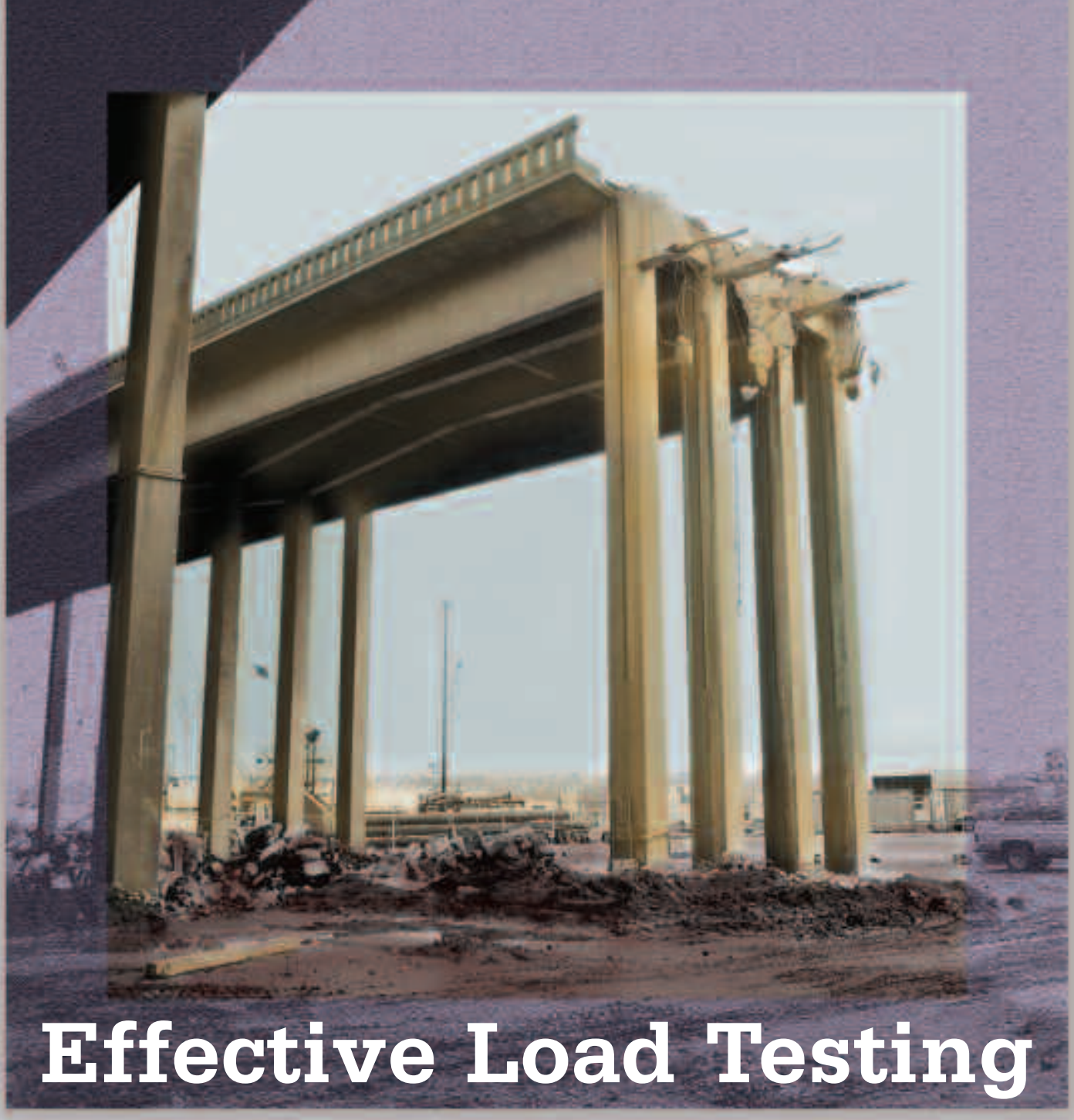

Effective Load Testing

To ensure that your scalable software system will be sturdy, stress test it and expose its hidden flaws.

By Alexander Podelko

Engineers who are learning about designing structures will attend a class on how to create a model of a proposed structure and determine how well it will hold up to various environmental factors. One of those factors may be weight, or load. Other factors may be wind, rain, or stress. When engineers write software, the same factors apply. However, the load or stress applied to software is not bricks, wind, or rain. It is the number of users.

Alexander Podelko is principal performance engineer at Hyperion Solutions in Stamford, Conn. He holds a Ph.D. in Computer Science from Gubkin University and an MBA from Bellevue University. He can be contacted at alexander_podelko@hyperion.com.

