To Auto Scale or not to Auto Scale

Nathan D. Mickulicz
YinzCam, Inc. / Carnegie Mellon University

Priya Narasimhan
YinzCam, Inc. / Carnegie Mellon University

Rajeev Gandhi
YinzCam, Inc. / Carnegie Mellon University

Abstract

YinzCam is a cloud-hosted service that provides sports fans with real-time scores, news, photos, statistics, live radio, streaming video, etc., on their mobile devices. YinzCam's infrastructure is currently hosted on Amazon Web Services (AWS) and supports over 7 million downloads of the official mobile apps of 40+ professional sports teams and venues. YinzCam's workload is necessarily multi-modal (e.g., pre-game, in-game, post-game, game-day, non-gameday, in-season, off-season) and exhibits large traffic spikes due to extensive usage by sports fans during the actual hours of a game, with normal game-time traffic being twenty-fold of that on non-game days.

We discuss the system's performance in the three phases of its evolution: (i) when we initially deployed the YinzCam infrastructure and our users experienced unpredictable latencies and a large number of errors, (ii) when we enabled AWS' Auto Scaling capability to reduce the latency and the number of errors, and (iii) when we analyzed the YinzCam architecture and discovered opportunities for architectural optimization that allowed us to provide predictable performance with lower latency, a lower number of errors, and at lower cost, when compared with enabling Auto Scaling.

1 Introduction

Sports fans often have a thirst for real-time information, particularly game-day statistics, in their hands. The associated content (e.g., the game clock, the time at which a goal occurs in a game along with the players involved) is often created by official sources such as sports teams, leagues, stadiums and broadcast networks.

From checking real-time scores to watching the game preview and post-game reports, sports fans are using their mobile devices extensively [11] and in growing numbers in order to access online content and to keep up to date on their favorite teams, according to a 2012 report from Burst Media [10]. Among the surveyed sports fans, 45.7% said that they used smartphones (with 31.6% using tablets) to access online sports-content at least occasionally, while 23.8% said that they used smartphones (with 17.1% using tablets) to watch sporting events live. This trend prevails despite the presence of television–in fact, fans continued to use their mobile devices to check online content as a second-screen or third-screen viewing-experience even while watching television.

YinzCam started as a Carnegie Mellon research project in 2008, with its initial focus being on providing in-venue replays and in-venue live streaming camera angles to hockey fans inside a professional ice-hockey team's arena [5]. The original concept consisted of a mobile app that fans could use on their smartphones, exclusively over the in-arena Wi-Fi network in order to receive the unique in-arena video content. While YinzCam started with an in-arena-only mobile experience, once the research project moved to commercialization, the resulting company, YinzCam, Inc. [15], decided to expand its focus beyond the in-venue (small) market to include the out-of-venue (large) market.

YinzCam is currently a cloud-hosted service that provides sports fans with real-time scores, news, photos, statistics, live radio, streaming video, etc., on their mobile devices anytime, anywhere, along with replays from different camera angles inside sporting venues. YinzCam's infrastructure is currently hosted on Amazon Web Services (AWS) and supports over 7 million downloads of the official mobile apps of 40+ professional sports teams and venues within the United States.

Given the real-time nature of events during a game and the range of possible alternate competing sources of online information that are available to fans, it is critical for YinzCam's mobile apps to remain attractive to fans by exhibiting low user-perceived latency, a minimal number of user errors (visible to user in the form of timeouts occuring during the process of loading a page), and

