From the Outside Looking In: Probing Web APIs to Build Detailed Workload Profiles

Nan Deng, Zichen Xu, Christopher Stewart and Xiaorui Wang
The Ohio State University

Abstract

Cloud applications depend on third party services for features ranging from networked storage to maps. Web-based application programming interfaces (web APIs) make it easy to use these third party services but hide details about their structure and resource needs. However, due to the lack of implementation-level knowledge, cloud applications have little information when these third party services break or even unproperly implemented. This paper outlines research to extract workload details from data collected by probing web APIs. The resulting workload profiles will provide early warning signs when web APIs have broken component. Such information could be used to build feedback loops to deal with possible high response times of web APIs. It will also help developers choose between competing web APIs. The challenge is to extract profiles by assuming that the systems underlying web APIs use common cloud computing practices, e.g., auto scaling. In early results, we have used blind source separation to extract per-tier delays in multi-tier storage services using response times collected from API probes. We modeled median and 95^{th} percentile delay within 10% error at each tier. Finally, we set up two competing storage services, one of which used a slow key-value store. We probed their APIs and used our profiles to choose between the two. We showed that looking at response times alone could lead to the wrong choice and that detailed workload profiles provided helpful data.

1 Introduction

Cloud applications enrich their core content by using services from outside, third party providers. Web application programming interfaces (web APIs) enable such interaction, allowing providers to define and publish protocols to access their underlying systems. It is now common for cloud applications to use 7 to 25 APIs for features ranging from storage to maps to social networking [13]. For providers, web APIs strengthen brand and broaden user base without the cost of programming new features. In 2013, the Programmable Web API index grew by 32% [7], indexing more than 11,000 APIs.

Web APIs hide the underlying system's structure and resource usage from cloud application developers, allowing API providers to manage resources as they see fit. For example, a storage API returns the same data whether the underlying system fetched the data from DRAM or disk. However, when API providers manage their resources poorly, applications that use their API suffer. Early versions of the Facebook API slowed one application's page load times by 75% [25]. When Facebook's API suffered downtime, hundreds of applications, including CNN and Gawker, went down as well [33]. While using a web API, developers would like to know if the underlying system is robust. That is, will the API provide fast response times during holiday seasons? How will its resource needs grow over time? Depending on the answers, developers may choose competing APIs or use the API sparingly [13, 33].

An API's *workload profile* describes its canonical resource needs and can be used to answer what-if questions. Based on APIs' recent profiles, cloud applications are able to adjust their behavior accordingly to either mask low response times of slow APIs, or take advantage of fast APIs. Prior research on workload profiling used 1) white box methods, e.g., changing the OS to trace request contexts across distributed nodes [22, 26] or 2) black box methods that inferred resource usage from logs [29]. Both approaches would require data collected within a API provider's system but, as third parties, providers have strong incentives to provide only good data about their service. Without trusted inside data, workload profiles must be forged by probing the API and collecting data outside of the underlying system (e.g., client-side observed response times).

In this paper, we propose research on creating workload profiles for web APIs. Taken by itself, data collected by probing web APIs under constrains the wide range of systems that could produce such data. However, when we combined that data with constraints imposed by common cloud computing practices, we have created usable and accurate workload profiles. One cloud computing practice that we have used is auto scaling which constrains queuing delays, making processing time a key factor affecting observed response times. In early work, we have found success profiling processing times with blind source separation methods. Specifically, we used observed response times as input for independent component analysis (ICA) and extracted normalized processing times in multi-tier systems. These per-tier distributions are our workload profiles.

We validated our profiles with a multi-tier storage service. We used CPU usage thresholds to scale out a Redis cache and database on demand. Our profiles captured 50th, 75th and 95th percentile service times within 10% of direct measurements. We showed that our profiles can help developers choose between competing APIs by setting up two storage services. One used Apache Zookeeper as a cache instead of Redis, a mistake reported in online forums [6,8]. Zookeeper is a poor choice for an object cache because it fully replicates content on all nodes. We lowered the request arrival rate for the service with Zookeeper cache such that our API probes observed lower average and 95th percentile response times compared to the other service. These response times could be misleading because the service that used Redis was more robust to increased request rates. Fortunately, our workload profiles revealed a warning sign: Tier 1 processing times on the service using Zookeeper had larger variance than expected. This signaled that too many resources, i.e., not just DRAM on a single node, were involved in processing.

