App–Bisect: Autonomous Healing for Microservice-based Apps

Shriram Rajagopalan Hani Jamjoom

{shriram, jamjoom}@us.ibm.com

IBM T. J. Watson Research Center, New York

Abstract

The microservice and DevOps approach to software design has resulted in new software features being delivered immediately to users, instead of waiting for long refresh cycles. On the downside, software bugs and performance regressions have now become an important cause of downtime. We propose *app-bisect*, an autonomous tool to troubleshoot and repair such software issues in production environments. Our insight is that the evolution of microservices in an application can be captured as mutations to the graph of microservice dependencies, such that a particular version of the graph from the past can be deployed automatically, as an interim measure until the problem is permanently fixed. Using canary testing and version-aware routing techniques, we describe how the search process can be sped up to identify such a candidate version. We present the overall design and key challenges towards implementing such a system.

1 Introduction

Web applications hosted in the platform as a Service (PaaS) clouds are being re-architected to be microservice-oriented [24, 35]. In addition, developers and system administrators are adopting new software practices such as continuous integration and deployment of new features instead of periodic upgrade cycles [19–23]. Such practices have collectively come to be termed as *DevOps*. The frequency of such deployments varies anywhere from few times a week to 50 times a day [15]. In order to continuously incorporate user feedback, application owners now choose to *release early and release often* foregoing rigorous testing in exchange for hastening the time to market a product feature. In such a rapidly evolving application deployment, bugs and performance regressions occur more often than usual. Application availability is impacted because debugging modern cloud-based distributed applications is often a time consuming task.

There are many performance profiling tools [1,3–5,10, 16,29–31,37] to help developers identify software issues causing degradation in throughput and response times to the end user. However, they provide only monitoring capabilities. There are also design patterns like circuit breakers [13, 36] that enable distributed applications to prevent transient and component-level errors from cascading. Persistent failures, those stemming from software bugs and under-tested features, continue to require human intervention. Instead of forcing applications to function poorly while waiting for a software update, we argue that an interim measure is possible. Specifically, *applications should be automatically reverted to an older version that was known to provide a better end user experience*.

Inspired by autonomous computing [7,9,11,12,14,17], we present *app-bisect*[1]: an autonomous system service that can identify and heal microservice-based applications deployed in the cloud. By representing the application as a graph of microservices that evolve over time, any update to a microservice can be viewed as a mutation of the graph. Typical of canary testing, app-bisect systematically tests previous mutations by deploying them in production and diverting a portion of user traffic to the various versions. On finding the least destructive combination of versions, one that offers the desired end user experience, app-bisect diverts all user traffic to it until the human operator intervenes.

Time traveling through the "changelog" of an application's persistent state is not a new concept. Chronus [18] helps debug failures caused by configuration errors in single-machine applications, by performing a binary search over previously recorded states of the application, to locate the point in time when the configuration error was made. App-bisect, on the other hand, targets distributed applications, searching through the updates to

[1]The name app-bisect is inspired by the git-bisect tool that is used by developers find the commit that introduced a bug

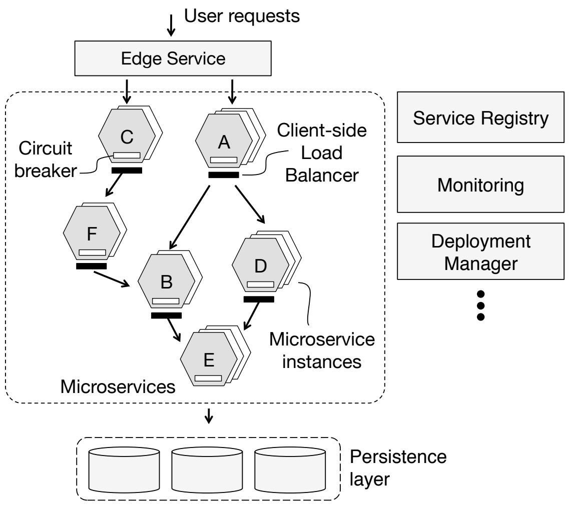

Figure 1: A simplified view of a microservice-based application deployment.

the service dependency graph. Debugging is a secondary goal. Business continuity is its primary focus, with the intent of bringing the application or its older version back online.