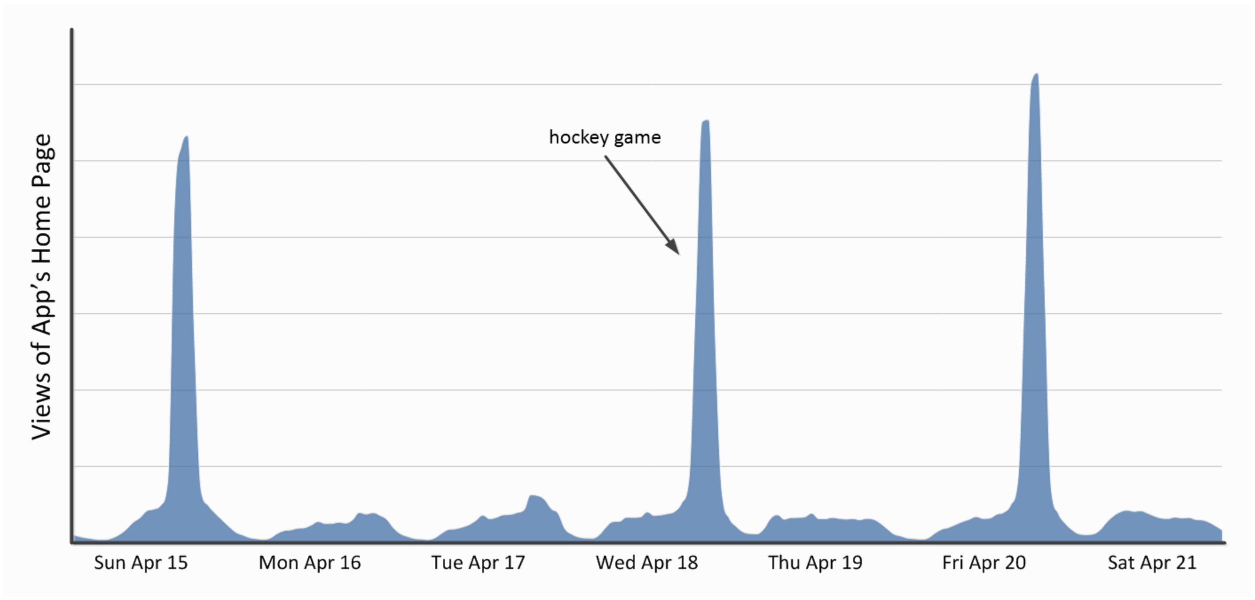

Figure 1: The trend of a week-long workload for a hockey-team's mobile app, illustrating modality and spikiness. The workload exhibits the spikes due to game-day traffic during the three games in the week of April 15, 2012.

real-time information updates, regardless of the load of the system. Ensuring *a responsive user experience* is the overarching goal for our system. Our infrastructure needs to be able to handle our *unique spiky workload*, with game-day traffic often exhibiting a twenty-fold increase over normal, non-game-day traffic, and with unpredictable game-time events (e.g., a hat-trick from a high-profile player) resulting in even larger traffic spikes.

Contributions. This paper describes the evolution of YinzCam's production architecture and distributed infrastructure, from its beginnings three years ago, when it was used to support thousands of concurrent users, to today's system that supports millions of concurrent users on any game day. We discuss candidly the weaknesses of our original system architecture, our rationale for enabling Amazon Web Services' Auto Scaling capability [2] to cope with our observed workloads, and finally, the application-specific optimizations that we ultimately used to provide the best possible scalability at the best possible price point. Concretely, our contributions in this paper are:

- An AWS Auto Scaling policy that can cope with such unpredictably spiky workloads, without compromising the goals of a responsive user experience;
- Leveraging application-specific opportunities for optimization that can cope with these workloads at lower cost (compared to the Auto Scaling-alone approach), while continuing to meet the user-experience goals;
- Lessons learned on how Auto Scaling can often mask architectural inefficiencies, and perform well (in fact, too well), but at higher operational costs.

The rest of this paper is organized as follows. Section 2 explores the unique properties of our workload, including its modality and spikiness. Sections 3, 4, and 5 describe our Baseline, Auto Scaling, and Optimized system configurations, respectively. Section 6 presents a performance comparison of our three system-configurations. Section 7 describes related work. Finally, we conclude in section 8.

2 Motivation

In this section, we explore the properties our workload in depth, describing the natural phenomena that cause its modality and spikiness. First, let's consider the modal nature of our workloads. Each workload begins in the non-game mode, where the usage is nearly constant and follows a predictable day-night cycle. In Figure 1, this can be seen on April 16, 17, 19, and 20. On game days, such as April 15, 18, and 20, our workload changes considerably. In the hours prior to the game event, there is a slow build-up in request throughput, which we refer to as pre-game mode. As the game-start time nears, we typically see a load spike where the request rate increases rapidly as shown in Figure 2. The system now enters in-game mode, where the request rate fluctuates rapidly throughout the game. In the case of hockey games, these fluctuations define sub-modes where usage spikes during

