outsystems

Massively Scaling Mobile Apps

2017

OutSystems Engineering

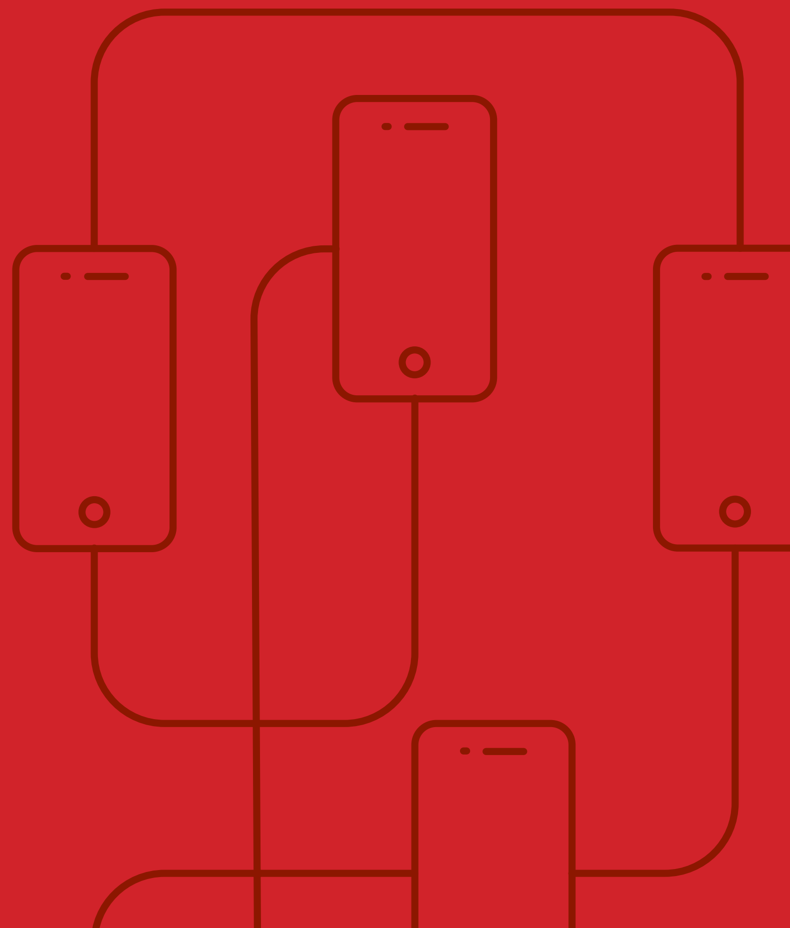


Table of Contents

Introduction	**3**
Getting Set Up	**4**
The Old Black Friday Magic: 11,250 Requests Per Second	**8**
Moment 0: The First Test	**11**
Milestone 1: 3,000 Concurrent Users	**12**
Milestone 2: 3,750 Concurrent Users	**14**
Milestone 3: The 7,500 Users Flash Mob	**19**
Conclusion	**22**

Introduction

People have high expectations of enterprise application performance based on their experiences with consumer apps used by millions. These apps effectively defined the standards and user frustration thresholds that make or break businesses.

From a user's standpoint, consistently good performance is a given; anything less is a trigger for abandonment. Maintaining that consistency is, therefore, key. And a challenge OutSystems felt an urgent need to address.

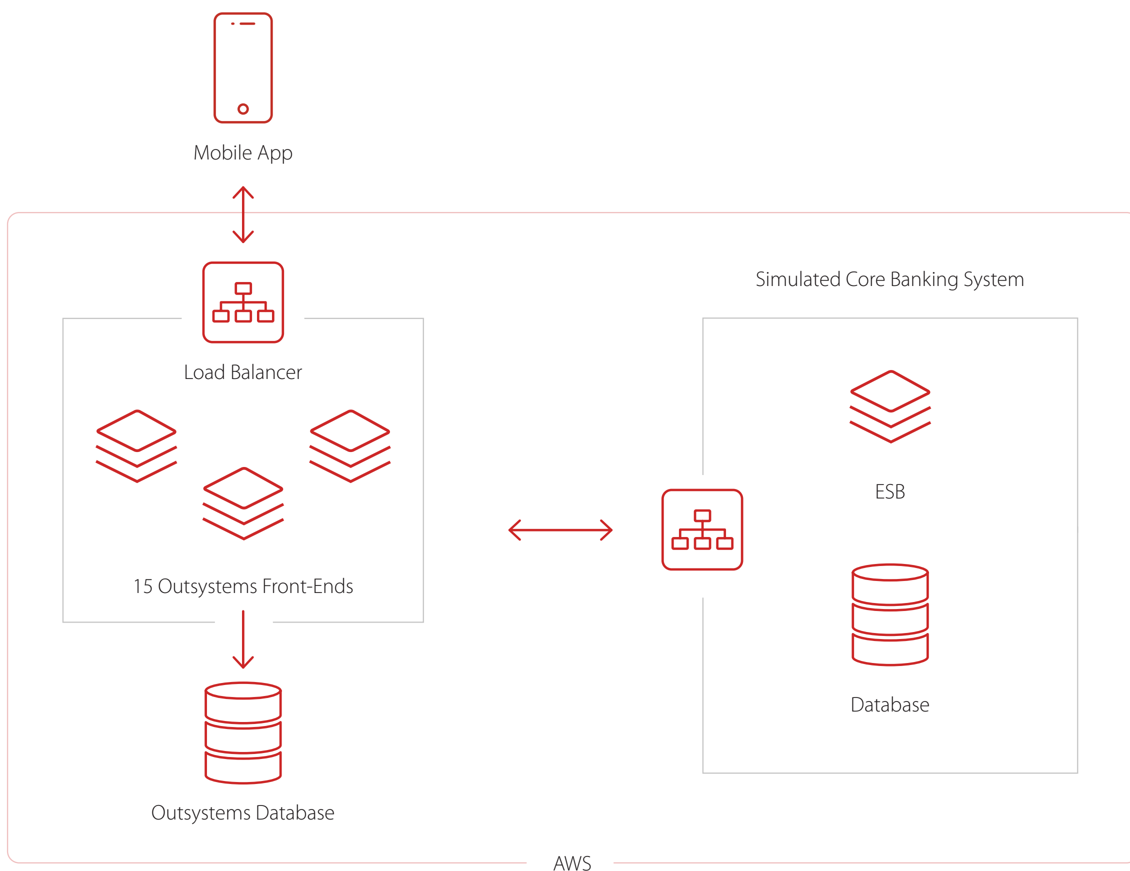
The OutSystems low-code development platform is designed to scale vertically and horizontally. Its underlying distributed architecture–with load balancing and the removal of single points of failure in its execution environment–is enterprise-grade. The platform, therefore, adjusts to specific requirements and can grow to deployments that support millions of users with automated resource management and optimization.

In this paper, we share the scenario, setup and testing used to prove exactly how massively scalable the OutSystems platform is.