Unfortunately, theories don't guarantee a required level of performance for software. Much has been written about how to design scalable software, what best practices and design patterns to use, and even how to build models to predict performance. While that is important to creating scalable software, this still isn't an exact science, and testing multiuser applications under realistic as well as stress loads is really the only way to ensure appropriate performance and reliability in production.

Performance testing is emerging as an engineering discipline of its own,

based on "classic" functional testing from one side and system performance analysis and capacity planning from another side. The terminology is still vague in this field and the borders are fuzzy, but from a practical point of view, we probably can break down all testing into two key classes: those with a multi-user load (such as load, performance, stress, volume) and those without it (such as functional regression).

The different terms of each of these classes specify why we are testing and what result we are looking for, rather than what process is being used. There is no precise definition for each term; the exact meaning can differ from source to source. For example, performance testing could mean that we are most interested in response time; load testing could mean that we want to see the system's behavior under a specific load, and stress testing could mean that we want to find the system's breaking point. We still do the same things: apply a multi-user load and get some metrics. The difference is only in the details—what load we apply and what metrics are more important to us.

These two classes do not match the terms completely: We can do performance testing for one user, measuring response times with a stopwatch, or test the functionality of the system while having a 100-user load in the background. Moreover, performance is an element of functionality in some sense. We still refer to testing under a multi-user workload as load or performance testing and contrast it to classical single-user functional testing.

Both classes of testing have a lot in common. Still, load testing has significant specifics and requires some special approaches and skills. Quite often, applying the best practices and metrics of functional testing to load testing results in disappointments. Many things that are trivial to an experienced performance engineer can easily escape the attention of a person with less experience, which often results in unrealistic expectations, less than optimal test planning and design, and misleading results.

In this article I'll outline some issues to consider in performance testing and present the typical pitfalls from a practical point of view. The list is meant to contrast load testing with functional testing; the things common to the two are not discussed here. Although most of the recommendations are still valid for functional testing, they are much more important for performance testing. The selection is based on the extensive experience of Hyperion Performance Engineering Group and on extensive discussions with experts in load testing.

This article doesn't pretend to be comprehensive or to maintain a strict scientific approach; it is merely based on observations, mainly relating to the performance testing of distributed business applications. Some adjustments may be necessary for other classes of software.

What to Test

Since functional testing offers an unlimited number of possible test cases, the art of testing is to choose a limited set of test cases that will best check the product functionality given resource limitations. It is much worse with load testing: Each user can follow a different scenario (a sequence of functional steps), and even the sequence of steps of one user versus the steps of another user could affect results significantly.

Load testing can't be comprehensive. Several scenarios (use cases, test cases) should be chosen. Usually they are the most typical scenarios, the ones that most users are likely to follow. It is a good idea to identify several classes of users—for example, administrators, operators, users, and analysts. It is simpler to identify typical scenarios for any particular class of users. With that approach, rare use cases are ignored. For example, all administrator-type activities can be omitted, as there are few of them compared with other activities.

Another important criterion is risk. If a "rare" activity presents a major inherent risk, it can be a good idea to add it to the scenarios to test. For example, if database backups can significantly affect performance and should be done in parallel with regular work, it makes sense to include a backup scenario in performance testing.

Code coverage usually doesn't make much sense in load testing; it's important to know what parts of code are being processed in parallel by different users (almost impossible to track, of course), not that any particular piece of code was executed. Perhaps it's possible to speak about component coverage, making sure that all of the important components of the system are involved in performance testing. For example, if different components are responsible for printing HTML and PDF reports, it's a good idea to include both kinds of printing.

Other Requirements

In addition to functional requirements (which are still valid for performance testing—the system still should do everything it is designed to do under load), there are two other classes of requirements: response times (how fast the system can handle individual requests, or what a real user will experience) and throughput (how many requests the system can handle simultaneously).

Acceptable response times should be defined in each particular case. A response time of 30 minutes may be excellent for a big batch job, but it is absolutely unacceptable for accessing a Web page for an online store. Although it is often difficult to draw the line here, common sense is the key. Keep in mind that for multiuser testing, we get a range of response times for each transaction, so we need to use some aggregate values, such as

average or 90th percentile (90 percent of response times are less than the given value).

Throughput is a little trickier. Quite often the number of users is used to define load. Without defining what each user is doing and how intensely, the number of users doesn't make much sense as a measure of throughput. For example, if 500 users are running short queries each minute, we have a throughput of 30,000 queries per hour. However, if the same 500 users are running the same queries at a rate of one per hour, we have a throughput of 500 queries per hour. So with the same 500 users, you can have a sixty-fold difference between loads (and, in all likelihood, hardware requirements for the system).

