An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications

Emre Ates[⋆], Lily Sturmann[◇], Mert Toslali[⋆]
Orran Krieger[⋆], Richard Megginson[◇], Ayse K. Coskun[⋆], Raja R. Sambasivan[†]
[⋆]Boston University, [◇]Red Hat Inc., [†]Tufts University

ABSTRACT

Diagnosing performance problems in distributed applications is extremely challenging. A significant reason is that it is hard to know where to place instrumentation *a priori* to help diagnose problems that may occur in the future. We present the vision of an automated instrumentation framework, Pythia, that runs alongside deployed distributed applications. In response to a newly-observed performance problem, Pythia searches the space of possible instrumentation choices to enable the instrumentation needed to help diagnose it. Our vision for Pythia builds on workflow-centric tracing, which records the order and timing of how requests are processed within and among a distributed application's nodes (i.e., records their workflows). It uses the key insight that localizing the sources high performance variation within the workflows of requests that are expected to perform similarly gives insight into where additional instrumentation is needed.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → *Distributed computing methodologies*.

KEYWORDS

performance, tracing, distributed systems, cross-layer

ACM Reference Format:
Emre Ates, Lily Sturmann, Mert Toslali, Orran Krieger, Richard Megginson, Ayse K. Coskun and Raja R. Sambasivan. 2019. An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications. In *ACM Symposium on Cloud Computing (SoCC '19), November 20–23, 2019, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3357223.3362704

1 INTRODUCTION

Instrumentation in the form of logs or performance counters is the de-facto data source engineers use to diagnose performance problems in production distributed applications. But, it is difficult to know a priori *where* instrumentation must be enabled (e.g., in which distributed-application nodes), in *which* data-center stack layers instrumentation must be enabled (e.g., application or guest OS), and *what* instrumentation must be enabled (e.g., in which specific functions) to help diagnose problems that may occur in the future [3, 20, 35–37]. Enabling all possible instrumentation all of the time is infeasible due to the resulting overhead. As a result, engineers must spend lengthy, time-consuming cycles manually exploring the space of possible instrumentation choices before they can identify the root cause of a new problem [7, 11, 20].

Past efforts to address this instrumentation decision problem focus only on what already-enabled instrumentation should be preserved in logs, not where instrumentation needs to be enabled in the first place [10], or on instrumentation for correctness problems, not performance [3, 5, 14, 18, 35–37, 40]. Replay-based approaches for performance diagnosis [4] circumvent the instrumentation decision problem, but may not reduce time to diagnosis because they require a separate offline phase. Many of these past efforts do not focus on distributed systems in which only a portion of requests may be problematic and in which different requests may access different distributed-application nodes. None of these efforts consider instrumentation placement across multiple data-center stack layers.

In this paper, we present the vision of a *continuously-running instrumentation framework for production distributed applications that, in response to a newly-observed performance problem, automatically explores the space of possible instrumentation choices within multiple stack layers and enables the instrumentation needed to help diagnose it*. We present two inter-related observations that make such a framework possible.

The first observation is that in many distributed applications, requests that exhibit similar workflows—i.e., that are processed similarly within and among the nodes of a distributed application and within lower data-center stack layers—should perform similarly [27, 29]. See Fig. 1 for two possible request workflows in a simple distributed application. Thus, if requests that *are expected to* perform similarly *do not do so*, there is something unknown about their workflows. This unknown behavior may be indicative of performance problems, such as unexpected slow code paths being executed, load imbalances (perhaps due to unintended hardware heterogeneity), or contention.

