Agile Performance Testing

Flexibility and Iterative Processes Are Key to Keeping Tests on Track

Agile software development involves iterations, open collaboration and process adaptability throughout a project's life cycle. The same approaches are fully applicable to performance testing projects. So you need a plan, but it has to be malleable. It becomes an iterative process involving tuning and troubleshooting in close cooperation with developers, system administrators, database administrators and other experts.

By Alexander Podelko

You run a test and get a lot of information about the system. To be efficient you need to analyze the feedback you get, make modifications and adjust your plans if necessary. Let's say, for example, you plan to run 20 different tests, but after executing the first test you discover a bottleneck (for instance, the number of Web server threads). Unless you eliminate the bottleneck, there's no point running the other 19 tests if they all use the Web server. To identify the bottleneck, you may need to change the test scenario.

Even if the project scope is limited to preproduction performance testing, an agile, iterative approach helps you meet your goals faster and more efficiently, and learn more about the system along the way. After we prepare a test script (or generate workload some other way), we can run single- or multi-user tests, analyze results and sort out errors. The source of errors can vary–you can experience script errors, functional errors and errors caused directly by performance bottlenecks—and it doesn't make sense to add load until you determine the specifics. Even a single script allows you to locate many problems and tune the system at least partially. Running scripts separately also lets you see the amount of resources used by each type of load so you can build a system "model" accordingly (more on that later).

The word "agile" in this article doesn't refer to any specific development process or methodology; performance testing for agile development projects is a separate topic not covered in this paper. Rather, it's used as an application of the agile principles to performance engineering.

WHY THE 'WATERFALL' APPROACH DOESN'T WORK

The "waterfall" approach to software development is a sequential process in which development flows steadily downward (hence the name) through the stages of requirements analysis, design, implementation, testing, integration and maintenance. Performance testing typically includes these steps:

- Prepare the system.
- Develop requested scripts.
- Run scripts in the requested combinations.
- Compare results with the requirements provided.
- Allow some percentage of errors according to the requirements.
- Involve the development team if requirements are missed.

At first glance, the waterfall approach to performance testing appears to be a well-established, mature process. But there are many potential—and serious—problems. For example:

- The waterfall approach assumes that the entire system—or at least all the functional components involved—is ready for the performance test. This means the testing can't be done until very late in the development cycle, at which point even small fixes would be expensive. It's not feasible to perform such full-scope testing early in the development life cycle. Earlier performance testing requires a more agile, explorative process.
- The scripts used to create the system load for performance tests are themselves software. Record/playback load testing tools may give the tester the false impression that creating scripts is quick and easy, but correlation, parameterization, debugging and verification can be extremely challenging. Running a script for a single user that doesn't yield any errors doesn't prove much. I've seen large-scale corporate performance testing where none of the script executions made it through logon (single sign-on token wasn't correlated), yet performance testing was declared successful and the results were reported to management.
- Running all scripts simultaneously makes it difficult to tune and troubleshoot. It usually becomes a good illustration of the shot-in-the-dark antipattern—"the best efforts of a team attempting to correct a poorly performing application without the benefit of truly understanding *why* things are as they are" (http://www.kirk.blogcity.com/proposed_antipattern_shot_in_the_dark.htm). Or you need to go back and deconstruct tests to find exactly which part is causing the problems. Moreover, tuning and performance troubleshooting are iterative processes, which are difficult to place inside the "waterfall." And in most cases, you can't do them offline—you need to tune the system and fix the major problems before the results make sense.
- Running a single large test, or even sev-

Alexander Podelko *has specialized in performance engineering for 12 years. Currently he is a Consulting Member of Technical Staff at Oracle, responsible for performance testing and tuning of the Hyperion product family.*

eral large tests, provides minimal information about system behavior. It doesn't allow you to build any kind of model, formal or informal, or to identify any relationship between workload and system behavior. In most cases, the workload used in performance tests is only an educated guess, so you need to know how stable the system would be and how consistent the results would be if real workload varied.

Using the waterfall approach doesn't change the nature of performance testing; it just means you'll probably do a lot of extra work and end up back at the same point, performance tuning and troubleshooting, much later in the cycle. Not to mention that large tests involving multiple-use cases are usually a bad point to start performance tuning and troubleshooting, because symptoms you see may be a cumulative effect of multiple issues.