The intensity of the load can be controlled by adding delays (often referred to as "think time") between actions in scripts or harness code. So one approach is to start with the total throughput the system should handle, then find the number of concurrent users, get the number of transaction per user for the test, and then try to set "think times" to ensure you'll have the proper number of transactions per user.

Finding the number of concurrent users for a new system can be tricky, too. Usually, information about real usage of similar systems can help you make an initial estimate. One widespread assumption for business applications, for example, is that 10 percent of named (registered in the system) users are active (logged) and 10 percent of these active users run concurrent requests (so 1,000 named users will mean 100 active users and 10 concurrent users). These numbers will depend greatly on the nature of the system.

Workload Implementation

If we work with a new system—that is, one against which we have never run a load test—the first question (after, of course, we know what to test) is how to create load. Are we going to generate it manually, use a load-testing tool, or will we create a test harness?

Manual testing will sometimes work if we want to simulate a small number of users, but even if it's well organized, manual testing will introduce some variation in each test, making our results less reproducible. Workload implementation using a tool (software or hardware) is quite straightforward when the system has a pure HTML interface, but even if there is so much as an applet on the client side, it can become a serious research task, not to mention requiring you to deal with proprietary protocols. Creating a test harness requires more knowledge about the system (for example, about an API) and some programming skills. Each choice requires different skills, resources, and investments.

In load testing, users can follow different scenarios, or sequences of steps, and this can affect test results significantly.

When starting a new load-testing project, the first thing to do is to decide how the workload will be implemented and to check that this way will really work. After we decide how to create the workload, we need to find a way to verify that the workload is actually being applied.

Workload Verification

Unfortunately, an absence of error messages during a load test does not mean that the system worked correctly. An important part of load testing is workload verification; we should be sure that the applied workload is doing what it is supposed to do and that all errors are caught and logged. Workload can be verified directly (analyzing server responses), or, in cases where this is impossible, indirectly (for example, analyzing the application log for the existence of particular entries). Many tools provide some way to verify workload and check errors, but a complete understanding of what exactly is happening is necessary.

For example, Mercury Interactive's LoadRunner reports only HTTP errors for Web scripts by default (such as "500 Internal Server Error"). If we rely on the default diagnostics, we could still believe that everything is going well when we're actually getting out-of-memory errors instead of the requested reports. To catch such errors, we should add special commands to check the content of HTML pages returned by the server to our script and enable these checks in the runtime options.

The Effect of Data

The size and structure of data can affect load test results. Using a small sample set of data for performance tests is an easy way to get misleading results. It is difficult to predict how much the data size will affect performance before real testing. The closer the test data is to production data, the better (although testing with larger data sets makes a lot of sense, to ensure that the system will work when more data has been accumulated).

Running multiple users hitting the same set of data (for example, playback of an automatically created script without proper modifications) is an easy way to get misleading results. This data could be completely cached, and we'll get much better results than in production. Or it could cause concurrency issues, and we'll get much worse results than in production. So scripts and test harnesses usually should be parameterized—fixed or recorded data should be replaced with values from a list of possible choices—so that each user uses a proper set of data. "Proper" here means that the data sets are different enough to avoid problems with caching and concurrency. The data sets should be specific for the system, the data and the test requirements.

Another easy trap with data is adding new data during the tests without sufficient consideration. Each new test will create additional data, so each test should be executed with a different amount of data. One way of running such tests successfully is to restore the system to the original state after each test or group of tests. Alternative-

ly, additional tests can be performed to prove that it may not matter in a particular case.

Exploring the System

At the beginning of a new project, it's good to run some tests first to figure out how the system behaves before creating formal plans. If no performance tests have been run, there's no way to predict how many users the system can support and how each scenario will affect overall performance.

It's good to check that we do not have any inherent functional problems: For example, is it possible to run all the requested scenarios manually? Are there any performance issues with just one or with several users? Are there enough computer resources to support the requested scenarios? If we find a functional or performance problem with one user, usually it should be fixed before starting performance testing with that scenario.

Even if there are big plans for performance testing, an iterative approach will fit better. As soon as a new script is ready, run it. This will help you understand how well the system can handle a specific load. The results you get can help you improve your testing plans and discover many issues early. By running tests, we are learning the system and may find out that the original ideas about the system were not entirely correct. A "waterfall" approach, with all scripts created before running any multiuser test, is dangerous here: We'll encounter issues later and may find out that a lot of work must be redone.