Service oriented architectures (SOA) in the past have attempted to use autonomic service substitution [2, 6] in case of a service failure. However, these approaches resort to looking for substitutes providing the same service from the current pool of executing services. In a microservices environment, functionality is rarely duplicated. At the same time, compared to a SOA application, the microservice-based application is highly dynamic, with new services, features and fixes being constantly deployed. Autonomous healing in this scenario faces a different set of challenges compared to prior work in SOA systems.

In the following sections, we discuss the scope, design and implementation challenges towards realizing an autonomous system-level service for troubleshooting and healing microservice-based applications.

2 Target Environment & Fault Model

Figure 1 illustrates a simplified view of a microservice deployment. We are primarily focused on applications deployed on Platform-as-a-service (PaaS) clouds [32–34]. We assume that applications follow the microservices and DevOps principles as described below. Our focus is primarily on user facing applications where end user's experience is the key indicator of application performance.

Microservices and DevOps. In the microservice approach, applications are structured as a loosely-coupled collection of small, well-defined stateless services that communicate with each other only through well-defined APIs. As exemplified by the Netflix architecture [35], this approach simplifies the design of scalable and robust cloud applications. Typical applications are composed of hundreds of instances of heterogeneous microservices, backed by scalable data stores.

As shown in Figure 1, services register themselves with a registry service like Zookeeper [8] that can be used for service discovery as well. The service discovery node enables services to discover the locations of other services. For scalability purposes, clients (services) periodically fetch the locations of other services in the system. When invoking other services, load balancing is typically performed at the client-side instead of querying the centralized registry for every invocation.

From a code development and operations perspective, each microservice has its own branch with updates to the service being committed to that branch. Developers and operations personnel work closely to quickly test and deploy new features, monitor user experience in controlled fashion using techniques like canary testing, and finally incorporate changes based on monitoring into the next iteration. The tight feedback loop created by this process enables software to evolve faster in response to user needs [19–23].

Fault model. We take a high-availability approach to our problem by treating a performance degradation as application downtime. For the purposes of our discussion, performance is purely associated with end user experience. In other words, we consider metrics like end-to-end response time and application throughput. We treat an application experiencing *prolonged periods* (on the order of tens of minutes) of performance loss as a failure.

We do not consider transient issues such as temporary drops in performance while one or more services are scaling-out, spurious timeouts, hard to reproduce non-deterministic bugs, etc. However, we take into account scenarios where there is a high frequency of errors when invoking a microservice's API, frequent crashes of a microservice, or other recurring error events in log files.

Our fault-model does not include scenarios where the performance issue is caused by failures in the underlying infrastructure such as hardware failures, network partitions, transient link congestion, etc. For example, if there is a long outage in a certain section of the data center, application performance issues in the affected portion of the data center are ignored until the system issues are resolved.

Detection & Recovery. Performance management requires monitoring, analyzing and maintaining the performance of an application. In a standard setup, the data from the monitoring subsystem is continuously fed into a real-time analytics subsystem. The analytics subsystem generates alerts whenever it observes performance issues. The maintenance subsystem acts on the alerts, typically by launching new instances (i.e., auto scaling) or alerting the human operator. The mean time to recover (MTTR) an application from a performance bug depends on how quickly the human operator responds to the issue. If the alert was raised at an hour when the operator is not available (e.g., midnight), the time to repair (and hence recover) is high, resulting in loss of traffic and revenue. Our goal is to minimize the MTTR by routing traffic through a previous version of the application where the issue is not present.

3 Design & Challenges

We introduce *app-bisect*, a system-level application-agnostic tool to react to performance issues that arises in microservice applications. App-bisect is intended to operate in unsupervised mode. The high-level operation of app-bisect is as follows: upon activation, it systematically tests various past-versions of the application's microservices by co-deploying them along with their present-version counter parts, until it finds a combination where the performance issue is not present. It then decommissions all other versions, effectively rolling back parts of the application to the past.