Using an agile, iterative approach doesn't mean redefining the software development process; rather, it means finding new opportunities inside existing processes to increase efficiency overall. In fact, most good performance engineers are already doing performance testing in an agile way but just presenting it as "waterfall" to management. In most cases, once you present and get management approval on a waterfall-like plan, you're free to do whatever's necessary to test the system properly inside the scheduled time frame and scope. If opportunities exist, performance engineering may be extended further, for example, to early performance checkpoints or even full software performance engineering.

> [Why wait until the entire system is assembled to start performance testing?]

TEST EARLY

Although I've never read or heard of anybody arguing against testing early, it rarely happens in practice. Usually there are some project-specific reasons—tight schedules or budgets, for instance—preventing such activities (if somebody thought about them at all).

Dr. Neil Gunther, in his book *Guerrilla Capacity Planning* (Springer, 2007), describes the reasons management (consciously or unconsciously) resists testing early. While Gunther's book presents a broader perspective on capacity planning, the methodology discussed, including the guerrilla approach, is highly applicable to performance engineering.

Gunther says there's a set of unspoken assumptions behind the resistance to performance-related activities. Some that are particularly relevant to performance engineering:

- The schedule is the main measure of success.
- Product production is more important than product performance.
- We build product first and then tune performance.
- Hardware is not expensive; we can just add more of it if necessary.
- There are plenty of commercial tools that can do it.

It may be best to accept that many project schedules don't allow sufficient time and resources for performance engineering activities and proceed in "guerrilla" fashion: Conduct performance tests that are less resource-intensive, even starting by asking just a few key questions and expanding as time and money permit.

The software performance engineering approach to development of software systems to meet performance requirements has long been advocated by Dr. Connie Smith and Dr. Lloyd Williams (see, for example, their book *Performance Solutions,* Addison-Wesley, 2001). While their methodology doesn't focus on testing initiatives, it can't be successfully implemented without some preliminary testing and data collection to determine both model inputs and parameters, and to validate model results. Whether you're considering a full-blown performance engineering or guerrilla-style approach, you still need to obtain baseline measurements on which to build your calculations. Early performance testing at any level of detail can be very valuable at this point.

A rarely discussed aspect of early performance testing is unit performance testing. The unit here may be any part of the system—a component, service or device. This is not a standard practice, but it should be. The later in the development cycle, the more costly and difficult it becomes to make changes, so why wait until the entire system is assembled to start performance testing? We don't wait in functional testing. The predeployment performance test is an analog of system or integration tests, but it's usually conducted without any "unit testing" of performance.

The main obstacle is that many systems are somewhat monolithic; the parts, or components, don't make much sense by themselves. But there may be significant advantages to test-driven development. If you can decompose the system into components in such a way that you may test them separately for performance, you'll only need to fix integration problems when you put the system together. Another problem is that many large corporations use a lot of third-party products in which the system appears as a "black box" that's not easily understood, making it tougher to test effectively.

During unit testing, variables such as load, security configuration and amount of data can be reviewed to determine their impact on performance. Most test cases are simpler and tests are shorter in unit performance testing. There are typically fewer tests with limited scope—for example, fewer variable combinations than in a full stress or performance test.

We shouldn't underestimate the power of the single-user performance test: If the system doesn't perform well for a single user, it certainly won't perform well for multiple users. Single-user testing is conducted throughout the application development life cycle, during functional testing and user acceptance testing, and gathering performance data can be extremely helpful during these stages. In fact, single-user performance tests may facilitate earlier detection of performance problems and indicate which business functions and application code need to be investigated further.

So while early performance engineering is definitely the best approach (at least for product development) and has long been advocated, it's still far from commonplace. The main problem here is that the mindset should change from a simplistic "record/playback" performance testing occurring late in the product life cycle to a more robust, true perform-

ance engineering approach starting early in the product life cycle. You need to translate "business functions" performed by the end user into component/unit-level usage, end-user requirements into component/unit-level requirements and so on. You need to go from the record/playback approach to using programming skills to generate the workload and create stubs to isolate the component from other parts of the system. You need to go from "black box" performance testing to "gray box."

If you're involved from the beginning of the project, a few guerrilla-style actions early on can save you (and the project) a lot of time and resources later. But if you're called in later, as is so often the case, you'll still need to do the best performance testing possible before the product goes live. The following sections discuss how to make the most of limited test time.

THE IMPORTANCE OF WORKLOAD GENERATION

The title of Andy Grove's book *Only the Paranoid Survive* may relate even better to performance engineers than to executives! It's hard to imagine a good performance engineer without this trait. When it comes to performance testing, it pays to be concerned about every part of the process, from test design to results reporting.