Unspecified Requirements

Usually, when people talk about performance testing they do not separate it from tuning, diagnostics or capacity planning. "Pure" performance testing is possible only in rare cases when the system and all optimal settings are well known. Usually some tuning activities are necessary at the beginning of the testing to be sure that the system is properly tuned and the results will be meaningful. In most cases, if a performance or reliability problem is found, it should be diagnosed further, until it becomes clear how to handle it. Generally speaking, performance testing, tuning, diagnostics and capacity planning are quite different processes, and not explicitly including any of them in the test plan will render the plan unrealistic from the beginning.

Time Considerations

Performance tests usually take more time than functional tests. Usually, we are interested in the steady mode during load testing. It means that all users need to log in and work for some time to be sure that we will see a stable pattern of performance and resource utilization. Measuring performance during transition periods can be misleading. The more users we simulate, the more time we will usually need to get into the steady mode. Moreover, some kinds of testing (reliability, for example) can require a significant amount of time, from several hours to several days or even weeks. Therefore, the number of tests that can be run per day is limited. It's especially important to consider this during tuning or diagnostics, when the number of iterations is unknown and can be large.

Simulating real users requires time, especially if it involves more than just repeating actions, such as entering orders, and perhaps some kind of process in which some actions will follow others. We can't just squeeze several days of regular work into 15 minutes for each user; this is far from a simulation of real work. Testing should involve a slice of work, not a squeeze.

In some cases we can make the load from each user more intensive, and respectively decrease the number of users to keep the total volume of work (that is, the throughput) the same. For example, you can simulate 100 users running a small report every five minutes instead of 300 users running that report every 15 minutes. In this case, we can speak about the ratio of simulated users to real users (1:3 for that example). This is especially useful when we need to perform a lot of tests during tuning of the system, or when we're trying to diagnose the problem to see the results of changes quickly. Quite often that approach is used when there are license limitations.

Still, "squeezing" should only be used in *addition* to full-scale simulation, not instead of it. Each user consumes additional resources for connections, threads, caches and so forth. The exact impact depends on the system implementation, so a simulation of 100 users running a small report every 10 minutes doesn't guarantee that the system will support 600 users running that report every hour.

Moreover, tuning for 600 users can differ significantly from tuning for 100 users. The higher the ratio between simulated and real users, the more you'll need to run all users to be sure that the system supports that number of users and is properly tuned.

What Affects the Process

Three specific characteristics of load testing affect the testing process and often require closer work with development to fix problems than functional testing does.

First, a reliability or performance problem often blocks further performance testing until the problem can be fixed or a workaround found.

Second, usually the full setup should be used to reproduce the problem. However, keeping the full setup at test for a long time can be expensive or even impossible.

Third, debugging performance problems is a sophisticated diagnostic process that usually requires close collaboration between a performance

engineer (running tests and analyzing the results) and a developer (profiling and altering code). Many tools, such as debuggers, work fine in a single-user environment, but do not work in the multiuser environment.

These three characteristics make it difficult to use an asynchronous process in load testing. An asynchronous process, most commonly used in functional testing, is one in which testers look for bugs and then log them into a defect tracking system. Afterward, development will prioritize the defects and fix each one. What is often required is the synchronized work of performance engineering and development to fix the problems and complete performance testing.

A Systematic Approach To Changes

The tuning (and often diagnostic) process consists of making changes in the system and evaluating their impact on performance (or the problems they uncover). It is important to take a systematic approach to these changes. This could be, for example, the traditional approach of "one change at a time" (also often referred to as "one factor at a time," or OFAT), or using design of experiments (DOE) theory. "One change at a time" here does not imply changing just one variable; it can mean changing several related variables to check a particular hypothesis.

The relationship between changes in the system parameters and changes in the product behavior is usually quite complex. Any assumption based on common sense can be wrong. A system's reaction can be quite the opposite of what's expected under heavy load, so changing several things at once without a systematic approach will not yield an understanding of how each change affects results. This could degrade the testing process and lead to incorrect conclusions. All changes and their effects should be logged to allow for rollback and further analysis.

Analyzing Results

The results of load testing are usually difficult to interpret as merely passed/failed. Even if we do not need to tune the system or diagnose a problem, we usually should consider not only transaction response times for all different transactions (usually using aggregating metrics such as average response times or 90th percentiles), but also other metrics such as resource utilization. Results analysis of load testing for enterprise-level systems can be quite difficult and should be based on good knowledge of the system and its performance requirements, and it should involve all possible sources of information: measured metrics, results of monitoring during the test, all available logs and profiling results (if available). For example, a heavy load on load-generator machines can completely skew results, and the only way to know that is to monitor those machines.

Results always vary in multiuser tests, due to minor differences in the test environment. If the difference is large, it is worth analyzing why and adjusting tests accordingly. For example, you can restart the program (or even reboot the system) before each test to eliminate caching effects. ☒