As an autonomous system tasked with maintaining application performance, app-bisect needs to know when to repair an application and when not to. By operating at the system-level, app-bisect leverages knowledge of both the application deployment as well as the underlying infrastructure. This visibility enables app-bisect to repair applications only when the performance issue can be attributed to the application and not the underlying infrastructure.

In the remainder of this section, we describe various design points, the challenges and related implementation details.

3.1 A World of Dependencies

The application's response to the user is a composition of outputs from various microservices constituting the application. While the application certainly depends on all its microservices, individual microservices may also depend on each other for their functions. Given that the microservices can be developed and updated independently, when updates are made to a microservice's data model or API, developers maintain backward compatibility by continuing to support the previous version until all dependent microservices are updated.

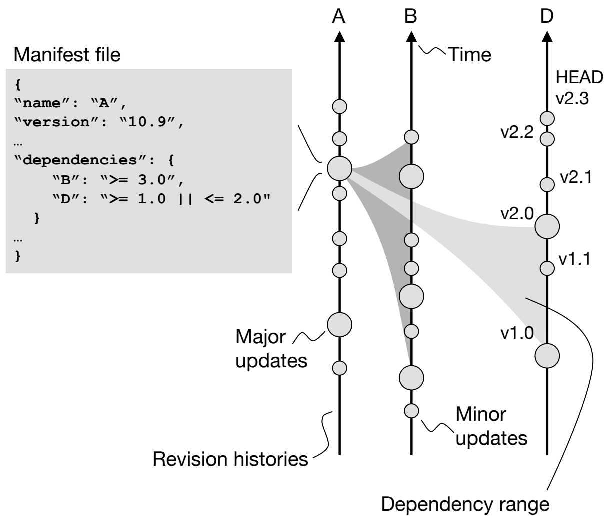
Manifest file
```
{
“name”: “A”,
“version”: “10.9”,
…
“dependencies”: {
    “B”: “>= 3.0”,
    “D”: “>= 1.0 || <= 2.0"
  }
…
}
```

Figure 2: Dependencies across microservices. Microservice A depends on B and D. A given version of service A may be compatible with only a specific set of versions of B and D. Such dependencies are also commonly found among packages in various Linux distributions.

During build or deployment time, every microservice specifies its dependencies on the minimum required version of other microservices in the application (Figure 2). These dependencies are provided in manifest files and are already enforced in several platform as a service clouds [32–34]. With every new update to a microservice, its version number in the manifest file is updated. The PaaS platform deploys the instances of respective versions of microservices accordingly. At runtime, microservices advertise their versions and discover the presence of appropriate dependents using systems like Zookeeper [8] that enable service registration and discovery.

Treating the application as a Linux installation. When deploying an older version of one microservice, older versions of other dependent services may need to be deployed as well. App-bisect takes a *package management* approach used commonly in various Linux distributions. It records the microservice dependencies and the application topology at every update to a microservice. When a previous version of one microservice is deployed, it also deploys appropriate versions of other dependent microservices with the *latest* version possible. As shown in Figure 2, the dependency on a range of versions enables app-bisect to start with the latest possible version that may include additional bug fixes compared to the earliest possible version.

3.2 A Seemingly Straightforward Solution

A straightforward approach to restoring application performance is to identify the root cause, the exact update to a microservice that resulted in the current performance degradation. The microservice is reverted to a version prior to the update. At the same time, other dependent microservices in the application are reverted to the latest version possible while maintaining overall compatibility.

Unfortunately, it is non-trivial to trace the root cause of performance degradation in a microservice-based deployment. Even well engineered data center scale applications like Facebook [4] and LinkedIn [10] resort to sophisticated techniques to identify the root cause.

It is relatively easy to test a simple three-tiered web application where the cause and effect can be observed. For example, ordering for an item would result in changes to the database immediately. However microservice-based applications are typically event-based. Worker services take items off a task queue and service them one after another. In such a system, how does one correlate the action and the eventual result? There is no cause-effect correlation in such systems. A problem that manifests in one microservice may have been caused by another service in the call chain. At other times, a bug introduced in an update to one service may manifest only after other microservices are updated to a point where they start activating the bug prone code path.

