WHITE PAPER

Get performance requirements right—think like a user

By Scott Barber

Performance testing is hard. It's hard technically, logically, logistically and managerially. In my years of experience as a performance-testing consultant, I've seen many performance-testing challenges conquered. I've also seen many challenges that the software testing industry as a whole continues to face, but with which it has made virtually no progress since Y2K.

It doesn't surprise me that performance testing is still largely under-done, under-funded and under-scheduled. It surprises me a little bit that performance testers are still finding it difficult to interact directly with development teams due to organizational structure, but this is changing. It surprises me a lot that we seem to have made no progress at all in the areas of determining application performance requirements, goals and meaningful objectives for the performance-testing effort.

This surprises me because my experience suggests that when we do successfully verbalize—not even quantify, but simply verbalize—application performance requirements and goals, plus the objectives of performance testing, the team finds a way to overcome technical, logical, logistical and managerial challenges to achieve success: a well-performing application.

Keep that in mind as you read the remainder of this paper. Whether you are a manager, tester, developer or analyst, the following discussion applies to you. For organizational purposes, the paper is broken down into several areas through which we will walk sequentially to establish requirements, goals and objectives that will significantly assist your team in assuring adequate application performance—or at least expected application performance. Don't worry too much about whether or not your project applies these concepts in the same organizational areas or in the same sequence; the key is simply to think about the concepts and how you can apply them on your project.

Remember that when all is said and done, only one performance requirement really matters: Making sure application users are not annoyed or frustrated by poor performance. Application users don't know or care about the results of performance tests, how many seconds past the "too long" threshold it takes a display to appear on the screen or what your throughput is. The only thing users notice is whether or not the application seems slow—and whether they notice is based on anything from their mood to the application speed with which they have become accustomed. We'll talk about how to convert these feelings into numbers, but never forget to validate your quantification by putting the application in front of real users.

Identify critical business transactions

Before we can determine the desired performance characteristics of an application, we have to understand two things: what the application does and how we expect it to be used. Although it might be easier if we could reference a table of industry-standard response times for various actions or activities, no such standard exists. Some have been proposed, but for every proposed performance standard, there are at least 100 examples of cases in which achieving the proposed standard wouldn't adequately satisfy users of the application.

To identify the most critical business transactions a performance test needs to include, think in terms of:

- frequently used transactions
- performance-intensive transactions
- business-critical transactions.

An example

If we applied the above thinking to a generic online bookstore application, we might find that the most frequent transactions are "search," "add to cart" and "login"; the most performance-intensive transactions are "search" and "view my order history"; and the most important business-critical transactions are "order items," "create account" and "check order status." Other available transactions might be "view FAQ," "update account information" and "rate this book."

Practical justification
Experience proves that if we give the transactions that fall into one of these categories the highest priority in performance testing, we are more likely to achieve success than if we start by assuming every transaction will be part of performance tests. Why? Because we almost never have time to build a test that exercises every possible transaction. And let's face it: What are the odds of losing a customer over the sluggishness of a "FAQ" or "rate this book" feature? Certainly, if you have the time and resources to test transactions beyond those that are most critical, it's even better, but if you get that far, I encourage you to periodically ask yourself, "What is the most valuable test I can develop or execute right now?" In the vast majority of software testing efforts, there isn't enough time to test everything, so you should ask yourself that question after every test you run.

Implications on test data
Once you determine the most critical business transactions for purposes of performance testing, you can begin designing test data. There are two reasons for moving directly to test data. First, depending on your application, test data may be complex to generate or take time to extract from existing production data. Frequently, the majority of test data comes from a production database supporting a previous version of the application. When that is the case, production data often contains confidential information that must be sterilized before it can be used—which can consume a significant amount of time. The second reason to identify test data early is once you know which transactions to focus on and have test data defined, you can begin executing performance tests immediately after the developers complete a beta release of the code supporting those transactions.

Bear in mind several key considerations when designing test data for performance testing. The most obvious consideration is volume. In one of the most frustrating scenarios, an eight-hour stability test will fail and crash the system—after executing for 7.5 hours—because your test scripts ran out of data. Another challenge is the requirement for unique data. If some or all of your data has to be unique—for instance, new users may need to have a unique e-mail address—you could need literally tens of thousands of e-mail addresses, all of which may need to be aliased to a catch-all account or filtered by the corporate mail server to keep it from getting overloaded, since the application sends out confirmation of registration messages. Performance tests are regularly executed many, many times during both script development and testing. If you can't refresh the database at will, you will go through a lot of unique data during your performance-testing effort. So no matter how much data you think you need, generate as much as you reasonably can; you are likely to need it before testing is done.