Getting Set Up

The scenario for this paper is a mobile banking app. This type of app goes way beyond your run-of-the-mill "requests per second" stress test. It's massive: millions of users, with thousands of concurrent accesses, strain on CPU, massive throughput and extreme peaks. This is what we pitted against our OutSystems mobile back-end—the server side of OutSystems.

An OutSystems client—a large European bank—provided a baseline architecture and user volume for validating results when testing on this scale.



This scenario is not about building a core banking system. Instead, it simulates the banking Enterprise Service Bus (**ESB**) with the addition of an average baseline latency of 500 ms to each request, determined by measuring our customer's ESB responses over time in real-world applications.

The goal was to deliver consistently in a user comfort zone and then scale for massive concurrency. This would require making adjustments to the testing plan and the load thrown at the front-end servers.

The Targets

The targets for the scenario were application load and quality.

APPLICATION LOAD

Application load was based on 20 million active users, a publicly available number for the banking app of one of the world's largest banks. A very high level of concurrency was necessary also—one that went beyond the average for some of the biggest banking systems in the entire world. There was no other way to get a clear understanding of what happens in scalability situations for enterprise-grade systems. With the help of our client, the targets were set:

- Average requests per second in prime time: 2,915
- Maximum requests per second in prime time: 11,228

QUALITY

With a defined target for how much load, the next step was a target for **quality**. The goal was to provide a mobile user experience that would fit in user comfort thresholds. And, to stay clear of providing a broken experience that would drive up user frustration levels. Lessons from the web show that a one-second slowdown in page load time can cost businesses a lot of money. Beyond that, according to kissmetrics.com, one in four people abandons a website if its pages take longer than four seconds to load.

For a mobile app, users are even more demanding. It's hard to find a time threshold that people agree on. However, based on anecdotal evidence and experience, the mobile user's threshold was set at 1.2 seconds.

With that threshold, we would be able to apply Apdex: a measure of response time based against a set threshold. It measures the ratio of satisfactory response times to unsatisfactory response times. The response time is measured from an asset request to completed delivery back to the requestor." — New Relic's Apdex: Measuring user satisfaction.

THE APDEX FORMULA IS:

$$Apdex = \frac{Satisfied + (Tolerated/2)}{Total\ Requests}$$

So, say there are 200 requests to an application whereby 60 are served in less than 1.2 seconds (the threshold), 25 are served between 1.2 seconds and 4.8 seconds (4 times the threshold), and 15 requests take longer than 4.8 seconds to serve. These are the satisfied, tolerated and frustrated requests. Using the values, Apdex = (160 + 25/2)/200 = 0.862. Loosely speaking, this means about 86% of users are satisfied with the service.

The OutSystems goal was to get the highest Apdex possible for the 1.2-second threshold.

BASELINE CONFIGURATION

The configuration was right out of the box with no tuning whatsoever and a small tweak: an increase in the limit of database connections to 2000 to account for the simulation of **a lot** of users. We weren't sure how the servers would handle 1000 concurrent users with standard parameters and configurations, but we wanted to start from there and then fine-tune our settings as we reached the saturation points.

LAB INFRASTRUCTURE SPECIFICATIONS

The following configurations were used for the tests:

- **Front-end servers:** Amazon Web Services (AWS) m4.xLarge Instances: 4vCPU,16GB RAM.
- **Database:** SQL Server 2014 on an AWS Relational Database Services (RDS) db.m4.2xlarge server: 8vCPU, 32GB RAM. (The first test was done with SQL Server 2014 on an AWS RDS db.m4.xlarge server: 4vCPU, 16GB RAM.Based on the results of the test, we increased its performance.)
- **ESB:** This part of the system involved a second infrastructure running an OutSystems app that simulates a latency of every request to a fixed value of 500 ms.
- **Test servers:** JMeter and AWS instances with enough memory and throughput so the testing tool wouldn't become a bottleneck. The strategy consisted of **one orchestrator** instructing other distributed servers to inject the requests into the platform and then sending the results back to the orchestrator for final processing and reporting generation. Because the amount of data was immense, scaling with the values proposed required one JMeter **distributed server** for each front-end added to the OutSystems farm.

The result was an astronomical setup. We had one r4.xlarge server for each m4.xlarge front-end server. Say we'd have 10 OutSystems front-end servers: we would couple that with 10 JMeter servers. A dream setup for any JMeter enthusiast.

SETTING UP THE USERS

The scope was a login and user database with hundreds of thousands of unique users. Authentication was solely against OutSystems front-ends.

The universe of unique users was defined to 200,000 to start the tests. Scaling up and rotating the users (multiple logins for each user) turned out not to be a variable for the response times.

The test plan also included two cookies and cache manager elements that forced each user to be a completely independent iteration from the previous one, thereby allowing massive concurrency without the same unique user being logged in more than once.

THE MOBILE BANKING APPLICATION

The application for the scalability test is a regular mobile banking app, with all the screens you would expect:

- A login page with touch ID recognition for alternative authentication
- A screen where users can access specific accounts and recent account activity
- An overview screen with aggregated details for each account the user has

These represent 60-70% of the transaction load as reported by our European bank customer. A user interaction with the following steps was simulated:

1. User login
2. Synchronization for offline (yes, the app was built to work offline)
3. Fetch overview for 1st account
4. Fetch overview for 2nd account
5. Fetch overview for 3nd account
6. Get activity for 1st account
7. Get activity for 2ndt account
8. Logout (not a lot of people log out, but we are slightly OCD about these things)