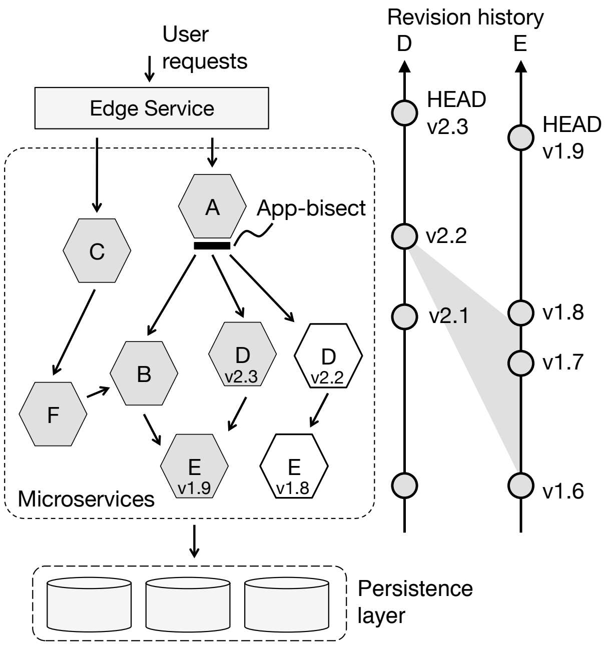

Figure 3: App-bisect in action. Revision histories of services D & E are shown on the right. Microservice D is being tested with an alternate version (v2.2). Version 2.2 of D is compatible up to v1.8 of microservice E. App-bisect deploys the latest version of service E (v1.8). It then routes portion of request traffic between the two versions of service D.

3.3 Playing Hide-n-Seek with Canaries

App-bisect takes an *inverted canary testing* approach to auto healing. It starts by identifying the most recently updated microservice. As shown in Figure 3, it deploys the previous version of the microservice and routes portion of the user traffic across the previous version, very similar to the canary testing process conducted in modern web applications when testing new features. App-bisect's philosophy is to not attempt to identify the root cause. Rather, it focuses on identifying a deployment graph that does not exhibit the performance degradation.

Is this searching for a needle in a haystack? Theoretically, in an application with n microservices with m updates to each service, the search space of all possible deployment combinations is $O(n^m)$. Deploying and testing each version is infeasible and beats the purpose of a fast auto-response tool. Fortunately, the search space can be drastically pruned by taking into account the dependencies across microservices.

Help! My app is back to Hello World! App-bisect continues to test various past versions until it finds one that meets the performance requirements specified in the performance monitor. However, the reader may ask *how far back in time will app-bisect go?* With every rollback, there is potential for loss of features that were exposed, until now, to the end user. The application owner can bound the search process; specifically, as shown in Figure 4, lower bounds, known as *global restore points* can be created to signify point in time until which the application owner is willing to rollback the application. App-bisect searches through various combinations of the application from the current state until the global restore point.

What about data consistency? One question that arises in this context is what happens to the state stored in the data stores as essentially different versions of the same microservices are accessing the same data store. Our approach to searching for a substitute version leverages the canary testing practice used by modern web applications. Canary testing is typically conducted directly in production deployments. Hence the microservices are engineered from the very beginning to handle scenarios where multiple versions can co-exist, accessing the same data store backends. App-bisect piggybacks on this capability to provide a gradual feature downgrade in certain parts of the application. One drawback of this approach is that the search is limited to microservice versions that do not involve changes to the data model in the data store.