The remainder of this paper is arranged as follows: We discuss cloud computing practices that make web API profiling tractable in Section 2. We make the case for blind source separation methods in Section 3 and then present promising early results with ICA in Section 4. Related work on workload profiling is covered in Section 5. We conclude by discussing future directions for the proposed research.

2 The Cloud Constrains Workloads

Salaries for programmers and system managers can make up 20% of an application's total expenses [32]. Web APIs offer value by providing new features without using costly programmer time. However, slow APIs can drive away customers. Shopping carts abandoned due to slow response times cost $3B annually [14]. Web APIs that increase response times can hurt revenues more than they reduce costs. Developers could use response times measured by probing the API to assess the API's value. However, response times reflect current usage patterns. If request rates or mixes change, response times may change a lot. The challenge for our research is to extract profiles that apply to a wide range of usage patterns.

A key insight is that common cloud computing practices constrain a web API's underlying systems. Web APIs hosted on the cloud are implicitly releasing data about their system design. In this section, we describe paradigms widely accepted as best practices in cloud computing. Also, they constrain underlying system structures and resource usage enough to extract usable workload profiles.

Tiered Design: The systems that power web APIs must support concurrent requests. They use distributed and tiered systems where each request traverses a few nodes across multiple tiers (a tier is a software platform, e.g., Apache Httpd) and tiers spread across many nodes. Client-side observed response times are mixtures of per-tier delays. Multiple tiers confound response times since relatively slow tiers can be masked by other tiers, hiding the effect of the slow tier on response time [30]. In the cloud, tiers are divided by Linux processes, containers or virtual machines. Each tier's resource usage can be tracked independently.

Auto Scaling: APIs hosted the cloud can add and remove resources on demand. Such auto scaling reduces variability in queuing delay, i.e., the portion of response time spent waiting for access to resources at each tier. Since per-tier delays and their variance can be reduced by auto scaling [17, 18, 20], it could further reduce the visibility of a poorly implemented component to outsiders. Meanwhile, the stability of per-tier delays caused by auto scaling [17] gives users opportunity to collect and analyze more consistent response times with less consideration about the changes of the per-tier delay distributions.

Make the Common Case Fast: To keep response times low, API providers trim request data paths. In the common case, a request touches as few nodes and resources as possible with each tier performing only operations that affect the request's output. Well implemented APIs make the common case as fast as possible and uncommon cases rare. This design philosophy skews processing times. Imbalanced processing time distributions are inherently non-Gaussian.

Alternative Research Directions: Our research treats data sharing across administrative domains as a fundamental challenge. An alternative approach would enable data sharing by building trusted data collection and dissemination platforms. Developers would prefer APIs hosted on such platforms and robust APIs would be used most often. The challenge would be enticing

API providers to use the platform. Another approach would have API providers support service level agreements with punitive consequences for poor performance. We believe that approaches based on inferring unknown workload profiles, enabling data sharing or enriching SLAs all provide solid research directions.

3 Blind Source Seperation

Blind Source Separation (BSS) describes statistical methods that 1) accept signals produced by mixing source signals as input, 2) place few constraints on the way source signals are mixed, and 3) output the source signals. Real world applications of BSS include: magnetic resonance imaging, electrocardiography, telecommunications, and famously speech source separation. The most widely used BSS methods include: independent component analysis (ICA), principle component analysis (PCA), and singular value decomposition (SVD). All of which are commonly taught in graduate courses [12,23].

Workload profiling for web APIs aligns well with BSS. First, second- and third-order statistics can enrich first-order response times collected from the client. Response times alone can mislead developers. Second, there are a wide range of BSS methods distinguished by their constraints on source signals and mixing methods. The research challenge is to figure out which BSS methods yield usable workload profiles (not devising new statistical methods). The systems community can best answer this question. Finally, BSS methods can reach a wide range of web developers that may have encountered BSS during graduate studies or online courses. Given the cost savings from avoiding web APIs that perform poorly, developers will likely find it worthwhile to install BSS libraries which have been written in many languages from MATLAB to Java to C.