Be a performance test architect. The sets of issues discussed below require architect-level expertise.

1) Gathering and validating all requirements (workload definition, first and foremost), and projecting them onto the system architecture:

Too many testers consider all information they obtain from the business side (workload descriptions, scenarios, use cases, etc.) as the "holy scripture." But while businesspeople know the business, they rarely know much about performance engineering. So obtaining requirements is an iterative process, and every requirement submitted should be evaluated and, if possible, validated. Sometimes performance requirements are based on solid data; sometimes they're just a guess. It's important to know how reliable they are.

Scrutinize system load carefully as well. Workload is an input to testing, while response times are output. You may decide if response times are acceptable even after the test, but you must define workload beforehand.

The gathered requirements should be projected onto the system architecture because it's important to know if included test cases add value by testing different sets of functionality or different components of the system. It's also important to make sure we have test cases for every component (or, if we don't, to know why).

2) Making sure the system under test is configured properly and the results obtained may be used (or at least projected) for the production system:

Environment and setup considerations can have a dramatic effect. For instance:

- What data is used? Is it real production data, artificially generated data or just a few random records? Does the volume of data match the volume forecasted for production? If not, what's the difference?
- How are users defined? Do you have an account set with the proper security rights for each virtual user or do you plan to re-use a single administrator ID?
- What are the differences between the production and test environments? If your test system is just a subset of your production system, can you simulate the entire load or just a portion of that load? Is the hardware the same?

It's essential to get the test environment as close as possible to the production environment, but performance testing workload will never match production workload exactly. In "real life," the workload changes constantly, including user actions nobody could ever anticipate.

Indeed, performance testing isn't an exact science. It's a way to decrease risk, not to eliminate it. Results are only as meaningful as the test and environment you created. Performance testing typically involves limited functional coverage, and no emulation of unexpected events. Both the environment and the data are often scaled down. All of these factors confound the straightforward approach to performance testing, which states that we simply test X users simulating test cases A and B. This way, we leave aside a lot of questions: How many users can the system handle? What happens if we add other test cases? Do ratios of use cases matter? What if some administrative activities happen in parallel? All of these questions and more require some investigation.

Perhaps you even need to investigate the system before you start creating performance test plans. Performance engineers sometimes have system insights nobody else has; for example:

- Internal communication between client and server if recording used
- Timing of every transaction (which may be detailed to the point of specific requests and sets of parameters if needed)
- Resource consumption used by a specific transaction or a set of transactions

This information is additional input—often the original test design is based on incorrect assumptions and must be corrected based on the first results.

Be a script writer. Very few systems today are stateless systems with static content using plain HTML—the kind of systems that lend themselves to a simple "record/playback" approach. In most cases there are many obstacles to creating a proper workload. If it's the first time you see the system, there's absolutely no guarantee you can quickly record and play back scripts to create the workload, if at all.

Creating performance testing scripts and other objects is, in essence,

a software development project. Sometimes automatic script generation from recording is mistakenly interpreted as the entire process of script creation, but it's only the beginning. Automatic generation provides ready scripts in very simple cases, but in most nontrivial cases it's just a first step. You need to correlate and parametize scripts (*i.e.,* get dynamic variables from the server and use different data for different users).

After the script is created, it should be evaluated for a single user, multiple users and with different data. Don't assume the system works correctly just because the script was executed without errors. Workload validation is critical: We have to be sure the applied workload is doing what it's supposed to do and that all errors are caught and logged. This can be done directly, by analyzing server responses or, in cases where that's impossible, indirectly—for example, by analyzing the application log or database for the existence of particular entries.

> **Don't assume the system works correctly just because the script was executed without errors.**

Many tools provide some way to verify workload and check errors, but a complete understanding of what exactly is happening is necessary. For example, HP LoadRunner reports only HTTP errors for Web scripts by default (500 "Internal Server Error," for example). If we rely on the default diagnostics, we might still believe that everything is going well when we get "out of memory" errors instead of the requested reports. To catch such errors, we should add special commands to our script to check the content of HTML pages returned by the server.

When a script is parameterized, it's good to test it with all possible data. For example, if we use different users, a few of them might not be set up properly. If we use different departments, some could be mistyped or contain special symbols that must be properly encoded. These problems are easy to spot early on, when you're just debugging a particular script. But if you wait until the final, all-script tests, they muddy the entire picture and make it difficult to see the real problems.