Localizing the source of the observed variation gives insight into *where* (e.g., in which node) additional instrumentation is needed to identify the unknown behavior. Focused search strategies (e.g., based on domain knowledge or machine learning) can then be used to explore in *which* stack layer instrumentation is needed and *what* instrumentation is needed to explain

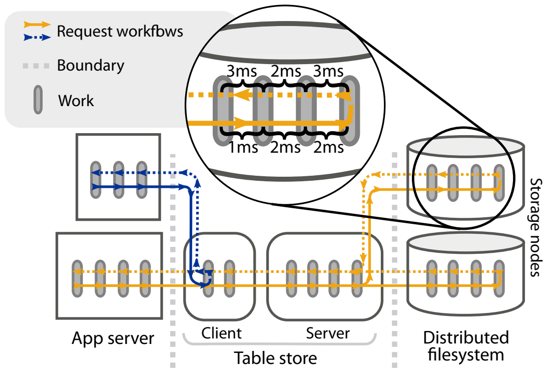


Figure 1: Two simple workflows. The figure shows workflows for two READ requests in a simple distributed application comprised of an app server, a table store, and a storage system. The first request (blue) hits in the table store's client cache, whereas the second request (orange) requires a storage-node access. (Figure from Sambasivan et al. [28])

the variation. For cases where performance problems manifest as consistently slow requests instead of high variation, a similar process could be used that focuses on identifying dominant contributors to request response times.

The second observation is that recent work on workflow-centric tracing of distributed applications, also called end-to-end tracing, [8, 12, 13, 16, 17, 19, 25–30, 33] makes it possible to capture requests' workflows by inserting tracing instrumentation to the application. Workflow-centric tracing works by propagating context (e.g., request IDs) with individual requests as the requests are executed by the distributed application. Records of log points executed by requests are tagged with requests' context. (We call log points that record context *trace points*.) Later, a trace reconstructor gathers trace-point records from different machines and stitches together ones with related context to create traces (graphs) of requests' workflows. Sampling techniques can be used to keep tracing's overhead low enough (e.g., $< 1\%$) to be used in production, as is done at many companies today [16, 17, 25, 30].

Building on the above two observations, we present the following contributions: *1) Pythia*, an initial approach for an automated, cross-layer instrumentation framework that uses performance variation, response times, and workflow-centric tracing to both identify where additional instrumentation is needed and to test whether newly-enabled instrumentation gives further insight into an observed problem. Our approach currently targets private clouds where all of the data-center stack layers are controlled by the same entity. *2)* An architecture for enabling this approach. *3)* An initial exploration of ways to represent the space of possible instrumentation choices across the application and guest kernel's system calls, and an initial set of search strategies for choosing what instrumentation to enable. *4)* A validation of our approach for Pythia by using it to manually diagnose a performance problem in OpenStack [24], a popular open-source distributed application.

2 THE VISION OF PYTHIA

Pythia will be an always-on instrumentation framework that is deployed alongside distributed applications and any lower data-center stack layers that support workflow-centric tracing (e.g., the guest OS or hypervisor). Examples of applications that support tracing today include OpenStack [24], Ceph [34], and HDFS [32]. We have recently started work on a Linux kernel that supports tracing [31]. To create traces that show activity across all tracing-enabled layers, applications and stack layers that use different tracing implementations (e.g., Jaeger [16] or Zipkin [39]) must agree on a common format for trace points and context (e.g., the OpenTracing standard [25]).

The instrumentation Pythia will control—i.e., selectively enable and disable—will consist of trace points added to distributed applications and lower stack layers, as well as variables values that could be captured within trace points, such as function parameters, queue lengths, and performance-counter values. Trace points could be statically embedded within applications and lower stack layers and controlled via signals sent to them. Alternatively, they could be injected into pre-determined locations (e.g., function boundaries) during runtime [7, 11, 19].

Pytha's operational steps: Pythia will operate in a continuous cycle, as shown in Figure 2. At the beginning of time, Pythia will take as input: 1) initial, low-fidelity expectations of which requests' should perform similarly and 2) workflow skeletons created with a set of trace points that are always enabled in the distributed application and lower stack layers. Both initial expectations and workflow skeletons represent starting points that Pythia will automatically refine at each step of its cycle.

Initial expectations are specified by developers and focus on properties of requests' critical paths. This is because requests' performance depends only on their critical paths. We expect initial expectations to be of the form "expect all requests of the same type whose critical paths access the same distributed-application services to perform similarly." But, they could also include additional application-specific details. Pythia will transform initial expectations into regular expressions (a series of trace-point names that must match) during runtime. *Workflow skeletons* must contain sufficient instrumentation to identify critical paths observed during runtime.