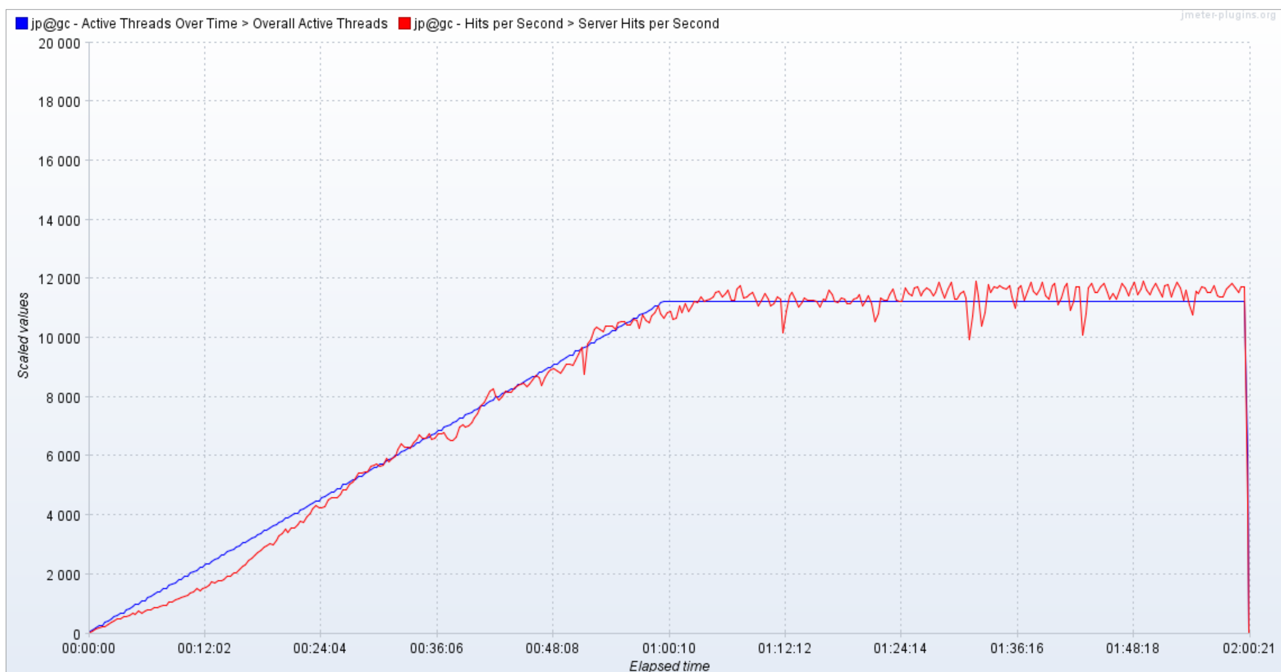
In total, these represent 15 different requests to OutSystems front-ends, eight of which need to get information from core systems, and therefore interact with the ESB.

With our configuration and targets in place, this happened.

That Old Black Friday Magic: 11,250 Requests Per Second

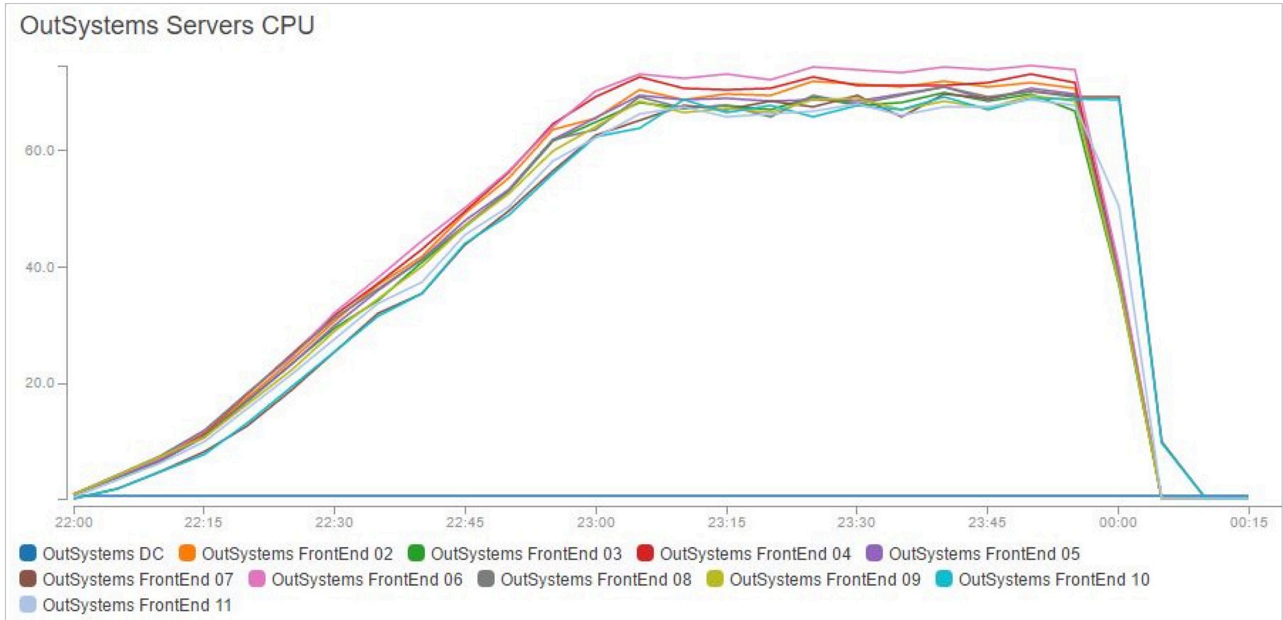
Our initial challenge was to scale to Bank of America numbers. But, we did not want a regular Bank of America scenario; it had to be more extreme, like "Black Friday." In other words, the goal was a huge amount of transactions (similar to those on the day after Thanksgiving in the US), an amount that normally brings enterprise-grade setups to their knees.

We tested millions of users, with thousands of concurrent accesses per second. Our goal was the magic figure of 11,250 requests per second. The number of users was 750 per-front end, a number we knew would be sufficient after testing to see that we didn't hit the threshold for network, CPU processing power or IO. So we did 11,250 over 750, scaling our setup to 15 front-ends. These were the results.