Speaking of unique data, many a performance test has been proven invalid as a result of insufficiently unique test data. For example, if every user in the bookstore example searches for the same book, that search is likely to be stored in a cache somewhere, thus effectively eliminating the database from the tests. This classic situation often leads to performance statistics that are not only wrong but have the side effect of leading the team to believe an application's performance is significantly better than it is.

Another key for designing test data is distribution of data. It is absolutely critical to have a realistic distribution of test data to achieve valid performance test statistics. It is equally important to remember your users are going to be random. They will enter data you'd never expect, and unexpected data affects performance the most. Consider the volume of data that would be returned if a bookstore user searched for all books containing the word "the" in the title. One wouldn't expect a user to do that, but trust me, I've seen production log files: It really happens.

Finally, consider including invalid data in your performance tests. Users will enter invalid data that exercises different components of the system than valid data, thus changing performance results. Luckily, in most cases, error trapping invalid data is less performance-intensive than processing valid data. In these cases, inclusion of invalid data in performance tests will lead to slightly better results than non-inclusion, thus misleading you to think the software is performing adequately when it isn't. You need to know your application and make an informed decision rather than blindly choosing to include or exclude invalid data based on guesses, assumptions or experience with previous applications.

One last thought on performance test data design: The best possible test data is test data collected from a production database. The next best test data is that which is collected through beta releases and/or user-acceptance testing. If at all possible, get data from real-world usage. No data generated by testers will ever represent users better than actual data from human users. A word of caution—be careful when using sensitive production data that may violate regulatory or privacy rules. If this is a concern, consider data privacy solutions that can scramble or generate appropriate test data in the testing environment.

Determine speed criteria for critical business transactions
Once you have identified the business transactions to design performance testing around, you can begin the process of verbalizing performance requirements and goals for those transactions, as well as performance-testing objectives. But before discussing how to accomplish this task, let me first define requirements, goals and testing objectives.

A new way to think about performance requirements
Performance requirements—criteria that are absolutely non-negotiable due to contractual obligations, service level agreements or business needs. Only those criteria whose sub-par performance would unquestionably lead to a decision to delay a release are absolutely required.

Performance goals—criteria desired for release, but negotiable under certain circumstances. For instance, if a response time goal for a particular transaction is set at 3 seconds, but the actual response time is determined to be 3.3 seconds, it is likely stakeholders will choose to release the application and defer performance tuning of that transaction for a future release.

Performance testing objectives—items that add value to the team through the process of performance testing but are not intrinsically quantitative. For example, one objective might be to provide certain data to systems administrators to assist them in tuning systems under their purview. Another objective might be to determine peak application usage that the existing network can support.

Capturing requirements, goals and objectives
Using this terminology, performance requirements are quite easy to capture. Just review any contracts and legally binding agreements related to the software under development and get executive stakeholders to commit to any performance conditions that will cause them to hold up release of the software into production. The resulting criteria may or may not be related to any specific business transaction or condition, but if they are, those transactions or conditions must be included in performance testing.

Performance-testing objectives are also fairly easy to capture. The easiest way is to ask each member of the team what he or she would like tested. That might include providing resource utilization data under load, generating specific loads to assist with tuning an application server, or providing a report of the number of objects requested by each web page. Collecting performance-testing objectives early in the project is a good habit to get into; periodically revisiting them and checking in with team members to see if they would like new objectives added are equally beneficial.

Performance goals are tricky to both capture and quantify. Reports from many of the best performance testers around the world corroborate my experience that capturing and quantifying goals should be treated as separate activities. In my opinion, the single most common mistake related to performance testing is jumping straight to quantification without first verbalizing goals qualitatively. I strongly recommend capturing performance goals for both critical business transactions and the application as a whole in subjective, qualitative terms first. For example, ideal initial performance goals would be "no slower than the previous release," "at least as fast as our competitors" and "fast enough that the overwhelming majority of our potential users will not feel frustrated by poor performance."