3.1 Web API profiling using ICA

In early work, we have used ICA to profile per-tier delays. The input to ICA is a time signal, usually denoted as $\mathbf{x}$. This input signal $\mathbf{x}$ is a linear transformation of all sources, i.e., $\mathbf{x} = A\mathbf{s}$, where the mixing matrix A does not change over time. The output of ICA is $\mathbf{s}$, i.e., (normalized) source signals. The number of input signals should be greater than or equal to the number of source signals. The key theory behind ICA is central limit theorem, which states that the input signal created by summing two independent source signals is closer to the Gaussian distribution than both source signals, provided source signals are not Gaussian.

Let's return to the constraints imposed by cloud computing practices discussed in Section 2. Making the common case fast leads to imbalanced, non-Gaussian processing times. Auto scaling ensures that processing times are a key factor influencing response times— not queuing delays [17]. Finally, tiered design suggests that a request's response time is summed ($\mathbf{x} = A\mathbf{s}$) across tiers. Below, we describe exactly how we used ICA to profile APIs from observed response times.

System Model: Users interact with Web APIs by sending HTTP requests and receiving responses. The system underlying the API uses tiered design and auto scaling. Response times observed at the client side (i.e., by developers) are the sum of delays caused by repeated processing at each tier *inside* of the API's backend systems. Formally, the delay of tier i could be considered as a random variable s_i.

Recall, ICA requires more input signals than source signals. To acquire multiple signals at each point in time, we concurrently probe the API with multiple *request types*. Requests are of the same type if their access frequencies are the same across each tier. This means for request type j, the response time is $x_j = \mathbf{a_j}^T\mathbf{s}$, where $\mathbf{s} = (s_1, \ldots, s_N)^T$ is a vector consisting of random variables, N is number of tiers and $\mathbf{a_j}$ is a constant vector specific to request type j. Intuitively, the weight vector $\mathbf{a_j}$ reflects the frequency with which each tier is called during request execution and s_i reflects the tier's average delay [29]. Suppose there are M types of requests. Each observation is a response time of certain type of request. Then the problem is: If we can collect arbitrary number of observations, whether it is possible to recover per-tier delay (s_i) of a system.

ICA requires non-Gaussian and independently distributed source signals. It also depends on simultaneous observation from multiple mixtures. Cloud-based services meet these requirements. In the common case, OS and background jobs do not interfere with request executions but, when they do, they cause fat, non-Gaussian tails at each tier [28]. Also, per-tier delays are largely independent because different tiers usually run on separate virtual machines and are scaled independently. Finally, in most systems, average per-tier delays change on the order of minutes, not milliseconds. This fact helps us to make simultaneous observations by issuing several requests with different types concurrently. These concurrent requests triggers roughly the same per-tier delays, which makes the observation a linear transformation of the per-tier delays. Using notations defined above, an observation is a vector of response times $\mathbf{x} = A\mathbf{s}$, where $A = (\mathbf{a_1}, \ldots, \mathbf{a_m})^T$. By collecting a series of observations, we can apply ICA on these response times and recover the per-tier delay distributions.

Limitations: ICA recovers the shape of the source signal but not the energy. To predict response times, we would need to shift and scale the output. More generally,

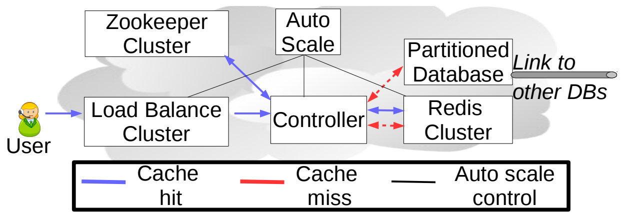
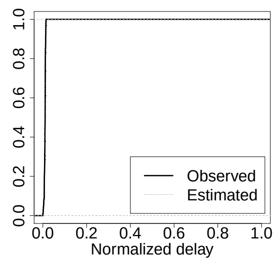
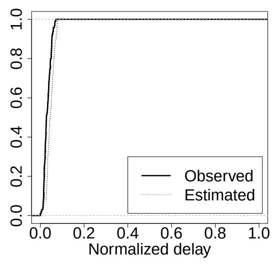


Figure 1: Components and datapath for a scalable email service hosted in the cloud.



(a) Cache tier delays.



(b) Database tier delays.

Figure 2: CDFs of observed and estimated delays of each tier. No other workloads to the system.