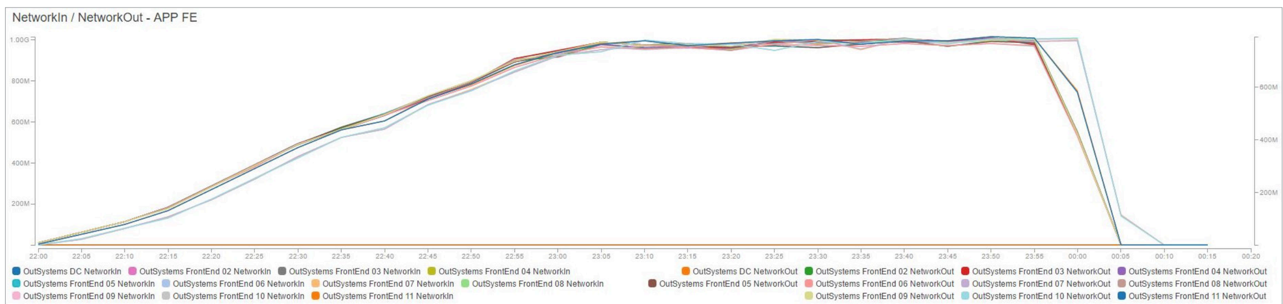


© 2017 OutSystems Technical White Paper - Massively Scaling Mobile Apps

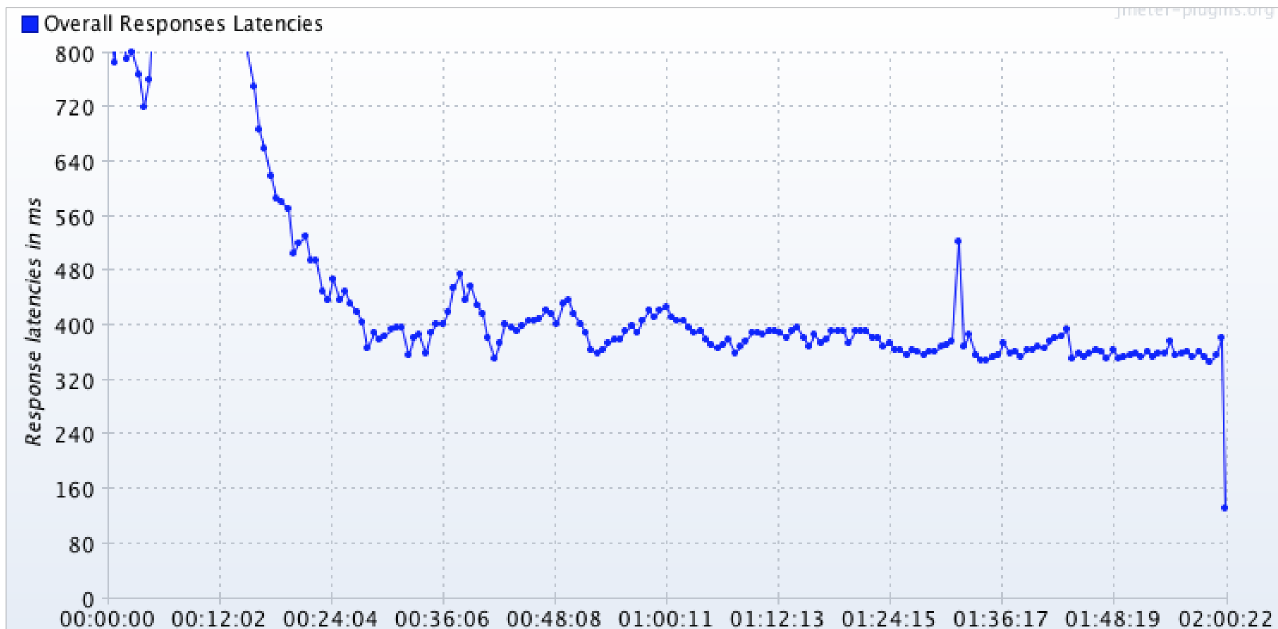
As you can see, OutSystems provided linear growth between users and requests per second. And, the CPU and network were within thresholds:



This was the impact on users:



This was the impact on users:



Latency chart for our 11,250 requests/sec burst. First 1 hour is warm-up of all the servers

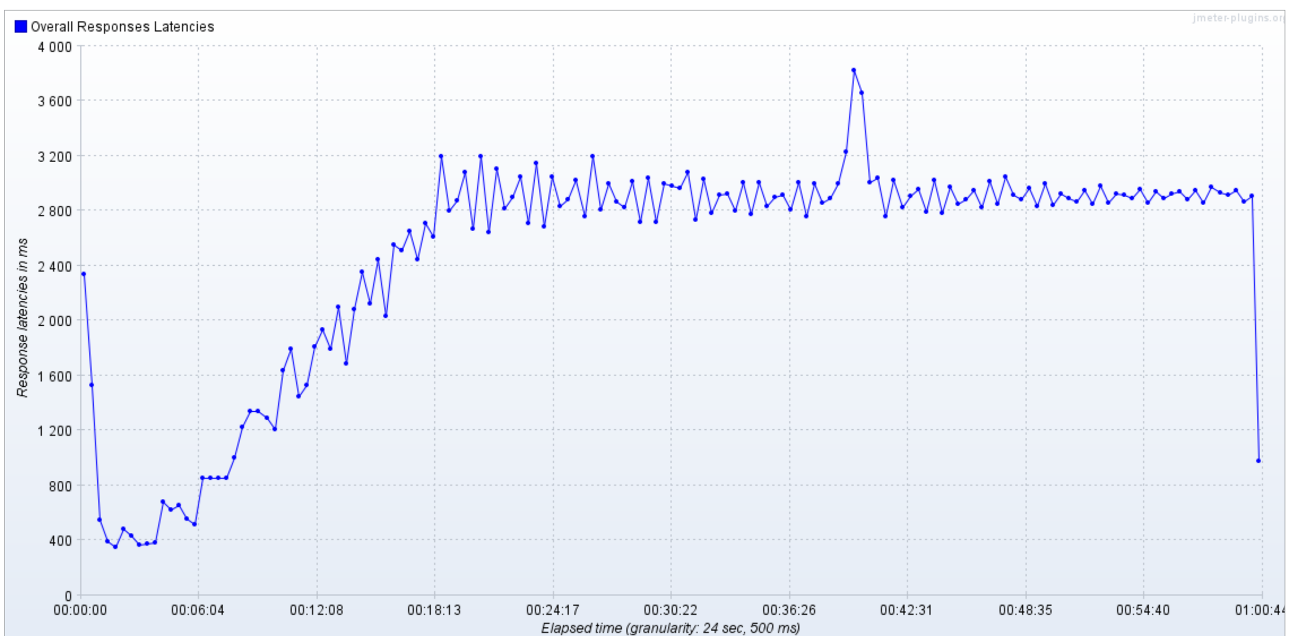
And our Apdex? Way up there, at 0.96.

There's no question that these numbers indicate the massive scalability of OutSystems. However, it did take some trial and error to get there. The following sections describe the tests and adjustments that resulted in our astounding final test numbers.

Moment 0: The First Test

We started with two front-ends and 1,000 concurrent users with the ramp-up defined as 10 steps (100 users every 2 minutes). So, the system was at full load about 18 minutes into the test.

This setup handled around 280 hits per second on the small farm, and the latency was about 3 seconds at full load, which is more than double the 1.2-second threshold target.



Latency over time. At about 18 minutes, the system is at full load.

Latency increased as the number of threads increased, so the next step was to make some adjustments.

Milestone 1: 3,000 Users

For the next test, we used the same architecture but with these changes:

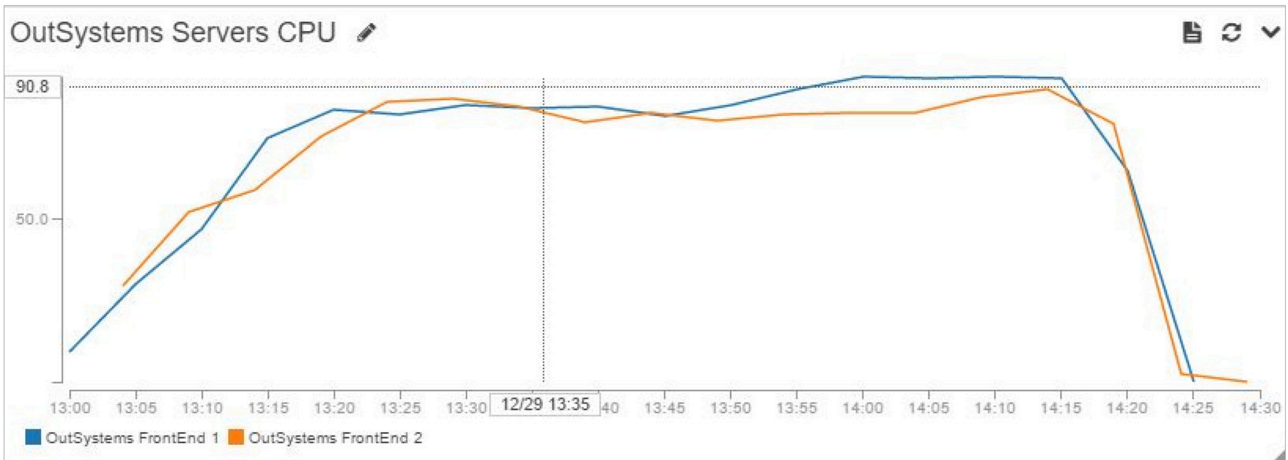
- Database server: Capacity increased
- Maximum number of database connections: 2,000 per front-end
- IIS: From classic to integrated mode. This removed limitations on maximum number of requests and executing threads per CPU.