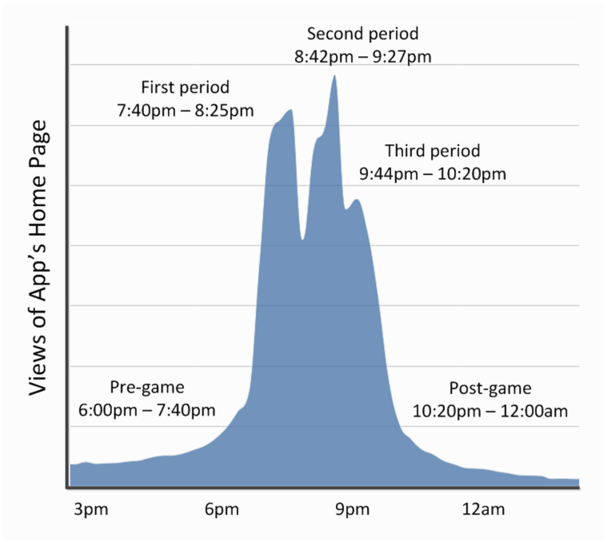

Figure 2: The trend of a game-time workload for a hockey-team's mobile app, illustrating modality and spikiness. The workload shown is for a hockey game in April 2012.

the first, second, and third hockey periods and drops during the 20-minute intermissions between periods. The load drops off quickly following the end of the game during what we call the post-game mode. Following the post-game decline, the system re-enters non-game mode and the process repeats.

In addition to modality, our workload exhibits what we call spikiness. We define a spiky workload to be one where the request rate more than doubles within a 15-minute period. The workload shown in Figure 2 exhibits spikiness at 7:30pm, towards the end of the pre-game phase. Between 7:30 and 7:45, the request rate of our workload more than doubled, and it nearly doubled again by 8:00pm. The request rate reached its maximum between 9:15 and 9:30 in the middle of the second period, when several significant scoring-events occurred. In addition to the rapid increases, our workload also shows equally-rapid decreases as the system enters the post-game mode, with the workload nearly halving in request rate between 10:15pm and 10:30pm.

To understand why our workload has these properties, we have to consider our users' demand for app content. Both figures 1 and 2 show the number of home-screen views in one of our hockey-team apps. Typically, this page shows news and media items as well as the box score of the previous game played. However, during games, the home screen shows real-time data such as live team-statistics and the game clock as well as a portion of team's Twitter feed. This effectively provides fans with a way to follow the game without being tuned in via radio or television. It also provides a second screen that can augment an existing radio or television broadcast with live statistics and Twitter updates. Our fans have found these features to be tremendously useful and compelling, and our workload shows that these features are used heavily during periods when the game is in play. At other times, the app provides a wealth of information about the team, players, statistics, and the latest news. These functions, which lack a real-time aspect, have their usage spread evenly throughout the day.

3 Configuration 1: Baseline

Figure 3 shows the architecture of our system when we first began publishing our mobile apps in 2010. The system is composed of two subsystems using a three-tier architecture, one consuming new content and the other pushing views of this content to our mobile apps. The two subsystems share a common relational database, shown in the middle of the diagram.

The content-aggregation tier is responsible for keeping the relational database up-to-date with the latest app-content, such as sports statistics, game events, news and media items, and so on. It runs across multiple EC2 instances, periodically polling information sources and identifying changes to the data. The content-aggregation tier then transforms these changes into database update queries, which it executes to bring the relational database up-to-date. Since these updates are small and infrequent, the load on the EC2 instances and database is negligible. The infrequency of updates also allows us to use aggressive query-caching on our app servers, preventing the database from becoming a bottleneck.

Our apps periodically retrieve new information from the content-storage tier in the form of XML documents. The app then displays information from the XML document to the user in various ways. The content-delivery tier is responsible for composing views of the relational data as XML documents that are ready for consumption by our apps. In response to a request for XML data, the content-delivery tier executes one or more database queries and synthesizes the results into an XML document on the fly. This task is both I/O-intensive and CPU-intensive. Fortunately, scaling up this component is simply a matter of adding additional EC2 instances behind a load balancer.