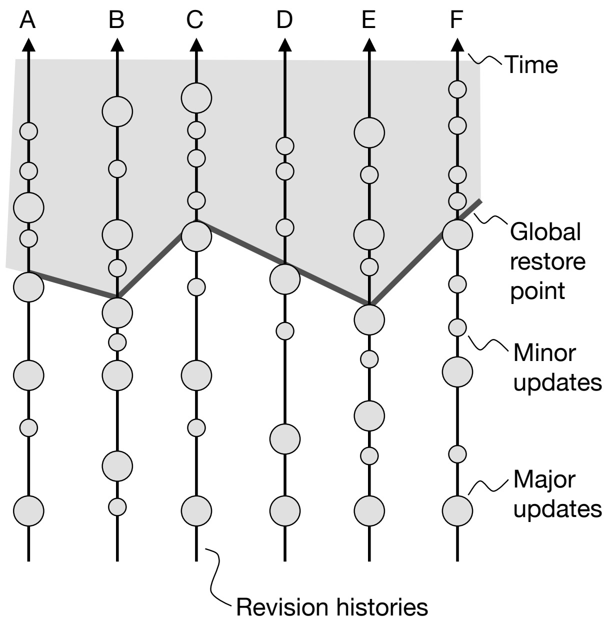

Figure 4: Search space of updates can be pruned with restore points and version dependencies.

3.4 Orchestrating Information Flow

At any given point in time during the search, there are two or more versions of multiple microservices in the application. Requests have to flow through a specific chain of microservices, where the dependencies are satisfied. This may not seem like a challenge as an application capable of handling canary testing would certainly have this capability built into it. However, app-bisect does not have control over the application layer code. Hence, it has to control the flow of information, such only a particular chain of specifically picked versions of microservices handle all aspects of a user request.

In order to route requests through a specific chain of microservices, app-bisect leverages the software defined networking substrate in public cloud data centers [25–28] networks to achieve *version-aware routing*. The combination of host IP address and the edge-switch port number can be used to uniquely identify a particular microservice and its respective version. App-bisect uses this information to setup flow forwarding rules that route requests through a particular chain of microservices.

Sharing microservices between deployments. During the search process, two chains of microservices being tested may have one or more microservices in common. While it may be operationally efficient to share such instances of such microservices, version-aware routing becomes hard. Specifically, if two microservice call chains diverge from a given microservice, app-bisect cannot decide where to route egress requests. It requires application layer support to perform such intelligent routing. On the other hand, as shown in Figure 3, when two microservice call chains converge to the same final set of microservices, app-bisect can reuse those microservice instances while still being able to route requests in a version-aware manner.

3.5 The Search for an Alternate Reality

A version of the application needs to be available always while app-bisect deploys, tests and destroys previous versions of the candidate microservices in the application. We chose to let the original (latest) version of the application remain operational for this purpose. Alternatively, a version corresponding to the restore point version could be deployed at the risk of unnecessarily losing features and bug fixes that are unrelated to the component performing poorly.

When a restore point is available, to speed up the search, app-bisect performs a binary search between the restore point and the version of the application corresponding to the latest update. As shown in Figure 4, the search algorithm starts by picking a random update in the highlighted search region. If a restore point is not available, app-bisect starts from the most recent update to a microservice. The alternate version of the microservice and its corresponding dependencies are deployed. As described earlier (and shown in Figure 3), using canary testing techniques, a portion of incoming traffic is redirected to the alternate version of the application deployment and its performance is monitored over a small period of time (e.g., 1-2 minutes).

If the alternate version does not trigger alerts from the performance monitoring tool, app-bisect proceeds to search the second half of the commit history between the alternate version and the latest version. If the alternate version also performs poorly, app-bisect proceeds to search the first half of the commit history between the restore point and the alternate version. Throughout the course of the search, the latest version of the application remains deployed, ensuring that some version of the application is always available to the user until an alternative version with better performance is found.

Sacrificing efficiency for speed. When a restore point is not provided, the search process can take a very long time. App-bisect parallelizes the search by testing multiple deployments simultaneously. This approach can be thought of as an *n-ary* version of the typically binary style canary testing. The upside to this approach is that the triage can be completed quickly, thereby reducing the impact on application availability.