Interesting fact:
Since IIS 7.0, integrated mode has been Microsoft's recommendation and it is now the default configuration for OutSystems.

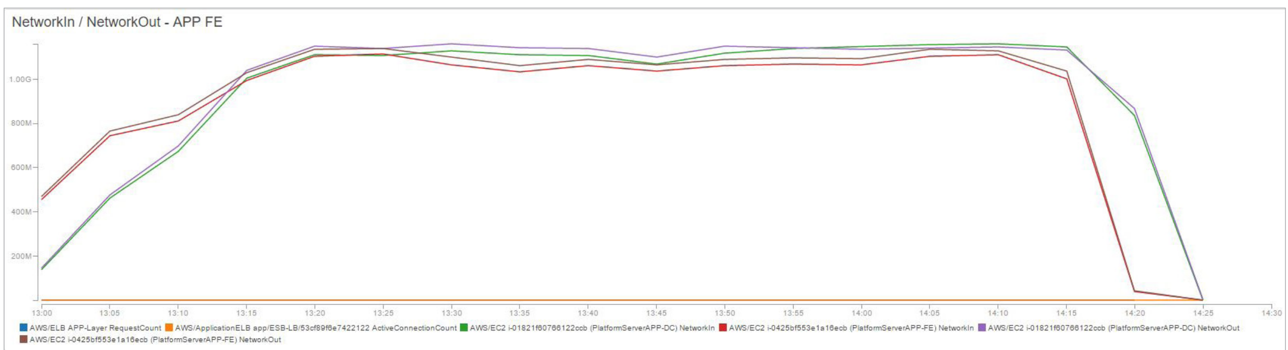
The 3,000 concurrent users were run with the same 10 step-profile as the previous test, and the results were markedly better: 1,650 requests per second on max load with latency dropping to an average of about 1.3 seconds.

Although these latency results were half those of the first test, they remained above the threshold value of 1.2 seconds, and therefore were far from the Apdex goal.

In addition, close to 7 million requests were sent to the front-end server in less than an hour and a half. There were 55 errors, or an error rate of 0.0007902%. Although that's an impressive rate, OutSystems was capable of doing better. Most companies would love that score, but for us it wasn't good enough. Not to mention that we were close to some server, CPU and network limits:



The CPU on the front-ends was very busy...



...and so was the network...

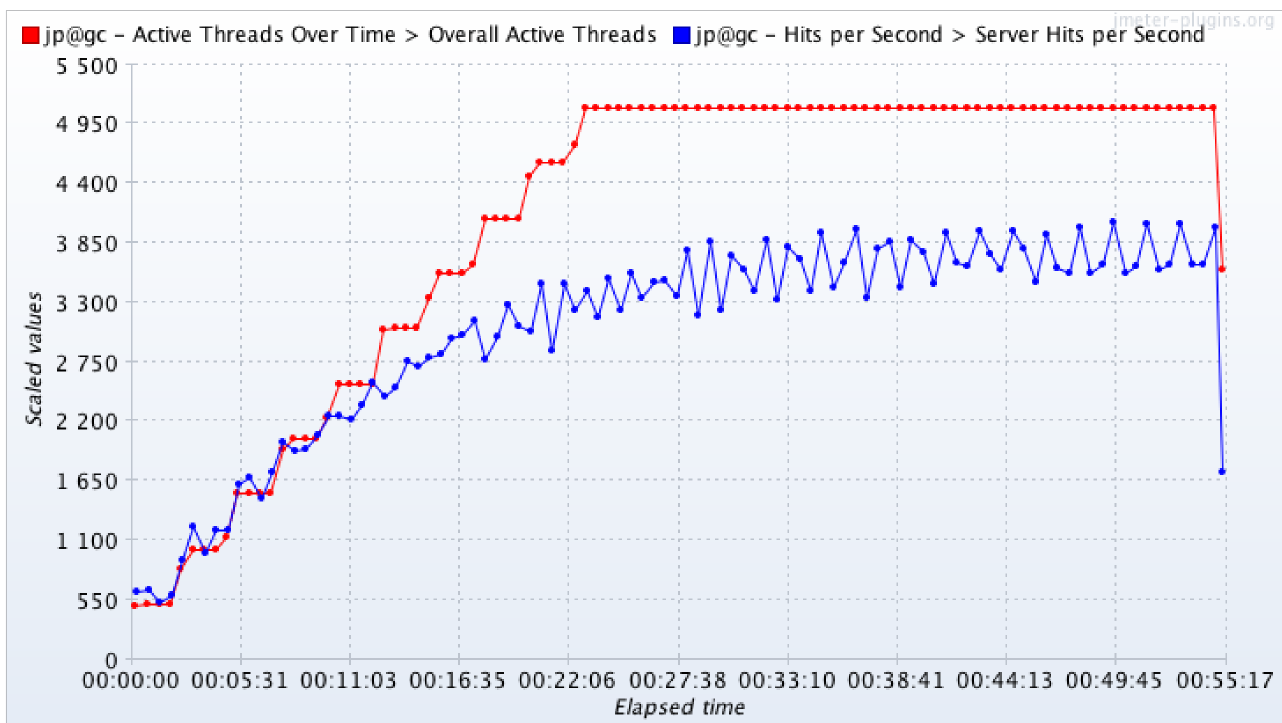
Fortunately, solving these issues was easy. CPU and bandwidth limitations are per Amazon instance, so front-ends were added.

Milestone 2: 3,750 Concurrent Users, 5 Front-Ends

For this test, we increased the number of front-ends to 5. It was also time to stop estimating the number of concurrent users and use math for scalability planning. Upon examination of the testing to this point, we saw each test took at least 15 seconds to execute. This time was based on the number of requests completed, the time for simulating user interaction, the time to process the request, and the waiting time for the ESB. Fifteen seconds is the minimum expected time for a full test, running on a server with no CPU, bandwidth or other limitations.

Coincidentally, each test handled 15 requests. They took different amounts of time but, on average, each test will put a load of 1 request per second on our servers. So, if we increase the number of users, the number of requests per second delivered by the server should keep up with that value. For example, 2,000 virtual users should lead to 2,000 requests/sec on the server, and so on.