In the *first step* of the the cycle, Pythia will extract workflow skeletons' critical paths to create critical-path skeletons, which are subsets of the traces that are created by discarding concurrent branches that do not affect overall request latency from the traces. In the *second step*, Pythia will group requests' critical-path skeletons as per the expectations of which ones should perform similarly. Groups are annotated with response-time distributions and performance variances. Each group also maintains a single representative critical-path skeleton which is annotated with detailed latency distributions between trace

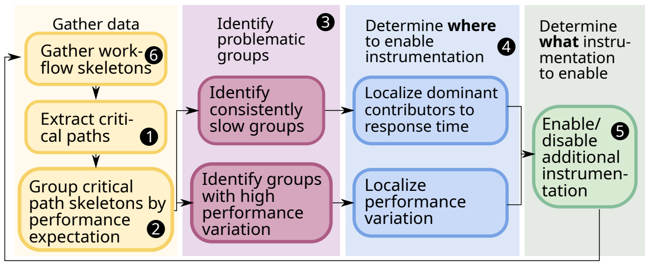

Gather data
Gather workflow skeletons 6
Extract critical paths 1
Group critical path skeletons by performance expectation 2
Identify problematic groups 3
Identify consistently slow groups
Identify groups with high performance variation
Determine where to enable instrumentation 4
Localize dominant contributors to response time
Localize performance variation
Determine what instrumentation to enable
Enable/disable additional instrumentation 5

Figure 2: Pythia's continuous cycle ofoperation.

points specified in the corresponding expectation.

In the *third step*, Pythia will examine the response-time distributions of requests assigned to different groups. It will identify problematic groups containing requests that either exhibit high coefficient of variation (CoV) or which are consistently very slow (i.e., have high response times with low variation). Coefficient of variation is a measure of variation that captures the intuition that groups that have high standard deviation compared to their mean are worse offenders than those with standard deviation equal to or lower than their mean. The threshold for deciding if a group is consistently slow could be determined by whether its mean response falls in the tail of the overall response-time distribution for all requests.

In the *fourth step*, Pythia will explore where to enable instrumentation for problematic groups. To do so, it will localize dominant contributors to performance variation or high response times within problematic groups' critical-path representatives.

In the *fifth step*, Pythia will use various search strategies to identity what instrumentation must be enabled in the identified areas for requests assigned to the problematic category, as well as which previously-enabled instrumentation to disable if it is not sufficient to explain observed variation. One simple search strategy for explaining variation is to enable more granular instrumentation hierarchically (e.g., first services of a distributed application, then components within services, then nodes that constitute components, then functions within nodes). Between cycles of enabling more granular instrumentation, this strategy could examine whether variable values that could be exposed in already-enabled trace points explains variation.

In the *sixth step*, Pythia will refine the expectations to account for the newly enabled or disabled instrumentation. It will also gather new workflow skeletons that are enriched with the additional instrumentation. *Separate garbage collection step*: In addition to the above steps, Pythia will run an additional step periodically to disable instrumentation that has not been observed on critical paths recently.

Concurrently with the above steps, Pythia will preserve 1) (enriched) workflow skeletons of slow requests and 2) the lineage of expectation modifications needed to bin slow requests into the group it was observed in. Engineers can examine these enriched skeletons and the lineage of expectations modifications to inform their diagnosis efforts. We expect that once Pythia has converged to a set of instrumentation choices for a problem, the last expectation modification (i.e., last set of instrumentation enabled) will be invaluable in identifying the root cause. Engineers can also influence future cycles by modifying expectations (e.g., to ignore variance in locations where it is unavoidable [27]) or by specifically asking certain trace points to be enabled or disabled.

3 HOW PYTHIA COULD AID DIAGNOSIS

We discuss how Pythia's approach would enable the instrumentation needed to help diagnose two different problems. We assume initial expectations that state that requests of the same type that access the same services should perform similarly.