However, despite being conceptually simple, we encountered multiple problems scaling up our system as the size of our user-base increased. Initially, scaling up the content-delivery tier was an entirely-manual task. We did this before every sporting event, and we could only guess how many additional servers we would need to handle the CPU load for the game. Since the spikes we see are of varying magnitude, we would often provision too

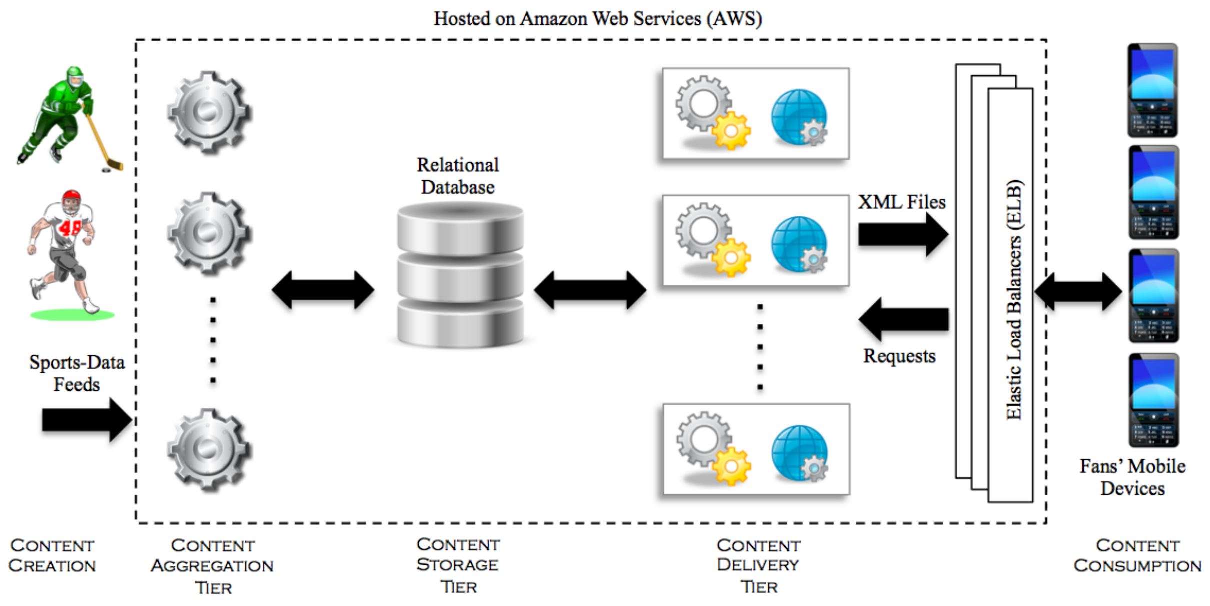

Figure 3: Baseline Configuration

much or too little, either wasting money on resources we didn't need or frustrating our users with high latency and errors. In the results section, we describe the high response latency of our baseline architecture when under-provisioned for a game workload.

4 Configuration 2: With Auto Scaling

We first adopted EC2 Auto Scaling [2] to deal with the CPU-load-management problem. Auto Scaling is a technique that allows system administrators to automatically scale a fleet of EC2 resources up or down to meet current workload-demands. We defined an Auto Scaling policy that allowed our system to adapt to the spikes in our workload.

We follow an aggressive scale-up policy in our system to cope with our spiky workloads. Spikes in our workload happen very quickly, and if we do not scale our resources quickly, many of our users experience high latency and errors while additional resources are being added. We use current CPU-usage in our content-delivery tier to determine when to add additional servers. We have set the CPU-usage threshold for scale up to a low value of 30% average over 1 minute, in an attempt to catch spikes early. We also scale up by doubling the size of the fleet to make sure that we have enough instances available to handle double the workload volume.

On the other hand, we defined a scale-down policy that was slow and cautious. There are periods of lower-volume usage (such as between periods in a hockey game) where we did not want to scale down prematurely. Furthermore, scaling down too rapidly could remove too many resources from the pool, forcing the system to immediately scale up after the next CPU-utilization check. This could cause the system to flap back and forth between two fleet sizes, wasting EC2 resources.

We downscale our fleet of servers by removing one server at a time and making sure that the CPU usage in our content-delivery tier is stable for a longer period of time before we do another round of downscaling. Thus, our Auto Scaling policy can be described as Multiplicative-Increase-Linear-Decrease (MILD). This policy was inspired by the Additive-Increase-Multiplicative-Decrease (AIMD) policy for congestion control used in TCP. As with congestion control, our scaling policy attempts to prevent load-collapse by cautiously modifying parameters of the system until the workload matches the system's capacity. The major difference between MILD and TCP's AIMD is that TCP gradually increases the workload until the network is at capacity. Since we do not have the ability to rate-limit our workload, we take the opposite approach of gradually reducing our system's maximum capacity until this capacity matches the workload.