Quantifying goals
After goals are captured qualitatively, you can begin the process of quantifying them. To quantify a goal of "no slower than the previous release," simply execute an equivalent performance test against the previous release and record the results as a baseline for comparison. To quantify a goal of "at least as fast as our competitors," take a series of single user performance measurements of competitors' software. Quantifying end-user satisfaction and/or frustration is more challenging, but, at least for our purposes, far from impossible.

All you really need to quantify end-user satisfaction is an application and some representative users. You don't need a completed application; a prototype or demo will do for a first pass at quantification. With just a few lines of code in the HTML of a demo or prototype, you can control how long it takes each page, screen, graphic, control or list to load. Using this method, create several versions of the application with different response characteristics. Then users can try each, telling you in their own words whether they find that version to be unacceptable, slow, reasonable or fast. Since you know the actual response times, you can start equating those numbers to users' reported degrees of satisfaction. It's not an exact science, but it's a very good starting goal—especially if you follow up by asking the same questions about performance testing every time you put an application in front of someone, be it for functional, user-acceptance beta testing or some other reason. That way, you are measuring response times in the background as users interact with the system, allowing you to collect more data and enhance your performance goals as the application evolves.

While you are quantifying performance goals with actual users, it's also a good idea to collect data for other timing-related issues. For instance, in the absence of log files to parse for actual production data, the best way to determine how long users spend reading or interacting with each page or screen is to observe them. Detailed observations of business transactions will be highly valuable later for creating tests that represent actual users as closely as possible.

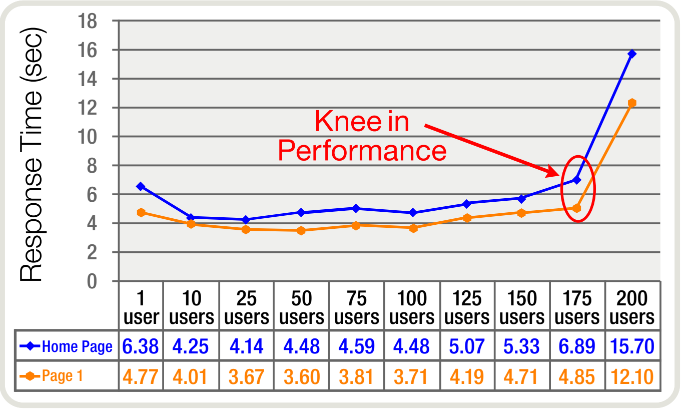
Determine application scalability and capacity criteria
Scalability and capacity, both highly technical areas, are tightly related and often fall under the umbrella of application performance. In these

two areas, you can define quantity and size criteria to correspond with your speed criteria. Although the terms scalability and capacity are frequently used interchangeably, they are quite different in critically important ways. Scalability concerns the change in performance characteristics when an application experiences increased usage. Capacity is a reflection of size and volume limitations—typically related to hardware and configuration. An application may scale poorly as a result of a capacity limitation, but it may scale poorly for any number of other reasons as well. In the same way, capacity limitations don't always reveal themselves during scalability testing.

How to think about scalability

Up to this point, our discussion of goals and requirements for speed has centered on one user at a time—which makes sense because a user doesn't know, or care, how many others are using the application or web site. That means in a perfect world, no amount of user volume will cause any degradation in speed from the user perspective. Of course, we know this isn't the way things really work. That's where scalability comes in.

At some point, every application will experience a usage volume that causes speed to be noticeably affected. Inevitably, once that volume is reached, it takes very little additional volume to slow the application down to an unusable rate. Performance testers frequently refer to the volume at which performance begins to degrade quickly as the "knee" in performance because of the way the condition looks when graphed (see Figure 1).



	1 user	10 users	25 users	50 users	75 users	100 users	125 users	150 users	175 users	200 users
Home Page	6.38	4.25	4.14	4.48	4.59	4.48	5.07	5.33	6.89	15.70
Page 1	4.77	4.01	3.67	3.60	3.81	3.71	4.19	4.71	4.85	12.10

Figure 1: Graphical depiction of scalability

Looking at Figure 1, notice response time, or speed, stays relatively stable until usage volume reaches the "knee." We say the application is "scaling gracefully" in this range of volumes prior to the knee. Our goal, obviously, is to have an application that achieves speed goals and volume of usage goals before reaching the knee—which leads to the question, "How do we determine volume of usage (or scalability) goals?"