In Section 5, we discuss how we validated Pythia's approach by using it to debug the **contention** problem. Our problems involve OpenStack [24], a distributed application for managing clouds and Ceph, a distributed storage application [34].

Contention in OpenStack: We run OpenStack on high-capacity machines, but SERVER CREATE requests still finish creating new OpenStack VMs very slowly. The root cause is that the number of concurrent servers that can be created within OpenStack's Nova Service is limited to ten by default. Additional SERVER CREATE requests are forced to wait on a semaphore.

For this problem, Pythia will initially bin critical-path skeletons into groups based on their type (e.g., SERVER CREATE, DELETE) and the Openstack services they access (e.g., Keystone, Nova, Glance). Pythia will identify groups containing SERVER CREATE requests as exhibiting high performance variation. This is because SERVER CREATEs received during periods of low concurrency will execute immediately, whereas others will have to wait varying amounts of time for the semaphore.

Pythia will localize the variation to the machines involved in the Nova service and start to enable more granular instrumentation in them. It will eventually enable trace points within the function that contains the semaphore. It will find that the queue length variable that could be exposed within these trace points explains the observed variation. This will give engineers a strong starting point to identify the problem's root cause.

Excessive disk seeks in Ceph: Ceph [34], configured with certain erasure-coding options, will service READ operations that require disk accesses very slowly. The root cause is that Ceph implements these erasure coding options via very small stripe sizes. As a result, a large number of disk seeks are needed to service large READs involving many stripes.

Pythia will initially bin critical-path skeletons into groups based on their type (e.g., READ or WRITE) and the Ceph services they access (e.g., metadata service, storage service). It will find that groups containing READ requests exhibit high variation because some hit in storage nodes' caches and others miss. It will eventually enable instrumentation that allows READ requests to be binned into different groups based on whether they hit or miss in cache. Next, Pythia will identify that READ requests that miss in cache are extremely slow. In the process of hierarchically enabling instrumentation, it will find that the dominant contributor to response times for these requests are small random access I/O operations to the disk.

4 PYTHIA'S DESIGN

Figure 3 shows our proposed design for Pythia. It can be separated into a *control plane* and an *instrumentation plane*. Components in the instrumentation plane are provided by different workflow-centric tracing infrastructures, and the components in the control plane form the crux of Pythia's functionality. All of the control plane components and the trace-reconstruction component can be implemented as scalable big-data jobs that operate close to real time. (See Chothia et al. [9] for how trace reconstruction can be done in close to real time.) We focus on previously undiscussed aspects of the control plane below.

Control-plane input: Pythia's control-plane takes as input traces in the form of directed-acyclic graphs, which can express concurrency and synchronization. Pythia also requires input traces to state hierarchical caller/callee relationships

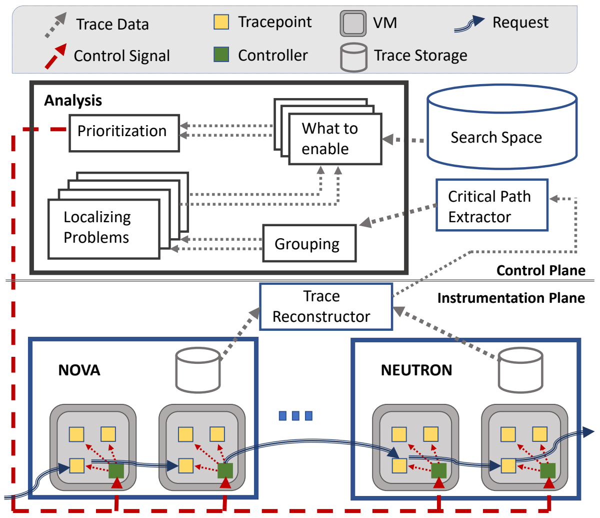


Figure 3: Pythia's design. In this diagram, Pythia is deployed to control instrumentation in OpenStack.

between semantically-meaningful intervals of the trace, also called *spans* [25, 30]. Example spans may include executions of distributed-application services, API calls, kernel system calls, or functions. Capturing hierarchical relationships in traces allow us to explore search-space representations and search strategies that make use of the hierarchy to guide instrumentation decisions. Today, many open-tracing compatible tracing infrastructures output traces as DAGs: they preserve the hierarchy and contain rudimentary support for preserving concurrency and synchronization. Other tracing infrastructures that only output traces as trees of hierarchical relationships can be converted to DAGs without loss of generality.