After preparing the five-front-end infrastructure, we started bombarding the servers with requests. There were 5,100 virtual users (corresponding to 5,100 active threads):



5,100 users and five front-ends

At the start of the test, both active threads and requests per second grew at the same rate (hits per second is JMeter's name for requests per second). This makes sense, since 1 active thread results in 1 request/sec.

As time advanced, the transactions per second started to diverge from hits per second, which is related to the warm-up time. But the interesting part is that the number of requests per second started stabilizing at between 4,000 and 3,300 requests per second. Since there's a one-to-one relationship between requests per second and users, it appears that the five-front-end farm can handle between 4,000 and 3,300 before users start getting a penalty in terms of response time.

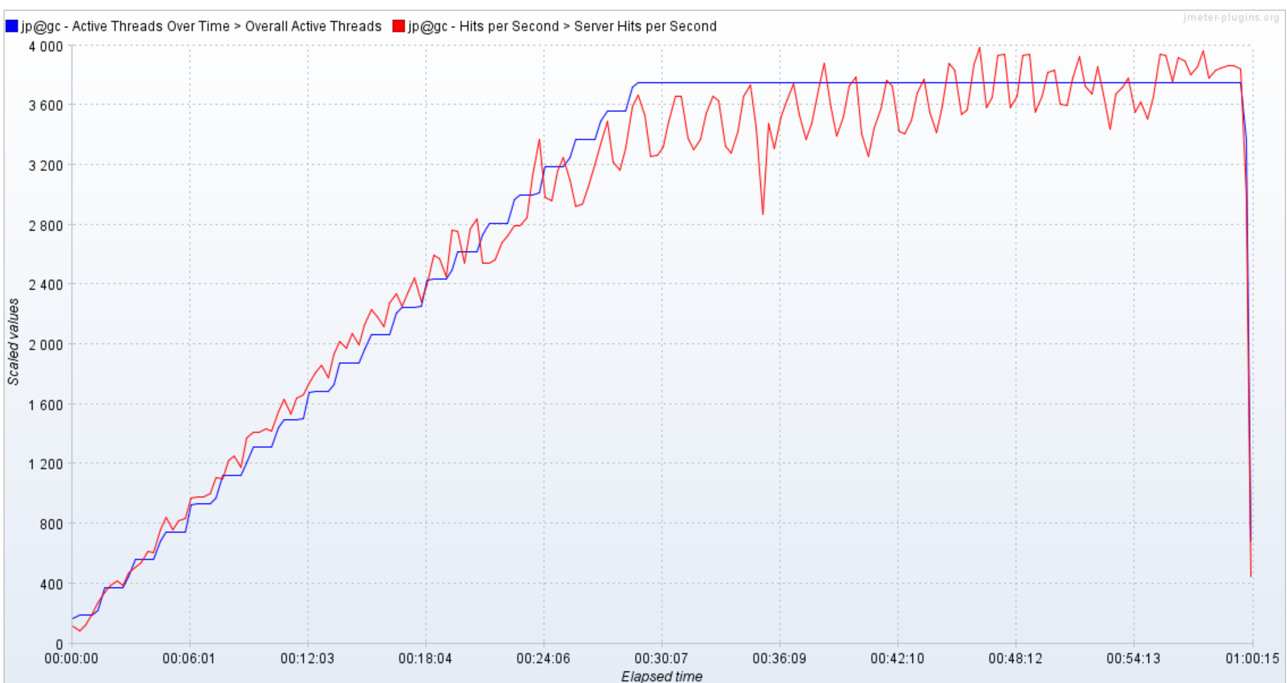
Because there are 5 front-ends, dividing those numbers by 5 indicates that OutSystems can handle between 800 and 660 concurrent users per front-end.

DOUBLE-CHECKING THE MATH

To make sure the math was correct, we ran tests with 600, 650, 700, 750 users per front-end, until we were sure we wouldn't hit the threshold for network, CPU processing power, or IO. That's how we got a number: **750 per front-end**.

In addition, our 8vCPU RDS DB was holding up. And, to be able to put in the load we wanted on the servers, we created one JMeter injector per front-end and the tests were being launched remotely through a separate JMeter orchestrator.

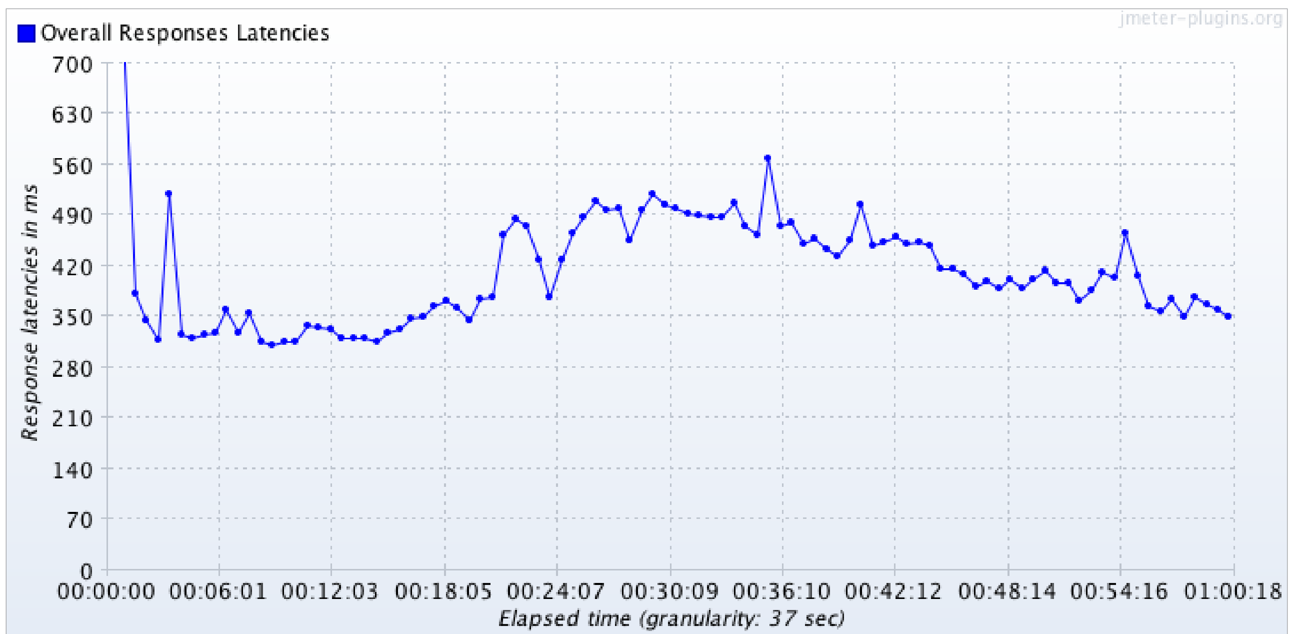
We were now confident about our numbers. **Five front-ends** means 5 x 750 virtual users: **3,750 virtual users**.
We pushed these users through 20 steps over 30 minutes, then we held maximum load for 30 minutes:



The result was what one might consider a thing of sheer beauty. It's a linear growth of requests per second according to the number of users.