Quantifying the volume of application usage

Determining and expressing an application's usage volume has been notoriously confusing since the advent of multi-user applications that communicate via stateless protocols (i.e., Internet-based applications). Terms like "concurrent users" and "simultaneous users" have been used frequently (and misused almost as frequently) since then. Rather than advise you to avoid those terms at all cost, I will explain what they actually mean.

In Figures 2 and 3, each line segment represents a user activity, and different activities are represented by different colors. For the sake of this discussion, the red-line segment represents the activity of "Load the Home Page." Users (or possibly sessions or threads) are represented horizontally across the graph. For simplicity's sake, let's assume the same activity takes the same amount of time for each user. The time elapsed between the "Start of Model" and "End of Model" lines is one hour. Let us first look out from the perspective of the server (in this case, a web server). See Figure 2.

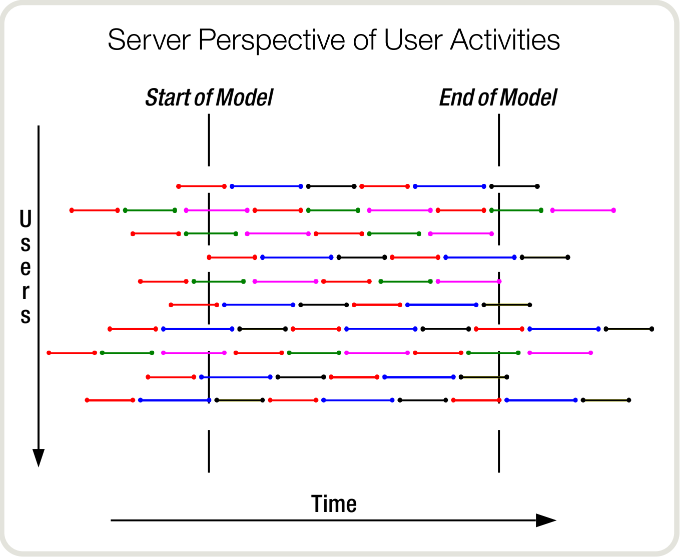


Figure 2: Server perspective of user activities

Reading the graph from top to bottom, left to right, notice user 1 surfs to page "red," then "blue," "black," "red," "blue" and "black." User 2 also starts with page "red," but then goes to "green," "purple," etc. Also take note that virtually any vertical slice of the graph between start and end times will reveal 10 users accessing the system, meaning this distribution is representative of 10 concurrent, or simultaneous, users. The server knows 10 activities are occurring at any moment in time, but not how many actual users are interacting with the system to generate those 10 activities.