4 Conclusion

Release early, release often, and listen to your consumers[2] is the philosophy that is driving the fast adoption of microservices and DevOps principles among the developer community. However, its not all free lunch. While the common web application has now become a distributed resilient application, the complexity of troubleshooting issues has also spread from a single machine to a deployment spanning data centers. App-bisect is an attempt to tackle this complexity based on futuristic visions of autonomous computing.

5 Discussion Topics

Controversial Points: Feature upgrades tend to be pushed out much more frequently nowadays than they used to be in the past. In this context, there are two controversial parts to this paper. The first is the belief that distributed applications are better off traveling back to a slimmer (feature wise) past rather than sticking to a slow and bloated present. The second controversial part is the notion that an autonomic system can decide to enforce this belief. *What assurances does the developer community need to embrace such autonomic tools in production?*

Feedback: App-bisect cannot not guarantee a bounded completion time for the search process. Neither can a developer when asked to diagnose a software issue whose root causes are unknown. However, we are looking for community feedback on pruning the search space and speeding up the search process through intelligent insights into the application. For example, app-bisect could apply machine learning techniques to continuously understand the changes in performance, as the application evolves. How can this learning be leveraged when troubleshooting the application?

Open Issues: One of the unsolved issues in the paper is how to automatically route requests in a version-aware manner when multiple microservice chains share one or more microservices. As an example, consider the scenario where two microservices (a producer and a consumer) use asynchronous notifications and work queues. When creating instances of different versions of producers and consumers, it is possible that the different producers append tasks to a globally shared queue service like Amazon's Simple Queue Service. Consumer services could end up consuming tasks from incompatible producers, resulting in erroneous execution or data corruption.

[2]A quote by Eric S. Raymond in his 1997 essay "The Cathedral and the Bazaar"

Idea Falling Apart: Microservices can have complex dependencies that span the persistent data stores. Unless the microservices are stateless and data models in the persistent stores can handle downgrades gracefully, app-bisect may not work. As discussed earlier, app-bisect's usefulness depends on its time to discover and isolate bugs. More importantly, app-bisect should not introduce additional failures or corrupt any data.

References

[1] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance Debugging for Distributed Systems of Black Boxes. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)* (2003).

[2] ATHANASOPOULOS, D., ZARRAS, A. V., AND ISSARNY, V. Service Substitution Revisited. In *Proc. of IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2009).

[3] CHEN, M. Y., KICIMAN, E., FRATKIN, E., FOX, A., AND BREWER, E. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Proc. of the International Conference on Dependable Systems and Networks (DSN)* (2002).

[4] CHOW, M., MEISNER, D., FLINN, J., PEEK, D., AND WENISCH, T. F. The Mystery Machine: End-to-End Performance Analysis of Large-Scale Internet Services. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)* (2014).

[5] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. X-trace: A Pervasive Network Tracing Framework. In *Proc. of USENIX Symposium on Networked Systems Design & Implementation (NSDI)* (2007).

[6] FREDJ, M., GEORGANTAS, N., ISSARNY, V., AND ZARRAS, A. Dynamic Service Substitution in Service-Oriented Architectures. In *IEEE Congress on Services* (2008).

[7] GHOSH, D., SHARMAN, R., RAO, H. R., AND UPADHYAYA, S. Self-healing Systems: Survey and Synthesis. *Decision Support Systems 42*, 4 (2007).

[8] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proc. of USENIX Annual Technical Conference (ATC)* (2010).

[9] KEPHART, J. O., AND CHESS, D. M. The Vision of Autonomic Computing. *Computer 36*, 1.

[10] KIM, M., SUMBALY, R., AND SHAH, S. Root Cause Detection in a Service-Oriented Architecture. In *Proc. of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2013).

[11] KRAMER, J., AND MAGEE, J. Self-managed Systems: An Architectural Challenge. In *IEEE Future of Software Engineering (FOSE)* (2007).

[12] MINSKY, N. H. On Conditions for Self-Healing in Distributed Software Systems. In *Proc. of Autonomic Computing Workshop* (2003).

[13] NYGARD, M. T. *Release It! Design and Deploy Production-Ready Software*. The Pragmatic Programmers, 2007, ch. 5.2.