Notice those fluctuations? They were expected because the testing script has **think times** in it, with some randomization included. **It tries to simulate human behavior**, and the hits are not constant during execution. The objective was achieving a certain number of requests based on having concurrent **human** users doing stuff in our use-case scenario.

What about latency for the users?



The spike right at the start can be discarded, because most of it is a warm-up period for the servers. As soon as it's clear, things start looking pretty good.

THE APDEX VALUE FOR THIS RUN

Our threshold was 1.2 seconds, but it's possible to be even more demanding. The latency measured is the time the data takes to get to the device while taking into account the time it takes to render the data on the device. To simplify, assume all screens take the same time to render: 370 milliseconds. This value is the time it takes to render a list with 100 elements in a Nexus 4, based on performance tests that were part of optimizing the platform.

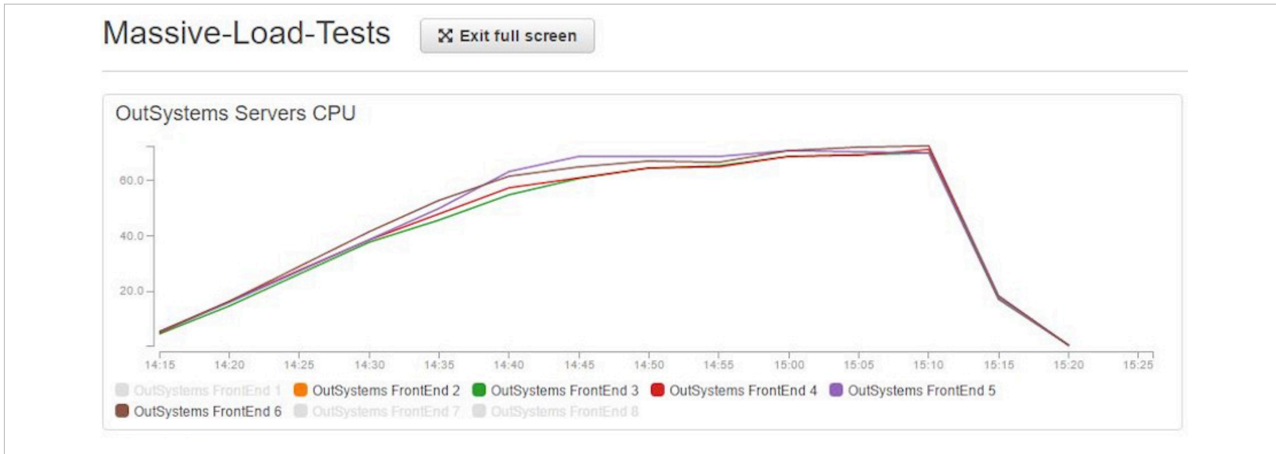
With a threshold time of 830 ms (1.2 seconds minus 370 milliseconds), the next step is to count the number of requests that satisfy the user (latency under 830 ms), the ones the user tolerates (latency under 4 times the 830 ms threshold), and divide them by the total number of requests. We did this for all requests that happened after the warm-up period.

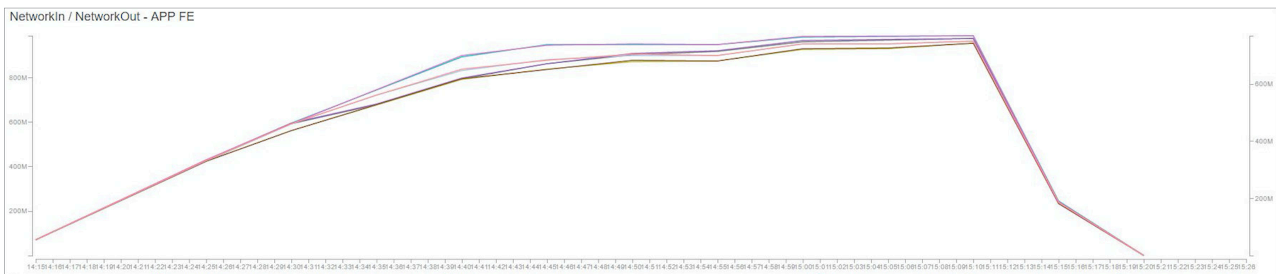
Screen	Satisfies	Tolerates	Total	Apdex
Accounts	341,539	99,542	441,521	0.89
Integrated Position	1,472,362	289,540	1,764,077	0.92
Movements	1,021,913	300,638	1,323,555	0.89
Login	439,779	1,642	441,487	0.99
Logout	439,789	1,651	441,487	0.99
Offline Sync	397,453	43,896	441,497	0.95
Total	**4,112,835**	**736,909**	**4,853,624**	**0.92**

These are really amazing results. It means the worst operations have an Apdex of 0.89: 89% of the customers are satisfied. Even more impressive was the the overall Apdex of 0.92.

CHECKING ON THE SERVERS

And what about the servers? Were they begging us to stop? Had we pushed them to capacity overload? Here's the answer:

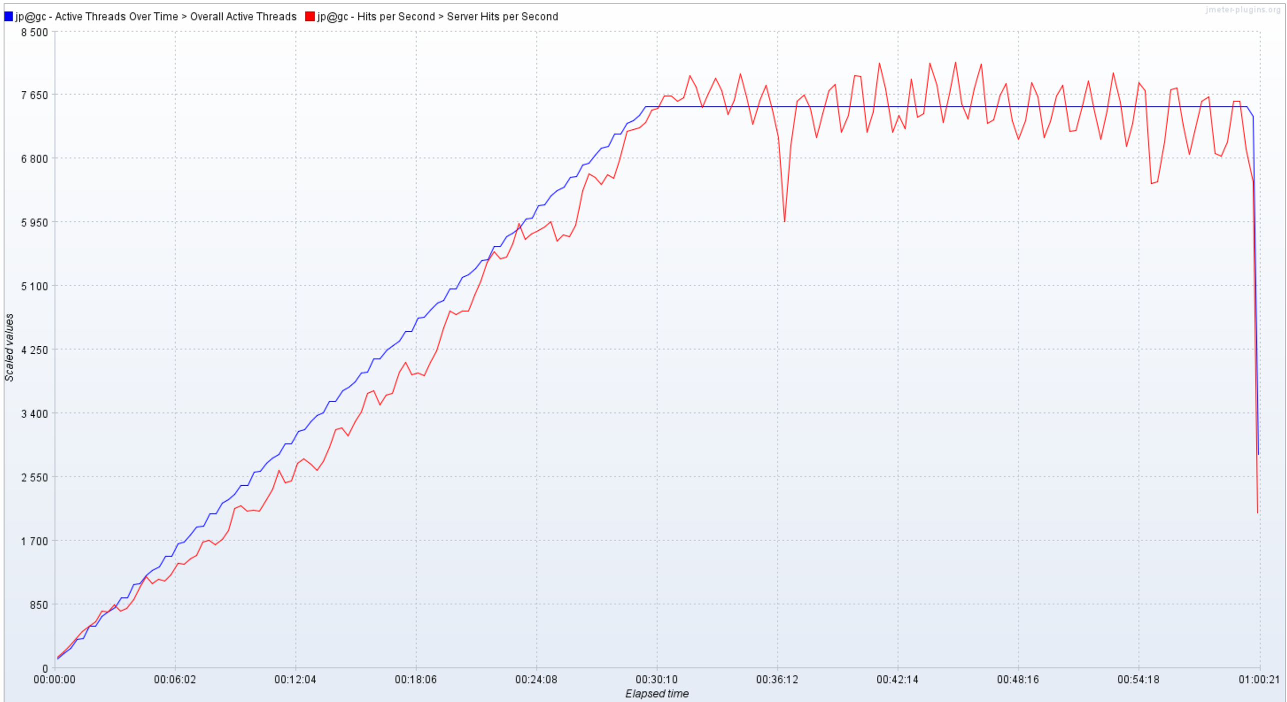




Which leads us to...

Milestone 3: The 7,500 Users Flash Mob

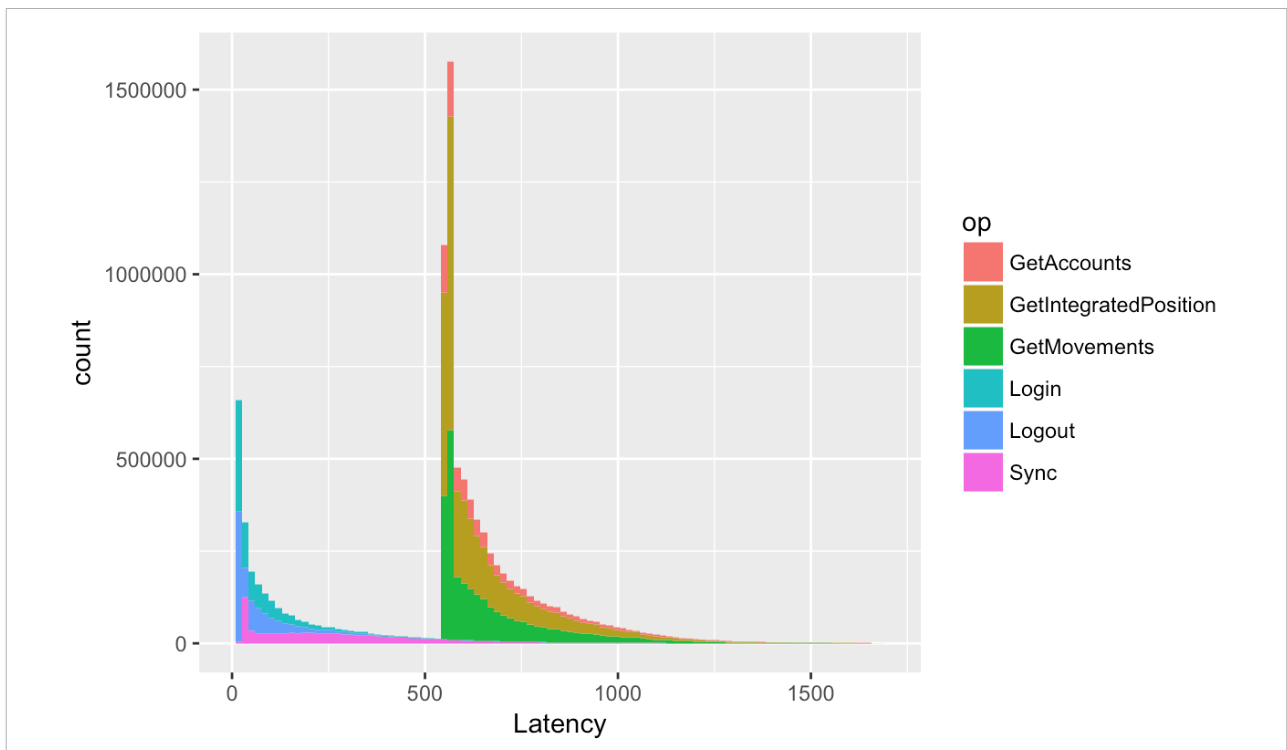
Our next round of testing was for 7,500 concurrent users which, at 750 users per front-end, meant the goal was for 10 front-ends to handle the load and answer flawlessly. As you can see, our setup met that goal:



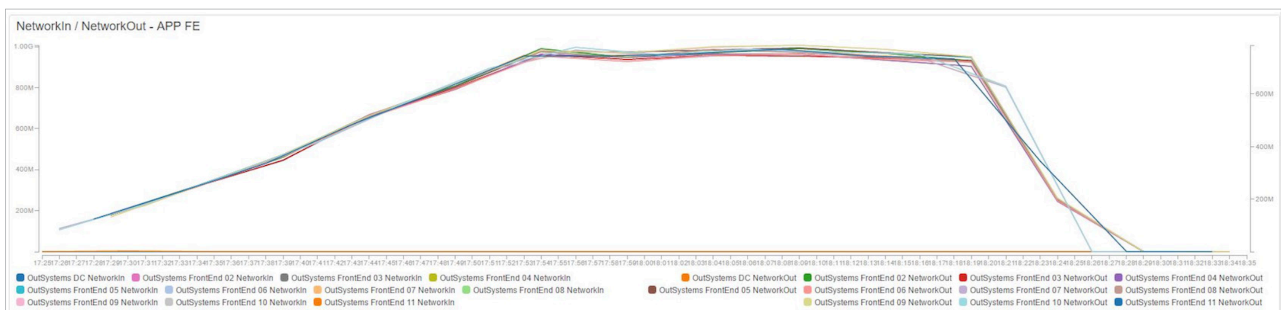
But, how fast were users getting their answers on their mobile devices?

These values were very good. The Apdex was even better: a value of 0.94, using the same 830 milliseconds as a threshold.

So, all the operations had an Apex of above 0.91. But it's still interesting to see how they plot in terms of frequency of latency. It becomes pretty obvious which operations are affected by the 500-millisecond penalty introduced by the ESB.



The servers, as expected, kept up with the demands:



And, after this test, we got the Black Friday success story described at the beginning of this paper.

Conclusion

These milestones all made one thing clear. With a few adjustments and some fine-tuning, OutSystems runs like clockwork in a "Black Friday" scenario. We eventually surpassed Bank of America's results with our solution. So, you can be confident that OutSystems can scale to meet whatever demands your front-end users put on it.

And, do you know what we call this?
Craftsmanship. Engineering Craftsmanship.