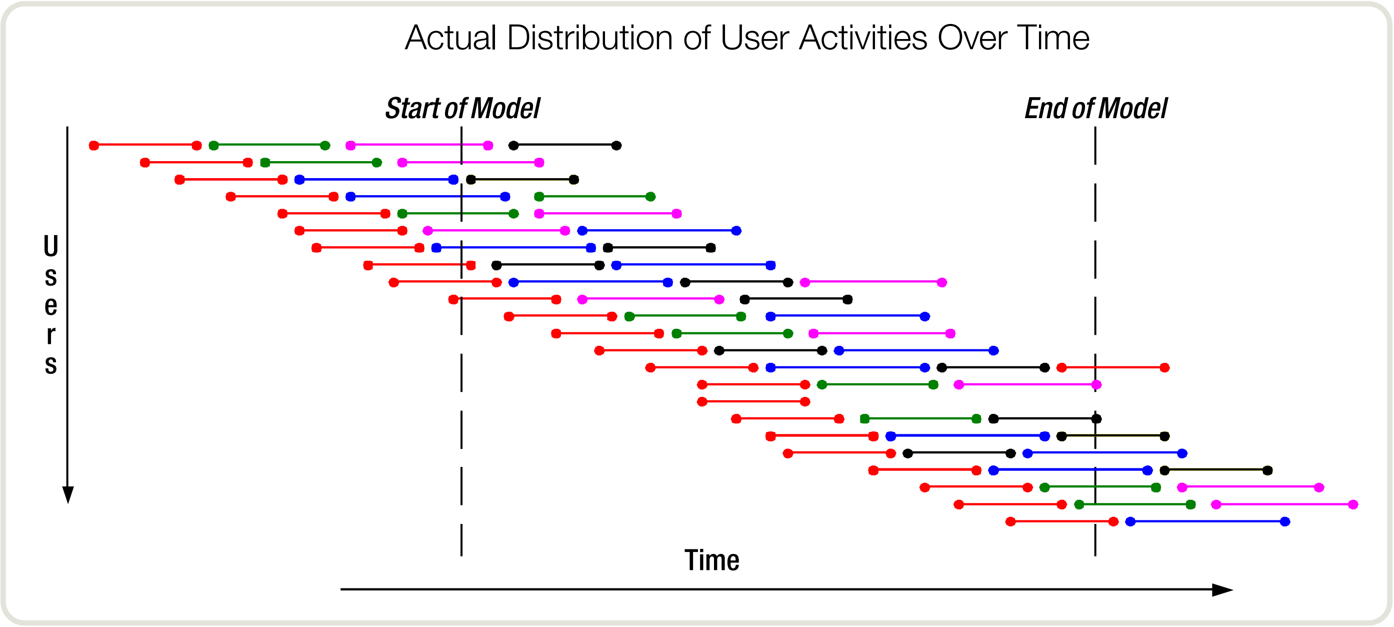


Figure 3: Actual distribution of user activities

Now look at a distribution of activities by individual user that would generate the server perspective graph in Figure 3.

In this graph, 23 individual users have been captured. These users conducted some activity during the time span modeled here. All 23 users began interacting with the site at different times. There is no particular pattern to the users' order of activities, except they all started with the "red" activity. These 23 users actually represent the exact same activities in the same sequence shown in Figure 2—demonstrating the difficulty in discussing concurrent users. Many individuals who use the term "concurrent users" aren't thinking from the server's perspective; they are thinking about the number of people at computers it would take to generate the load they have in mind. For this reason, I recommend simplifying the issue by expressing usage volume in terms of hourly users. Assuming the elapsed time between "Start of Model" and "End of Model" is one hour, the volume of the test could be expressed as either 10 concurrent users or 23 users per hour.

If we could overlay one of these graphs onto the other, we would see each activity is distributed identically over time. This is relevant because when we approach scalability from the perspective of number of hourly users, it tends to be a fairly straightforward and easily understandable task.

Calculating hourly usage

To establish a scalability goal for usage volume in terms of hourly users, first determine your expectations concerning:

- total number of unique users at the end of the first year
- distribution of users across the day/week/month
- length of time a user will interact with the application each time he or she accesses it
- number of times per day/week/month/year a single user will access the application.

For our online bookstore, let's assume:

- marketing predicts there will be 1,000,000 unique users during the first year
- access is evenly distributed throughout the month, but most users will typically access the site between 9 a.m. EST and 9 p.m. PST (15 hours) daily
- users will spend 15 minutes on the site each time they visit, on average
- similar sites report each user accesses the site once every other month on average.

Using that data:

Total monthly users—1,000,000 total users ÷ 2 (i.e., one access every other month) = 500,000 monthly users

Average daily users—500,000 total monthly users ÷ 30 days per month = 16,667 daily users

Average hourly users—16,667 average daily users ÷ 15 hours per day = 1,111 hourly users.

Considering the above numbers are averages and we didn't account for peak periods, I recommend setting a graceful scalability goal of 2,500 hourly users.

If pushed, we could convert hourly users to concurrent or simultaneous users by dividing 2,500 by 4 (because 1 hour = 4 x 15 minutes, the amount of time we assume an average user spends on the site), yielding a graceful scalability goal of 625 concurrent users.

Moving from scalability to capacity
From scalability goals, we derive capacity goals. Capacity goals focus not on system usage but on the system itself. For our online bookstore, we need to determine how much data regarding books, orders and user information the database will need to store to support those million users. In addition, we need to determine the associated hardware, software and configuration requirements for a database of that size; the network bandwidth our 2,500 hourly users will require; and the web server(s) throughput that will support that bandwidth. Because we are implementing a Service-Oriented Architecture (SOA), we will also have to determine how many credit cards we expect to process on a daily basis in order to negotiate the appropriate service level agreement with our credit card processing service provider.

Obviously, this is only a partial list of possible items in the capacity category. Every piece of supporting hardware and software should have some capacity goal associated with it—and all those goals can be derived from the scalability goal. Conducting this exercise early in the project will help to eliminate unwanted surprises later, when a target scalability load would serve only to show the database server is undersized or the available network bandwidth is insufficient.