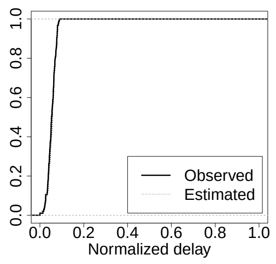
BSS methods provide less data hindering the workload profiles. However, as we will soon show, normalized distributions suffice to identify some warning signs. Also, ICA does not match recovered distributions to tiers. We currently do this manually. With a library of expected per-tier distributions, we could use information gain or EMD to automate this process.

4 Preliminary Results

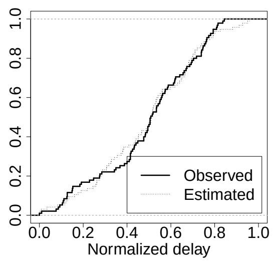
To validate our approach, we build a distributed key-value storage service consisting of 2 tiers — cache tier, which uses Redis [4] as an in-memory cache; and database tier, which uses MySQL. The cache tier consists of multiple instances, and is automatically scaled based on the workload. Users access the system through a web API. The API can handle get and put requests. Inside the system, we monitor per-tier delays and use them as the ground truth to compare with the estimations. Figure 1 shows the architecture of our system. Each replicated component runs in its own virtual machine. Each virtual machine runs on 112 core cluster. Each core has at least 2.0 GHz, 3MB L2 cache, 2GB of DRAM memory, and 100GB of secondary storage. When probing the API, we manually set the cache miss rate.

4.1 Per-tier delay distributions recovery

As we mentioned in previous sections, we would like to recover per-tier delay distributions in an API's backend. We first test our technique against a vanilla system with no other users. We probe the API by sending requests every second. The API only serves our probing requests. We collects response times after 100 seconds and run ICA to estimate per-tier delay distributions only based on these response times. We also collected actual delay at each tier by looking at Redis and MySQL logs.



(a) Cache tier delays.



(b) Database tier delays.

Figure 3: CDFs of observed and estimated delays of each tier. The system is serving one day workload from the WorldCup trace.

Figure 2 shows the actual and estimated cumulative distribution functions (CDFs) of normalized per-tier delays. Since there is no other users using the API, the variation of delays in each tier is small. We can see that our approach could precisely estimate the distributions with error less than 3%. Next, we ran with background workload in addition to the API probes. We used HttpPerf to simulate 600 users performing 50% reads and writes. The background workload increased the tail but our per-tier delays were still within 5%.

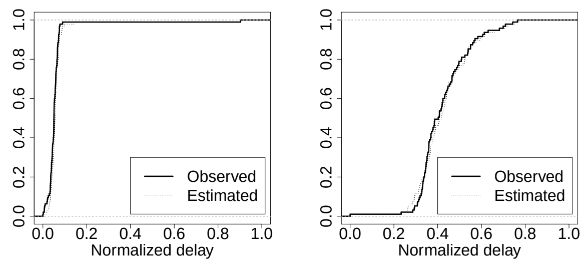
4.2 Impact of Real Workloads

We repeat the experiment from Section 4.1 but run some real workload using the API to examine the performance of our approach. We simulate one day workload from WorldCup98 [1](Day 78), and probe the API 100 times when it is serving the workload. We can see from Figure 3 that both tiers exhibit even heavier tails because of the variation of request rates over the day. Even though the variation of per-tier delays becomes larger, the recovered distribution can still follow the actual distribution precisely with errors within 5%.

4.3 Choosing between competing APIs

Our ICA-based approach could help users get more information about the backend system implementation of the target API. When users try to choose an API from two competing ones, our approach could provide some insight in the systems which could be used as a guide line for users.

We replace Redis instances with a ZooKeeper [2] cluster in the cache tier in our experiment system to create a poorly implemented key-value storage service.



(a) Redis setup. (b) ZooKeeper setup.

Figure 4: CDFs of observed and estimated delays of the cache tiers.

ZooKeeper is a well-known centralized system supporting high-availability through redundant replicas. It keeps strong consistency using a Paxos-like protocol, which fully replicates content on all nodes. It is perfect to store small important configuration data but would be a bad choice for cache.