[14] SALEHIE, M., AND TAHVILDARI, L. Autonomic Computing: Emerging Trends and Open Problems. In *ACM SIGSOFT Software Engineering Notes* (2005), vol. 30.

[15] SCHAUENBERG, D. Development, Deployment and Collaboration at Etsy. In *International Software Development Conference (QCon)* (March 2014).

[16] SIGELMAN, B. H., BARROSO, L. A., BURROWS, M., STEPHENSON, P., PLAKAL, M., BEAVER, D., JASPAN, S., AND SHANBHAG, C. Dapper, A Large-Scale Distributed Systems Tracing Infrastructure. *Google research* (2010).

[17] TRUMLER, W., PIETZOWSKI, A., SATZGER, B., AND UNGERER, T. Adaptive Self-Optimization in Distributed Dynamic Environments. In *Proc. of IEEE International Conference on Self-Adaptive and Self-Organizing Systems* (2007).

[18] WHITAKER, A., COX, R. S., AND GRIBBLE, S. D. Configuration Debugging As Search: Finding the Needle in the Haystack. In *Proc. of USENIX Symposium on Operating Systems Design & Implementation (OSDI)* (2004).

[19] BARCLAYS BANK. Scaling DevOps. *DevOps Enterprise Summit (2014).*

[20] WALT DISNEY. Disney DevOps - To Infinity and Beyond. *DevOps Enterprise Summit (2014).*

[21] MACY'S. Transforming traditional Enterprise Software Development Processes by Applying DevOps and Agile principles at Scale. *DevOps Enterprise Summit (2014).*

[22] PNC BANK. DevOps: From the Center Out. *DevOps Enterprise Summit (2014).*

[23] USCIS. How DevOps Can Fix Federal Government IT – US Citizenship and Immigration Services. *DevOps Enterprise Summit (2014).*

[24] THOUGHTWORKS. Real-World Microservices: Lessons From the Frontline. *YOW! Australian Developer Conference*, September 2014.

[25] JAMES HAMILTON, AMAZON. AWS Innovation at Scale. *AWS re:Invent Conference (2014).* [ONLINE].

[26] AMIN VAHDAT, GOOGLE. Enter the Andromeda Zone – Google Cloud Platform's Latest Networking Stack. `http://googlecloudplatform.blogspot.com/2014/04/enter-andromeda-zone-google-cloud-platforms-latest-networking-stack.html`. [ONLINE].

[27] IBM. Managing and Monitoring Network Resources – IBM Cloud OpenStack Services. `http://open.ibmcloud.com/documentation/managing-and-monitoring-network-resources.html`. [ONLINE].

[28] RACKSPACE. Software Defined Networking in the Cloud – Rackspace Cloud Networks. `http://www.rackspace.com/cloud/networks`. [ONLINE].

[29] AppDynamics, Inc. `http://www.appdynamics.com/`. [ONLINE].

[30] NETFLIX. A Microscope on Microservices. `http://techblog.netflix.com/2015/02/a-microscope-on-microservices.html`, February 2015. [ONLINE].

[31] New Relic, Inc. `http://www.newrelic.com/`. [ONLINE].

[32] Cloud Foundry. `http://www.cloudfoundry.org/`. [ONLINE].

[33] IBM BLUEMIX. Platform as a Service. `www.ibm.com/software/bluemix`. [ONLINE].

[34] PIVOTAL. Platform as a Service. `http://pivotal.io/platform-as-a-service/pivotal-cloud-foundry`. [ONLINE].

[35] NETFLIX. Open Source Software Center. `http://netflix.github.io`. [ONLINE].

[36] HYSTRIX. Latency and Fault Tolerance for Distributed Systems. `https://github.com/Netflix/Hystrix/`. [ONLINE].

[37] ZHAO, X., ZHANG, Y., LION, D., FAIZAN, M., LUO, Y., YUAN, D., AND STUMM, M. lprof: A Nonintrusive Request Flow Profiler for Distributed Systems. In *Proc. of USENIX Symposium on Operating Systems Design & Implementation (OSDI)* (2014).