework** that automatically explores search space of instrumentation
  - Enables instrumentation needed to provide **visibility into new problem**
- Key enabling technology, **end-to-end tracing**, captures workflow
- Current targets: traditional distributed systems with concurrency, synchronization, and call/callee relationships preserved

Problems that Pythia can help with:
- Localizing **high performance variation**
  - Ex. load balancing problems
- Localizing contributors to **consistently slow requests**
  - Ex. insufficient resources
● Motivation
● Proposed Approach
● Initial experiences
Proposed Steps

1. Gather request workflows
   - Gather data

2. Group workflows by perf. expectation
   - Identify problematic groups
   - Identify consistently slow groups
   - ID groups with high perf. var

3. Localize dominant contributors to response time
   - Determine where to enable instr.
   - Localize performance variation

4. Enable additional instrumentation flatly
   - Determine what instr. to enable
   - Enable additional instrumentation hierarchically
Proposed Steps

1. Gather request workflows
2. Group workflows by perf. expectation
3. Gather data
4. Identify problematic groups
5. Determine where to enable instr.
6. Localize dominant contributors to response time
7. Localize performance variation
8. Enable additional instrumentation flatly
9. Enable additional instrumentation hierarchically
10. Determine what instr. to enable
Proposed Steps: Gathering data

1. Gather request workflows
2. Group workflows by perf. expectation
3. Gather data
4. Identify problematic groups
5. ID consistently slow groups
6. ID groups with high perf. var
7. Localize dominant contributors to response time
8. Localize performance variation
9. Enable additional instrumentation flatly
10. Enable additional instrumentation hierarchically

Determine **where** to enable instr.
Determine **what** instr. to enable
Gathering requests with end-to-end tracing

End-to-end tracing preserves traces (i.e., records) of request workflows. It works by:

- Propagating unique context (request ID) with individual requests
- Afterwards, stitching together trace points that have the same or related context
Proposed Steps: Gathering data

1. Gather request workflows
2. Group workflows by perf. expectation
3. Gather data
4. Identify problematic groups
5. ID consistently slow groups
6. ID groups with high perf. var
7. Localize dominant contributors to response time
8. Localize performance variation
9. Enable additional instrumentation flatly
10. Determine where to enable instr.
11. Enable additional instrumentation hierarchically
12. Determine what instr. to enable
Proposed Steps: Identifying problem groups

1. Gather request workflows
   - Group workflows by perf. expectation

2. ID consistently slow groups
   - ID groups with high perf. var
   - Perf. NOT similar

3. Localize dominant contributors to response time
   - Localize performance variation

4. Enable additional instrumentation flatly
   - Enable additional instrumentation hierarchically

Gather data
Identify problematic groups
Determine where to enable instr.
Determine what instr. to enable
Pythia’s current approach to IDing problematic groups

Example: Read image from glance request duration

Problem symptoms

Large std deviation

Multimodal distribution
Proposed Steps: Determining where to enable

1. Gather request workflows
2. Group workflows by perf. expectation
3. ID consistently slow groups
4. ID groups with high perf. var
5. Localize dominant contributors to response time
6. Localize performance variation
7. Enable additional instrumentation flatly
8. Enable additional instrumentation hierarchically

Gather data | Identify problematic groups | Determine where to enable instr. | Determine what instr. to enable
Pythia’s current approach to localizing perf. variation

Use both flat edges and hierarchy:

- The hierarchy **preserves** the call-graph relationship in the trace

- The flat model **captures** latency even in the gaps that the hierarchy leaves open
Proposed Steps: Determining **what** to enable

**Gather data**

1. Gather request workflows
2. Group workflows by perf. expectation

**Identify problematic groups**

1. ID consistently slow groups
2. ID groups with high perf. var

**Determine where to enable instr.**

1. Localize dominant contributors to response time
2. Localize performance variation

**Determine what instr. to enable**

1. Enable additional instrumentation flatly
2. Enable additional instrumentation hierarchically
Determining further instrumentation to enable

Using the hierarchical caller / callee relationship can narrow the search space.
Prioritizing spans deeper in the call graph

Using the hierarchical caller / callee relationship can narrow the search space.
Determining instrumentation: more advanced approaches

- Pattern-based approaches: skip levels
- Flat approaches: optimization strategies
Open questions about Pythia

- Other statistical tests for different problem types?
  - Short circuit the hierarchical search strategy?
- Right approach for exploring different stack layers?
- Can “evolutionary” approaches help (reinforcement learning)?
- Value of the hierarchical approach?
● Motivation
● Proposed Approach
● Initial Experiences
Current codebase focuses on localizing variation

Control plane
- High variation edges
  - Variation localization
- Localize performance variation

Data plane
- Distributed system
- Tracing infrastructure
What is OpenStack?

- Open source cloud management platform
- Microservices architecture
- Components include:
  - Nova = compute
  - Neutron = networking
  - Glance = image store ... and many others!
- OSProfiler is default tracing system
OpenStack: open source cloud management

https://docs.openstack.org/install-guide/get-started-logical-architecture.html
Is our fundamental assumption correct?

1. Is there an opportunity for Pythia to suggest additional instrumentation points in OpenStack?

2. What are the sources of high variation in OpenStack currently?
Methodology for determining instrumentation sufficiency

- Traced workload: ‘server create’ followed by ‘server delete’
- Current implementation groups traces according to DFT of structure
- Looking for areas with high variation

Experimental set up:

- OpenStack Research Environment for Mass Open Cloud
  - Modified single-node devstack environment (virtualized OpenStack: Pike release)
    - OSProfiler enabled
- Uses workload to issue 30 create / deletes in a row
  - Flavor: m1.tiny
  - Image cirros-0.3.5-x86_64-disk
Instrumentation not capturing interesting behaviors

server create requests
Example high variation operations in OpenStack

- ComputeManager's `build_and_run_instance` → Neutronv2 API's ClientWrapper proxy
- Keystone's auth/tokens → DHCPAgent's `port_delete`
- `rpc: nova.compute.manager.ComputeManager.terminate_instance` → `vif_driver:nova-compute`
- `vif_driver:nova-compute` → `rpc:nova-compute`
  - `nova.schedular.rpcapi.SchedulerAPI.update_instance_info`

Ultimately, there is some behavior not captured by the default instrumentation that might be important.
### Requirements imposed by Pythia on tracing system

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Open Tracing infra</th>
<th>Pythia adapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable / disable instrumentation</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Manifest of tracepoints</td>
<td>--</td>
<td>✔️</td>
</tr>
<tr>
<td>Collector accessible for automation</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Preserves synchronization</td>
<td>✔️</td>
<td>--</td>
</tr>
<tr>
<td>Well-defined end markers for traces</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

✓ = could provide  
-- = not applicable
Next Steps

- Replace OSProfiler with Jaeger
  - Extend Jaeger with ability to dynamically enable tracepoints
- Using Pythia to diagnose problems in OpenStack and other systems
- Creating a standalone tool to localize performance variation
- Understanding Pythia overhead: system resources, time
Pythia: automate time-consuming debugging cycle

Key insights:
○ Similar workflows should perform similarly
○ Performance variation and slow performance can be localized to specific parts of a request

Work in progress:
○ Analyzing baseline variation in OpenStack
○ Running tests on correlation between performance variation and system problems