We run two competing key-value storage services. One uses Redis as cache and another uses Zookeeper. We adjust the request rates for both services so that their mean and median response times are similar. For both services, we simulate concurrent user requests with a fixed rate. For the Redis setup, we issue 100 concurrent requests every 500ms; while for the ZooKeeper setup, we issue 50 concurrent requests every 500ms. Although ZooKeeper has an unstable and slow service times, the lower request rate and the tier-2 delays make the ZooKeeper setup look almost the same as, or even better than the Redis setup. The mean and median response times of ZooKeeper setup are 3.8ms and 2.7ms; while the mean and median response times of Redis setup are 4.5ms and 3.1ms. ZooKeeper setup performs even better than Redis setup by comparing these metrics. Fortunately, our profiles revealed the poorly implemented service.

Figure 4 shows the observed and estimated cumulative distribution functions of the cache tier for two services. Even though the database tier and the difference of request rate masked the unstable component in the ZooKeeper setup, our technique can still accurately recover per-tier delay distributions. It is clear on the graph that the ZooKeeper setup's cache tier has a fatter tail; while the Redis setup's cache tier is relatively stable with little variation. We further increased the request rate for the ZooKeeper setup to the same rate as the Redis setups's, the mean and median response times quickly increased to 8.6ms and 7.5ms.

5 Related Work

Workload profiling approaches differ according to their outputs, inputs, and targeted systems. Our research uses response time data collected as an outsider by probing web APIs. Prior work has been more invasive. PowerTracer [22] and Power Containers [26, 27] mapped local events to request contexts even when request executions moved across nodes. These low-level event traces were combined to produce diverse workload profiles ranging from per-node system call counts to per-tier energy efficiency. Events were collected using modified kernels. These approaches targeted services within a single administrative domain where trusted code bases could be changed. Magpie [10] and XTrace [15] also used modified kernels to collect events but events were not automatically linked to request contexts. The system manager manually linked events. As a result, these approaches could span multiple domains—if events can be linked. Instead of changing source code, Mantis [21] and ConfAid [9] modified application binaries to collect events, e.g., loop, branch, and method call counts.

Many domains prohibit code changes. For these systems, recent approaches have used aggregate CPU, network, and disk usage statistics collected by vanilla monitoring tools. Offline approaches measure these statistics under specific request arrival patterns [34], whereas online approaches passively collect statistics as traffic arrives [16, 29]. Generally, profiles produced by offline approaches extrapolate to a wide range of request patterns but online approaches are supported in more domains. Hybrid approaches balance coverage with practicality [20, 31]. In cloud services, some statistics are amenable to automatic resource provisioning. Specifically, resource pressure [24] and queue length [18] work well with threshold based auto scaling.

6 Discussion and Conclusion

Web APIs are surging because the RPC paradigm aligns well with cloud computing trends: First, large datasets stay in one place and second, growing network bandwidth leads to increased throughput. While traditional services underlie web APIs today, BSS methods will profile data parallel services in the future. Our ICA-based approach can profile map reduce, capturing worst-case service times for the map and reduce phases. However, iterative data parallel platforms, like Spark [5], present challenges. Emerging workloads that exhibit highly diverse behaviors within requests types because of time-varying demands also present challenges [3, 11, 19].

BSS has a strong track record in practice. A key next step for our work is to apply BSS methods to real web APIs. The challenge is to uncover more warning signs, preferably non-parametric signs that can be indentified directly from profiles. Another challenge in working with real web APIs is probing overhead. Web APIs enforce strict rules about the frequency and types of API access, e.g., 2 accesses per second per user [7].

Our current research closed the loop between cloud applications and web APIs, By providing more meaningful profiles of APIs, cloud applications will be able to control their internal system in a more effective way. It is worthwhile to explore how to build a robust cloud applications by using profiles of APIs recovered by BSS. For example, a cloud application may dynamically dispatch requests to different APIs based on their profiles to avoid busy hours of an API.

In conclusion, we proposed research on profiling third party web APIs using BSS techniques. Using data collected outside of an API provider's system, we are able to "look in" at detailed workload profiles. In early results, we used ICA to recover accurate profiles. We also showed that our workload profiles were helpful, providing insight into design of tested services.

References

[1] 1998 World Cup Workload. `http://ita.ee.lbl.gov/html/contrib/WorldCup.html`.

[2] Apache zookeeper. `http://zookeeper.apache.org/`.

[3] Carbon-aware energy capacity planning for datacenters.

[4] Redis. `http://redis.io/`.

[5] Spark: Cluster computing with working sets.