As described in section 6, Auto Scaling does solve the high-latency problem caused by high CPU load for the baseline configuration. Adding additional instances effectively adds more CPU resources, and when placed behind a round-robin load-balancer such as Amazon's ELB [3], each instance gets an equal share of the workload. Furthermore, Auto Scaling only increases the size of the fleet when the workload demands it, so we don't have to over-provision for each game. With Auto Scaling, we were able to scale up our system to get acceptable latencies in the face of our spiky workload even though our system had a sub-optimal design with glaring inefficiencies.

Unfortunately, masking inefficiencies with Auto Scaling does not come without a cost. We had to pay for up to 15 additional instances per team during each game, which adds up to a considerable increase in operation costs over an entire season of games (for our hockey apps, 82 regular-season games per team). At this point, we wondered if we could lower our operations costs by removing the inefficiencies in our architecture, thus lowering our CPU requirements and the number of instances required to handle our workload during games. The subsequent analysis of the inefficiencies in our system led us to the optimized architecture we describe next.

5 Configuration 3: Optimized

Our optimized architecture is the result of studying our baseline architecture, identifying inefficiencies, and modifying the architecture to correct them. After studying our system, we identified two major problems with our architecture. The first problem was in our request handling, where we realized that every request required the server to generate a new XML document from data stored in the database. Often, the system generated multiple identical XML documents within a short period of time. The second was in our database layer, where we noticed that certain queries were being executed multiple times within a few seconds, each returning the same result. These observations led us to add two caching-layers to our architecture, which we describe below.

In response to the observation that every request required XML generation, we added a caching layer in front of the content-delivery tier. This layer receives requests from clients and serves pre-generated XML content if the cache time on the content has not expired; otherwise, it regenerates the XML content using the content-delivery tier and stores the content to serve subsequent requests. This dramatically reduces instance CPU utilization, since new pages are only generated when cached content expires (instead of on every request).

We implemented our caching layer using the output-caching feature of the Microsoft IIS web server, which required very little additional code or configuration on top of our existing IIS-based content-delivery tier. We assigned groups of XML documents a cache-expiration time based on how much staleness the content could tolerate. For example, XML documents describing a news article are unlikely to change after publication and so have a cache-expiration time of a day or more, while documents describing the latest game events change frequently and have cache-expiration times on the order of seconds.

Our second optimization was another caching-layer between the content-delivery tier and the content-storage tier, in response to the observation that the content-delivery tier was executing multiple queries for the same data in rapid succession. While our relational database has an internal query cache, some of our queries generate tens of megabytes of data. When several of these queries are run in parallel, this volume of data can quickly saturate the network link between our EC2 instances and the database. To remedy this, we added a cache of query results in the content-delivery tier.

This cache is implemented in a similar fashion to the output cache. When a server in the content-delivery tier needs to execute a database query, it first looks up the query in a table of query results. If the query is found in the table and the result has not expired, the result from the table is used to generate the XML. Otherwise, the generator runs the database query directly and updates the table with the results. Like individual XML-documents, each query is assigned a cache time based on the staleness-tolerance of the data.

6 Evaluation

We evaluated our three system architectures using a HTTP load generator and a trace of a production workload. We ran the three-hour trace and recorded the average latency seen over consecutive 60-second intervals. We then compared the results of these tests to determine the effectiveness of each approach in reducing the response latency seen by our users.

We built our testbed entirely on EC2, utilizing EC2's internal network for communication between simulated clients and the view generation service. Using the logs collected during the April 18, we simulated clients by using a program that replays the log file exactly as it was recorded using the timestamps associated with each request. The EC2 instance used to simulate clients was of type m1.xlarge.

We ran our tests against a copy of our production system. This copy was set up in an identical configuration as our production system, except it was not serving production traffic. To make this copy, we duplicated both the load balancer configuration and the Auto Scaling configuration. We used the same operating system image as the production service with the same EC2 instance type (c1.medium). We also created an isolated read replica of our database used only by the EC2 instances involved in the test.