Special notes on network capacity and latency
As it turns out, while network capacity and latency problems are extremely common, they're also the easiest capacity issues to detect and diagnose. In fact, in most cases, network capacity and related issues can be determined before the application is built or bought. The challenge is determining required network bandwidth, not the available network capacity. At this point, you can plug in scalability estimates.

In terms of network latency, a surprising number of my projects have been stymied because I found out very late in the process there was a multi-second latency between an external firewall and an application. After witnessing this scenario play out on several occasions, I learned to include network latency on my list of performance goals and requirements. Simply having a number ensures latency is determined before it becomes an unexpected production issue, regardless of the numbers chosen for goals or requirements.

Identify system and functional reliability concerns
Even though it is common for performance testing to reveal system and functional reliability defects under load conditions, it is rare when acceptance criteria or target areas for investigation are identified in advance. Typically, identifying reliability criteria and areas for investigation changes the overall performance testing strategy minimally. Identification does, however, increase the likelihood of finding issues by motivating teams to enhance monitoring and data collection during testing.

For our purposes, functional reliability is simply the system's ability to meet the same functional requirements under load conditions as it does in single-user situations. For example, if users are expected to log in, search for a book and subsequently purchase the book, they need to be able to complete these tasks with the same degree of accuracy, security and ease whether one user or many users are interacting with the system.

System reliability, on the other hand, is nearly a synonym for availability. The difference is, system reliability encompasses accuracy and consistency in addition to availability. This differs from functionality, which does not address service qualities such as proper and timely display of all expected search results.

All of this seems simple enough to identify from a business perspective, and it is. The complicated part is identifying which technical aspects of the system to monitor for indicators that reliability may be in question. While it is possible to detect the effect of many reliability issues simply by conducting manual or automated functional testing when the system is under load, exclusive use of this method is at best inefficient. It is significantly more valuable to identify technical areas that can provide early indicators of deeper problems. These technical areas will vary dramatically from system to system, but common areas to consider include:

Resource allocation/contention—For example, a certain amount of memory or a specific number of threads may be allocated to perform a specific task. These amounts or numbers frequently appear adequate under low user loads or for small to average volumes of data; however, they may become fully consumed before the system reaches its target load. If these areas aren't monitored, the symptom may be a server error or slowdown that is difficult to find. With monitoring, it's easy to see resource consumption rising and get a clear indicator of the actual issue prior to observing the symptom.

Item or object locking—Most commonly, this is a database-related problem, but not always. For instance, if an administrator is adding books to a database, and the database is configured to block access to the books table during an update, any user searching for a book will have to wait until the update is complete before the search will be permitted. This may be desired for data integrity reasons or it may be problematic, either because an end user thinks there are no results for his or her search or the search takes an unreasonably long time. These issues don't exist only under load, but a good performance test will include a wide mix of activities occurring at the same time, making it more likely locking issues will be detected.

Sessions or states—Many applications today track user sessions and/or the current state of the user. Under load, a system can sometimes confuse users with one another, causing data corruption or worse; drop sessions or states that force users to log in again and lose their place in the transaction; or even fail to time out sessions, claiming unnecessary resources and causing the system to act as if an even higher load is applied.

Network connections—In most cases, we expect our networks to work and have adequate available bandwidth. When networks begin to get overburdened, packet collisions increase, causing traffic to increase even further due to re-transmissions. Ultimately, users will experience slowdowns or a completely unavailable system until enough of them abandon attempts to interact with the system, reducing the load to acceptable levels. This is the simplest network concern. If your system uses RAS, VPN or other types of secure or less common connections, many other types of network connection issues could give you reliability headaches if they are not anticipated and monitored during performance testing.

Determine the geographic distribution of system users
In my experience, most performance-testing projects choose to evaluate users with various connection speeds, but accept the risk of not evaluating users from a variety of geographic regions. Anecdotal research suggests, in terms of risk, this pattern should be reversed.

Counter to what seems to be a common assumption, limited research and reports from highly respected performance testers indicate that dial-up and high-speed users rank the same sites and pages as fast, average, slow and unacceptable when evaluating sites on the connection speed to which they have become accustomed. As a result, users seem to spend the same total amount of time on a site regardless of connection speed, meaning users on slow connections view fewer pages than those with fast connections. This implies that sites expecting a notable dial-up audience need to be especially aware of the efficiency of navigation and transactions, which are usability issues rather than performance issues.