[6] Stackoverflow question 10986702:is zookeeper appropriate for object caching? `http://stackoverflow.com`, 2012.

[7] Programmable web: Mashups, apis and the web as a platform. `http://www.programmableweb.com`, 2013.

[8] Stackoverflow question 1479442:real world use of zookeeper. `http://stackoverflow.com`, 2013.

[9] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *USENIX OSDI*, 2010.

[10] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *OSDI*, 2004.

[11] M. L. C. Stewart and K. Shen. Empirical examination of a collaborative web application. In *IEEE International Symposium on Workload Charactization*, Sept. 2008.

[12] P. Comon and C. Jutten. *Handbook of Blind Source Separation: Independent Component Analysis and Applications*. Academic Press, 1st edition, 2010.

[13] T. Everts. An 11-step program to bulletproof your site against third-party failure. `http://blog.radware.com/`, 2013.

[14] T. Everts. Case study: Understanding the impact of slow load times on shopping cart abandonment. `http://blog.radware.com/`, 2013.

[15] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, 2007.

[16] A. Gandhi, Y. Chen, D. Gmach, M. Arlitt, and M. Marwah. Minimizing data center sla violations and power consumption via hybrid resource provisioning. In *IGCC*, 2011.

[17] A. Gandhi, S. Doroudi, M. Harchol-Balter, and A. Scheller-Wolf. Exact analysis of the m/m/k/setup class of markov chains via recursive renewal reward. In *ACM SIGMETRICS*, 2013.

[18] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems (TOCS)*, 30(4):14, 2012.

[19] I. n. Goiri, W. Katsak, K. Le, T. D. Nguyen, and R. Bianchini. Parasol and greenswitch: Managing datacenters powered by renewable energy. Mar. 2013.

[20] X. Gu and H. Wang. Online anomaly prediction for robust cluster systems. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 1000–1011. IEEE, 2009.

[21] Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B.-G. Chun, L. Huang, P. Maniatis, M. Naik, and Y. Paek. Mantis: automatic performance prediction for smartphone applications. In *USENIX Annual Technical Conf.*, 2013.

[22] G. Lu, J. Zhan, H. Wang, L. Yuan, and C. Weng. Powertracer: Tracing requests in multi-tier services to diagnose energy inefficiency. In *Proceedings of the 9th international conference on Autonomic computing*, pages 97–102. ACM, 2012.

[23] D. J. C. MacKay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, New York, NY, USA, 2002.

[24] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. Agile: elastic distributed resource scaling for infrastructure-as-a-service. In *Proc. of the USENIX International Conference on Automated Computing (ICAC13). San Jose, CA*, 2013.

[25] A. Peters. Why loading third party scripts async is not good enough. `http://www.aaronpeters.nl/`, 2011.

[26] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, and Z. Chen. Power containers: An os facility for fine-grained power and energy management on multicore servers. In *ACM ASPLOS*, 2012.

[27] K. Shen, M. Zhong, S. Dwarkadas, C. Li, C. Stewart, and X. Zhang. Hardware counter driven on-the-fly request signatures. In *ACM ASPLOS*, Mar. 2008.

[28] C. Stewart, A. Chakrabarti, and R. Griffith. Zoolander: Efficiently meeting very strict, low-latency slos. In *Int'l Conference on Autonomic Computing*, 2013.

[29] C. Stewart, T. Kelly, and A. Zhang. Exploiting nonstationarity for performance prediction. In *ACM European Systems Conference*, Mar. 2007.

[30] C. Stewart, K. Shen, A. Iyengar, and J. Yin. Entomomodel: Understanding and avoiding performance anomaly manifestations. In *IEEE MASCOTS*, 2010.

[31] E. Thereska and G. R. Ganger. Ironmodel: Robust performance models in the wild. *ACM SIGMETRICS Performance Evaluation Review*, 36(1):253–264, 2008.

[32] TripAdvisor Inc. Tripadvisor reports fourth quarter and full year 2013 financial results, Feb. 2014.

[33] H. Tsukayama. Facebook outage takes down gawker, mashable, cnn and post with it. `http://21stcenturywire.com/`, 2013.

[34] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo. Automated profiling and resource management of pig programs for meeting service level objectives. In *Proceedings of the 9th international conference on Autonomic computing*, pages 53–62. ACM, 2012.