Prioritization component: This component is responsible for keeping enabled instrumentation to a pre-set limit (e.g., under x KB/s). After instrumentation choices are made for different groups in each cycle, this component will rank groups as per a measure of how problematic they are (e.g., highest CoV groups first, then groups with extremely slow performance). It will then enable instrumentation for groups in rank order until the budget is reached or no groups are left. In doing so, this component must be careful not to double count cases where multiple groups desire the same instrumentation. The prioritization component will always allow instrumentation to be disabled for groups. If the budget threshold is reached during any particular cycle, this component may kick off a garbage-collection round or may ask lower-ranked groups to disable some of their instrumentation.

4.1 Search space

All of the instrumentation Pythia can control is specified by the search space, which has three goals: (1) identify additional instrumentation that could be enabled within an area of the system, (2) avoid spurious instrumentation choices that may mislead Pythia, and (3) not overly restrict instrumentation choices.

We are currently exploring the utility of a forest of calling context trees (CCTs) [1], shown in Fig. 4, as the search space because they guarantee every path from root to leaf is a valid path in the system (goal #2). Nodes of our calling context trees are spans; edges represent caller/callee relationships.

The roots of our CCTs either represent unique application request types (e.g., NOVA BOOT or SERVER LIST) or entry points to code in lower data-center stack layers (e.g., Read or Write system calls in the kernel). Spans that represent calls to lower layers are also kept in higher-layer CCTs e.g., span E in Fig. 4.

To identify additional instrumentation that could be enabled for a high-variance or high-latency span in a problematic group (goal #1), we plan to use CCTs by first indexing into the correct CCT using the request-type of the groups' critical-path skeleton representative. Second, we will use an index to search for the problematic span in the CCT. If there are multiple matches, we will use an edit-distance algorithm to compare the calling contexts of the matches against that of the high-latency/high-variance span in the critical-path skeleton. We will choose the match with the lowest edit distance and return all of its descendents as the instrumentation that could be enabled.

Constructing CCTs: Exhaustive workloads in which all code paths are exercised can be run on the application and lower the kernel independently. If all trace points are enabled when running these workloads, the resulting traces can be merged to create our CCTs. Note that with this approach, goal #3 may not be met if the workloads are not exhaustive.

4.2 Search strategies

For both groups with high variance and high latency, Pythia will use a set of search strategies in combination with the search space to determine what instrumentation to enable. We discuss a few possible strategies here: *Hierarchical Search:* One strategy, as explained in § 2, would be to enable all trace points that are direct children of the problem edge in the CCT. This approach investigates high variation layer by layer; it is thorough, but brings high overhead. *Binary Search:* Hierarchical search can be accelerated by skipping layers when exploring. For Fig. 4, if variance/latency is localized to span A, spans C and D can be enabled next, thus only requiring the exploration of span B is C and D are healthy. *Cross Layer:* In a system with cross-layer instrumentation, instrumentation on different stack levels may be prioritized after the problem has been localized to a single VM, to establish which stack level is likely to contain the root cause of a problem. *Covariances:* For high variance groups, the covariances of edge pairs will be calculated, and if they are significant contributors to overall variance, trace points common to both edges are prioritized in the search space.

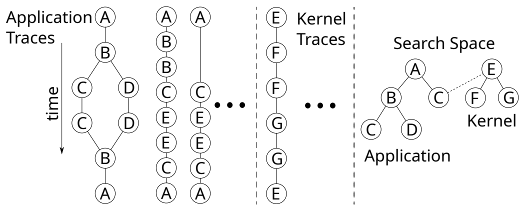


Figure 4: An example search space. The DAGs on the left hand side are traces obtained during profiling. The search space is on the right side. The dashed line indicates a reference to a separate CCT for the kernel. Span C has two nodes based on the calling context.