Our test trace was a three-hour log of traffic served by our production system during a hockey game in April

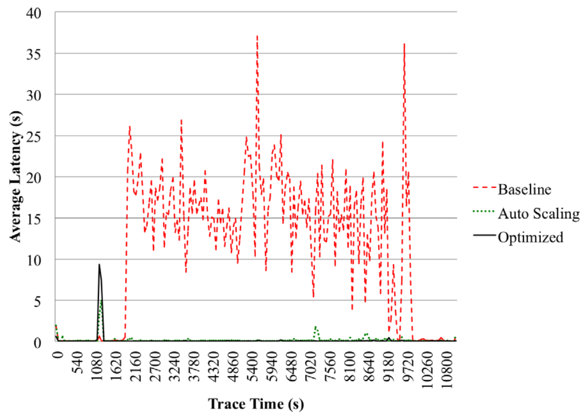

Figure 4: The average latency of our three system configurations throughout a 3-hour production workload trace. The workload was recorded during a hockey-game in April 2012.

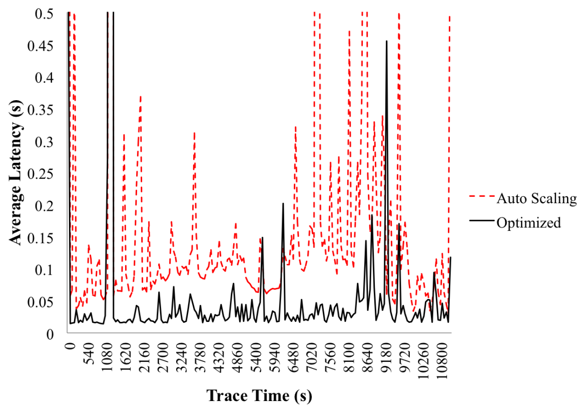

Figure 5: The same results shown figure 4, zoomed in to show the differences between the Auto Scaling and Optimized configurations.

2012. This trace captures all of the spikes in our workload. For each configuration of our system, we replayed this trace in real time using our client simulator. Each test was run in isolation on our test infrastructure, and all resources were rebooted between runs.

We tested three configurations of our system: Baseline, Auto Scaling, and with Optimizations. The Baseline configuration is the system described in section 3, running on a single instance through the entire trace. The Auto Scaling configuration is the Baseline configuration running with the Auto Scaling policies described in section 4, configured with one initial instance but a maximum of sixteen. Finally, the Optimized configuration is the system described in section 5, which extends the Baseline configuration with additional caching to improve performance. Like the Baseline configuration, the Optimized configuration is restricted to a single instance.

Figure 4 shows a comparison of average latency over 60-second windows throughout the entire 3-hour trace across all three configurations. The baseline configuration performs much worse than either the Auto Scaling or Optimized, due to the restricted amount of CPU time available on a single instance. Both Auto Scaling and Optimized perform far better through their respective mechanisms for coping with heavy load - Auto Scaling simply adds more resources to the pool, while Optimized makes more efficient use of CPU resources.

Figure 5 shows the same trace comparison as in figure 4, except zoomed-in to show the fine differences between the Auto Scaling and Optimized cases. This comparison shows that Optimized has both lower average-latency and lower jitter than the Auto Scaling case throughout the entire trace. Through careful optimization, we were able to outperform Auto Scaling using just a fraction of the eight instances that Auto Scaling required to achieve comparable performance.

7 Related Work

A substantial amount of research has been done in the area of using Auto Scaling to cope with workloads that have a high peak-to-average ratio. A number of related works use predictive models to forecast the workload and then use Auto Scaling to dynamically adjust the resource usage to match the predicted workload. The main difference between some of these proposals is the way predictive models are built and used. For instance, [12], [1], [8] use control-theoretic models to predict workloads while [7], [13] use autoregressive moving average (ARMA) filters for prediction. Mao et. al. [9] use Auto Scaling to scale up or down cloud infrastructure to ensure that all job deadlines are met under a limited budget. There is also a significant amount of research done in the area of workload modeling and prediction, even though the proposed systems do not explicitly mention using Auto Scaling to cope with varying work load. Gmach et. al. [6] use resource pools that are shared by multiple applications and propose a process for automating the efficient use of such resource pools. Shivam et. al.[14] use an active-learning approach to analyze the performance histories of hosted applications to build predictive models for future use of the applications and use the predicted values for future resource assignment. Chandra et. al. [4] propose to capture the transient behavior of web applications workload by modeling the server resource.