Geographic diversity, however, can have a significant impact on a user's satisfaction with performance. If your site has an international audience, but is solely hosted from a single server in a single location, users who happen to access your site from a location that is many hops from your server will experience increased response time due to network latency. While network latency is likely out of your control, it is important to determine the degree to which this phenomenon will affect your users. There are different approaches to this: One is to performance-test the site from various geographic locations; this approach may not be available to you and can be difficult to orchestrate. Another approach is to profile application behavior across the network, simulating the impact of varying latency and bandwidth constraints and measuring the characteristics of the transaction. This can be done locally by the performance-testing team, or better still, earlier in the application life cycle by the application architects and developers to ensure the application is suited to the network environment. Of course, if you determine this is a problem, the solution is generally to add mirror sites in various geographic regions rather than trying to improve performance to counteract network latency.

Examine unintended consequences
In almost every performance-testing project, a moment of panic occurs shortly after the application goes live, when the application's performance is not as expected. There are many possible reasons, ranging from poor testing to shockingly high usage, but most often it is the result of unintended consequences related to the production environment. We rarely get the chance to execute our performance tests in the actual production environment, or even in an accurate mirror of the production environment. Instead, we are often assured using low-power single PCs to represent virtual instances of high-power, multi-CPU servers will be "close enough." The truth is, it often isn't. Virtual instances, automatic updates or other operations, and different hubs, switches, routers, proxy servers and load balancers all have an impact on application performance. Virtual machines are generally well-isolated from one another, but not always as well-isolated as we'd like them to be. Simple things like scheduled back-up routines that aren't communicated in advance to developers and testers can result in serious contention issues if they overlap with late-night automatic batch jobs.

SOA environments can be even more complex. During testing, most service providers offer a separate service for users to plug into for testing and development. Once the new application kicks over to production, users plug into a new, untested service. Functionally, these services may be identical, but the production service's load is unlikely to match the load that the test and development service was under. In fact, the production service is unlikely to be housed in the same location or on the same hardware as the test and development service.

This means someone needs to be thinking about these things from day one, making a plan to mitigate the performance risks of putting a new or updated application into an existing production environment. The key to this kind of risk mitigation begins with a complete and accurate understanding of the target environment and all its complexities by the entire team. From there, the team can begin developing resource and performance "budgets" for the new application based on what is actually available for use in the production environment. Budgets should be monitored during performance testing, and flagged when they begin creeping toward their allowed value.

Of course, this is no substitute for finding a way to validate resource consumption, performance, configurations and assumptions by executing at least a couple of performance tests in the actual production environment. It will, at least, get people thinking about unintended consequences early—which is a major step in the right direction.

The bottom line

At the beginning of this paper, I made the point only one performance requirement really matters: making sure application users are not annoyed or frustrated by poor performance. Then I discussed various ways to convert that requirement into valuable and testable requirements, goals and objectives. I looked at the application from the end user's perspective but didn't forget the business objective. While parts of this article may not apply directly to your application or environment, the core message and general principles probably do.

If you remember nothing else from this paper, remember this: Performance-related requirements, goals and objectives may not be absolutely necessary for a skilled performance tester to add value to a project and application; however, giving a performance tester a set of well-defined, quantifiable requirements, goals and objectives that have not been derived from the application's business goals—and validated with feedback from real users—virtually ensures the end user is likely to be annoyed or frustrated by application performance even if the application achieves its requirements, goals and objectives. If your intent in conducting performance testing is to ensure application users are not frustrated by poor performance, you have to start by verbalizing the performance-related requirements, goals and objectives in subjective terms, quantifying those verbalizations where needed, and then designing performance tests to indicate the degree to which the requirements, goals and objectives have been achieved. To do any less is to say, through your actions, end-user satisfaction is not important for your application.

About the author
Scott Barber is Chief Technologist for PerfTestPlus, Executive Director of the Association for Software Testing and co-founder of the Workshop on Performance and Reliability. His particular specialties are testing and analyzing performance for complex systems, developing customized testing methodologies for individual organizations, testing embedded systems, teaching software testing, facilitating groups and authoring instructional materials. In addition, Barber is an international keynote speaker and contributor to various software testing publications. A member of ACM, IEEE, American MENSA and the Context-driven School of Software Testing, he is a signatory to the "Manifesto for Agile Software Development." For more information, e-mail Barber at sbarber@perftestplus.com or visit his web site (www.perftestplus.com).