The search space includes variable values that can be instrumented as well. If variables are chosen for instrumentation, Pythia inspects whether they are useful for explaining variance. We use Canonical Correlation Analysis (CCA) [15] to find the variables most highly correlated with overall request latency. If correlation is significant, these variables are included in the traces as possible explanations of variance. The grouping also considers these variables, e.g., if a binary variable is a good indicator of request latency, then two groups are constructed for either value of the binary variable. Tracing for variables that do not show high correlation with variance are disabled.

5 VALIDATING PYTHIA'S APPROACH

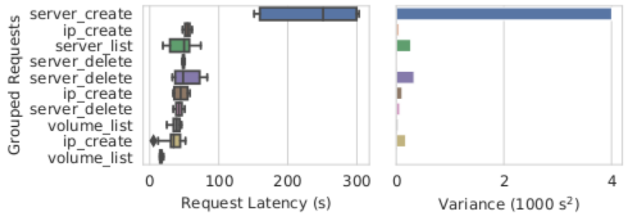
In this section we validate Pythia's approach by manually debugging the **contention** problem described in § 3 using the concepts described in previous sections. We show that *1)* existing instrumentation in OpenStack is not enough to diagnose this problem or similar problems, *2)* localizing variance to determine where instrumentation is needed and using CCA to selecting key-value pairs related to the ongoing performance problem provides insight into the problem, and *3)* grouping based on request type, structure and important variables can separate problematic requests from healthy ones.

Our exploration extends the tracing infrastructure already present in OpenStack and OSProfiler [22] by *1)* adding the capability to enable or disable trace points, *2)* reconstruct traces as DAGs, and *3)* expose queue lengths and similar variables in trace points. We also implemented code that extracts the traces' critical paths, places critical paths with the same request type into a single group, localizes performance variation to specific edges, and finds variables correlated to overall latency using CCA. Our code does not yet use traces' call graph hierarchies because we are not yet incrementally exploring the instrumentation space, but instead inspecting fully-enabled traces.

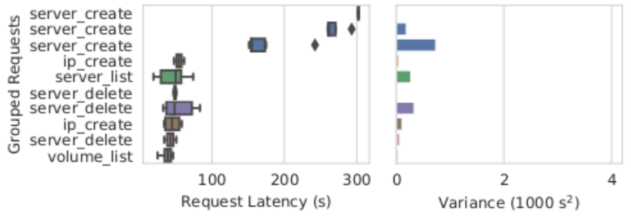
Experimental Setup. We run OpenStack version Pike on a public data center [21], on a VM with 8 vCPUs and 32 GB of memory. We design 3 workloads consisting of combinations of create/list/delete VM/floating IP/volume requests, and start 20 instances of each workload, multiplexed among 8 vCPUs.

Results. We collect traces based on their request type and group their critical paths based on their request type and order of execution of enabled trace points. Our grouping uses trace points that are enabled by default in OpenStack Pike. The grouping we obtain is shown in Fig. 5a. We inspect the SERVER CREATE group, which is the group with the highest variance.

The variance of edge latency for the SERVER CREATE group is highly localized—96% of the variance is within 6 edges. In order to find the root cause of variance, we inspect the edge with the highest variance. This edge corresponds to 7 lines of code, 4 of which are comments. In the remaining 3 lines, a semaphore (NOVA.COMPUTE.MANAGER.COMPUTEMANAGER._BUILD_SEMAPHORE) is acquired. After localizing the variance we add a trace point exposing the number of processes waiting for this semaphore to the traces. After another run, CCA shows that this new variable correlates the most (0.85 with P-value 10^{-5}) with overall request latency. Pythia would use this important variable in grouping, so we manually



(a) Grouping using OpenStack's default instrumentation. Groups with the same name represent requests of the same type with differing workflows.



(b) Grouping of OpenStack requests after enabling new instrumentation. Workflow information used for grouping now includes queue lengths as well—queue lengths of 0, 1-4 and above 5 are separated into three groups.