While we can certainly benefit from some of the workload-prediction-related work, as we demonstrate in Section 2, our workloads can be spiky and are often hard to predict in advance (e.g. predicting whether a player will score a hat-trick). We believe that contextual prediction e.g., prediction based on the analysis of statistics feeds to determine what might be going on in the game (a player close to scoring a hat-trick), or prediction based on the analysis of news feeds to learn about important events (like return of a favorite player), might be more appropriate and useful in our context.

8 Conclusions and Reflections

Auto Scaling is often provided by IaaS providers as a technique by which applications can scale up/down resources to meet the current demand. Our results show that Auto Scaling works well, in-fact so well that we were able to take a system with obvious architectural flaws and make it perform nearly as well as a fully-optimized version. However, hiding these inefficiencies comes with the price of additional infrastructure. In our case, our inefficient Baseline-configuration required up to eight times the resources of our efficient Optimized-configuration to achieve comparable performance.

On the other hand, the optimizations that we made were fairly simple and straightforward to identify and implement. This may not be the case with all systems, and optimizing effectively often requires considerable skill and effort. However, if it can be done, the payoff can be much greater than simply using Auto Scaling with an inefficient system. In our case, our thoughtful optimizations required greater insight and more development time, but paid off through lower costs, lower latency, and lower jitter than either of the other configurations.

Acknowledgments

The authors would also like to thank Justin N. Beaver and Michael P. Kasick for their contributions to YinzCam's cloud-hosted infrastructure and services.

References

[1] ABDELWAHED, S., BAI, J., SU, R., AND KANDASAMY, N. On the application of predictive control techniques for adaptive performance management of computing systems. *IEEE Transactions on Network and Service Management 6* (Dec. 2009), 212–225.

[2] AMAZON WEB SERVICES. Auto Scaling. `"http://aws.amazon.com/autoscaling/"`.

[3] AMAZON WEB SERVICES. Elastic Load Balancing.

[4] CHANDRA, A., GONG, W., AND SHENOY, P. Dynamic resource allocation for shared data centers using online measurements. In *ACM/IEEE International Workshop on Quality of Service (IWQoS)* (2003).

[5] COMPUTERWORLD. Context on ice: Penguins fans get mobile extras. `"http://www.computerworld.com/s/article/9134588/Context_on_ice_Penguins_fans_get_mobile_extras"`.

[6] GMACH, D., ROLIA, J., CHERKASOVA, L., AND KEMPER, A. Capacity management and demand prediction for next generation data centers. In *IEEE International Conference on Web Services* (2007), pp. 43–50.

[7] GONG, Z., GU, X., AND WILKES, J. Press: Predictive elastic resource scaling for cloud systems. In *International Conference on Network and Service Management* (2010), pp. 9–16.

[8] LI, J., J. CHINNECK, M. WOODSIDE, M. L., AND ISZLAI, G. Perfomance model driven QoS guarantees and optimization in clouds. In *ICSE Workshop on Software Engineering Challenges of Cloud Computing* (2009), pp. 15–22.

[9] MAO, M., LI, J., AND HUMPHREY, M. Cloud auto-scaling with deadline and budget constraints. In *IEEE/ACM International Conference on Grid Computing* (2010), pp. 41–48.

[10] MOBILE MARKETER. 45.7% of sports fans use smartphones to access content online. `"http://www.mobilemarketer.com/cms/news/research/14020.html"`.

[11] ONLINE MEDIA DAILY. Super sports fans engage on mobile. `="http://www.mobilemarketer.com/cms/news/research/14020.html"`.

[12] PADALA, P., SHIN, K., ZHU, X., UYSAL, M., WANG, Z., SINGHAL, S., MERCHANT, A., AND SALEM, K. Adaptive control of virtualized resources in utility computing environments. *ACM Operating Systems Review 41* (June 2007), 289–302.

[13] ROY, N., DUBEY, A., AND GOKHALE, A. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *IEEE International Conference on Cloud Computing* (2011), pp. 500–507.

[14] SHIVAM, P., BABU, S., AND CHASE, J. S. Learning application models for utility resource planning. In *International Conference on Autonomic Computing* (2007), pp. 255–264.

[15] YINZCAM, INC. `"http://www.yinzcam.com"`.