Figure 5: Each request type is grouped using structure. SERVER CREATE commands comprise a problematic group. Top 10 highest median latency groups are shown.

group the traces based on the value of this variable, as shown in Fig. 5b, finding that the high variance group does indeed get separated into three low-variance groups. Inspecting the initialization of this semaphore shows that the configuration option MAX_CONCURRENT_BUILDS indicates the number of simultaneous VM creations within a single host, explaining the root cause of the high variance in simultaneous VM creations.

6 DISCUSSION & OPEN QUESTIONS

Realizing our vision for Pythia will greatly reduce the amount of time and effort engineers spend diagnosing performance problems. Pythia will also improve the utility of the many diagnosis tools that use pre-existing logs or traces for performance diagnosis [2, 6, 23, 26, 29, 38]. We survey some important questions on the path to achieving our vision.

First, how detailed do initial expectations need to be for Pythia to converge to useful instrumentation choices quickly? *Second*, how can we minimize the amount of time Pythia must wait between cycles before making new instrumentation decisions? *Third*, are techniques other than sticking to an instrumentation budget needed to reduce the perturbation in performance Pythia may induce? How should the instrumentation budget be expressed by the developers? *Fourth*, out of the many possible search-space representations and search strategies, which ones are most useful?

7 SUMMARY

It is challenging to decide where instrumentation should be enabled to diagnose performance problems in distributed applications. We presented initial steps toward creating an automated instrumentation framework that can explore the search space automatically.

Acknowledgements:The authors would like to thank Mema Roussopoulos for shepherding the paper, and the anonymous reviewers for their insightful feedback. This research is supported by the National Science Foundation under Grant No. CNS-1815323 and by Red Hat, Inc.

REFERENCES

[1] Glenn Ammons, Thomas Ball, and James R. Larus. 1997. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI '97)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/258915.258924

[2] Dan Ardelean, Amer Diwan, and Chandra Erdman. 2018. Performance Analysis of Cloud Applications. In *NSDI '18: Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation*.

[3] Piramanayagam Nainar Arumuga and Ben Liblit. 2010. Adaptive bug isolation. In *International Conference on Software Engineering*. ACM Press, New York, New York, USA, 255–264.

[4] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *OSDI '12: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*.

[5] Thomas Ball and James R Larus. 1996. Efficient path profiling. In *MICRO 29: Proceedigs of the 29th Annual Internatoinal Symposium on Microachitecture*. IEEE/ACM.

[6] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. 2011. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *ESEC/FSE '11: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*.

[7] Bryan M Cantrill and Michael W Shapiro. 2004. Dynamic instrumentation of production systems. In *ATC '04: Proceedings of the 2004 USENIX Annual Technical Conference*.

[8] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, David Patterson, Armando Fox, and Eric Brewer. 2004. Path-based failure and evolution management. In *NSDI '04: Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation*.

[9] Zaheer Chothia, John Liagouris, Desislava Dimitrova, and Timothy Roscoe. 2017. Online reconstruction of structural information from datacenter logs. In *EuroSys '17: Proceedings of the 12th ACM SIGOPS European Conference on Computer Systems*.

[10] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. 2015. Log^2: A cost-aware logging mechanism for performance diagnosis. In *ATC '15: Proceedings of the 2015 USENIX Annual Technical Conference*.

[11] Ulfar Erlingsson, Marcus Peinado, Simon Peter, and Mihai Budiu. 2011. Fay: extensible distributed tracing from kernels to clusters. In *SOSP '11: Proceedings of the 23nd ACM Symposium on Operating Systems Principles*.

[12] Rodrigo Fonseca, Michael J. Freedman, and George Porter. 2010. Experiences with tracing causality in networked services. In *INM/WREN '10: Proceedings of the 1st Internet Network Management Workshop/Workshop on Research on Enterprise Monitoring*.

[13] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. 2007. X-Trace: a pervasive network tracing framework. In *NSDI '07: Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*.

[14] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. 2009. Debugging in the (very) large: ten years of implementation and experience. In *SOSP '09: Proceedings of the 22nd Symposium on Operating Systems Principles*.

[15] Harold Hotelling. 1936. Relations Between Two Sets of Variates. *Biometrika* 28, 3/4 (1936), 321–377. http://www.jstor.org/stable/2333955

[16] Jaeger [n.d.]. Jaeger: open-source, end-to-end distributed tracing. https://www.jaegertracing.io.

[17] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An end-to-end performance tracing and analysis system. In *SOSP '17: Proceedings of the 26th Symposium on Operating Systems Principles*.

[18] Ben Liblit, Alex Aiken, Alice X Zheng, and Michael I Jordan. 2003. Bug isolation via remote program sampling. In *PLDI '03: Programming Language Design and Implementation*. ACM.

[19] Jonathan Mace and Rodrigo Fonseca. 2018. Universal context propagation for distributed system instrumentation. In *EuroSys'18: Proceedings of the Thirteenth EuroSys Conference*.

[20] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot Tracing: dynamic causal monitoring for distributed systems. In *SOSP '15: Proceedings of the 25th Symposium on Operating Systems Principles*.

[21] Massachusetts Open Cloud. 2019. http://massopen.cloud.

[22] Mirantis OSProfiler [n.d.]. OSProfiler. https://docs.openstack.org/osprofiler/latest/.

[23] Karthik Nagaraj, Charles Killian, and Jennifer Neville. 2012. Structured comparative analysis of systems logs to diagnose performance problems. In *NSDI '12: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*.

[24] Openstack [n.d.]. OpenStack web site. https://www.openstack.org.

[25] OpenTracing website [n.d.]. OpenTracing website. http://opentracing.io/.

[26] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul Shah, and Amin Vahdat. 2006. Pip: detecting the unexpected in distributed systems. In *NSDI '06: Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*.

[27] Raja R. Sambasivan and Gregory R. Ganger. 2012. Automated diagnosis without predictability is a recipe for failure. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing*. USENIX Association, 21–21.

[28] Raja R. Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H Sigelman, Rodrigo Fonseca, and Gregory R. Ganger. 2016. Principled workflow-centric tracing of distributed systems. In *SoCC '16: Proceedings of the Seventh Symposium on Cloud Computing*.

[29] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. 2011. Diagnosing performance changes by comparing request flows. In *NSDI'11: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*.

[30] Benjamin H. Sigelman, Luiz A. Barroso, Michael Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a large-scale distributed systems tracing infrastructure*. Technical Report dapper-2010-1. Google.

[31] Skua: Extending end-to-end tracing into the Linux Kernel 2018. Skua: Extending end-to-end tracing into the Linux Kernel. https://devconfus2018.sched.com/event/FzVg.

[32] The Apache Hadoop Distributed File System 2013. The Apache Hadoop Distributed File System. http://hadoop.apache.org/hdfs/.

[33] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R. Ganger. 2006. Stardust: tracking activity in a distributed storage system. In *SIGMETRICS '06/Performance '06: Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*.

[34] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell D E Long, and Carlos Maltzahn. 2006. Ceph: a scalable, high-performance distributed file system. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*.

[35] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be conservative: enhancing failure diagnosis with proactive logging. In *OSDI' 12: Proceedings of the 10th conferences on Operating Systems Design & Implementation*.

[36] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2012. Improving software diagnosability via log enhancement. *ACM SIGPLAN Notices* 47, 4 (June 2012), 3–14.

[37] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *SOSP '17: Proceedings of the 26th Symposium on Operating Systems Principles*.

[38] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. 2016. Non-Intrusive Performance Profiling for Entire Software Stacks Based on the Flow Reconstruction Principle. In *OSDI '16: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*.

[39] Zipkin [n.d.]. Zipkin. http://zipkin.io/.

[40] Zhiqiang Zuo, Lu Fang, Siau-Cheng Khoo, Guoqing Xu, Shan Lu, Lu Fang, Siau-Cheng Khoo, and Guoqing Xu. 2016. Low-overhead and fully automated statistical debugging with abstraction refinement. In *OOPSLA '16: Proceedings of the ACM international conference on Object oriented programming systems languages and applications*.