My group specializes in performance testing of the Hyperion line of Oracle products, and we've found that a few scripting challenges exist for almost every product. Nothing exceptional—they're usually easily identified and resolved—but time after time we're called on to save problematic performance testing projects only to discover serious problems with scripts and scenarios that make test results meaningless. Even experienced testers stumble, but many problems could be avoided if more time were spent analyzing the situation.

Consider the following examples, which are typical challenges you can face with modern Web-based applications:

1) Some operations, like financial consolidation, can take a long time. The client starts the operation on the server, then waits for it to finish, as a progress bar shows on screen. When recorded, the script looks like (in LoadRunner pseudocode):

```
web_custom_request("XMLDataGrid.asp_7",
"URL={URL}/Data/XMLDataGrid.asp?Action=
EXECUTE&TaskID=1024&RowStart=1&ColStart=
2&RowEnd=1&ColEnd=2&SelType=0&Format=
JavaScript", LAST);

web_custom_request("XMLDataGrid.asp_8",
"URL={URL}/Data/XMLDataGrid.asp?Action=
GETCONSOLSTATUS",
LAST);

web_custom_request("XMLDataGrid.asp_9",
"URL={URL}/Data/XMLDataGrid.asp?Action=
GETCONSOLSTATUS",
LAST);

web_custom_request("XMLDataGrid.asp_9",
"URL={URL}/Data/XMLDataGrid.asp?Action=
GETCONSOLSTATUS",
LAST);
```

Each request's activity is defined by the *?Action=* part. The number of *GETCONSOLSTATUS* requests recorded depends on the processing time.

In the example above, the request was recorded three times, which means the consolidation was done by the moment the third *GETCONSOLSTATUS* request was sent to the server. If you play back this script, it will work this way: The script submits the consolidation in the *EXECUTE* request and then calls *GETCONSOLSTATUS* three times. If we have a timer around these requests, the response time will be almost instantaneous, while in reality the consolidation may take many minutes or even hours. If we have several iterations in the script, we'll submit several consolidations, which continue to work in the background, competing for the same data, while we report subsecond response times.

Consolidation scripts require creation of an explicit loop around GETCONSOLSTATUS to catch the end of the consolidation:

```
web_custom_request("XMLDataGrid.asp_7",
"URL={URL}/Data/XMLDataGrid.asp?Action=
EXECUTE&TaskID=1024&RowStart=1&ColStart=
2&RowEnd=1&ColEnd=2&SelType=0&Format=
JavaScript", LAST);

do {

sleep(3000);

web_reg_find("Text=1","SaveCount=abc_count",
LAST);


web_custom_request("XMLDataGrid.asp_8",
"URL={URL}/Data/XMLDataGrid.asp?Action=
GETCONSOLSTATUS", LAST);

        } while (str-
cmp(lr_eval_string("{abc_count}"),"1")==0);
```

Here, the loop simulates the internal logic of the system, sending *GETCONSOLSTATUS* requests every three seconds until the consolidation is complete. Without such a loop, the script just checks the status and finishes the iteration while the consolidation continues long after that.

2) Web forms are used to enter and save data. For example, one form can be used to submit all income-related data for a department for a month. Such a form would probably be a Web page with two drop-down lists (one for departments and one for months) on the top and a table to enter data underneath them. You choose a department and a month on the top of the form, then enter data for the specified department and month. If you leave the department and month in the script hardcoded as recorded, the script would be formally correct, but the test won't make sense at all—each virtual user will try to overwrite exactly the same data for the same department and the same month. To make it meaningful, the script should be parameterized to save data in different data intersections. For example, different departments may be used by each

user. To parameterize the script, we need not only department names but also department IDs (which are internal representations not visible to users that should be extracted from the metadata repository). Below is a sample of correct LoadRunner pseudocode (where values between { and } are parameters that may be generated from a file):

```
web_submit_data("WebFormGenerated.asp",
"Action=http://hfmtest.us.schp.com/HFM/data/WebFormGenerated.asp?FormName=Tax+QFP&caller=GlobalNav&iscontained=Yes",
        ITEMDATA,
        "Name=SubmitType", "Value=1",
ENDITEM,
        "Name=FormPOV", "Value=TaxQFP",
ENDITEM,
        "Name=FormPOV", "Value=2007",
ENDITEM,
        "Name=FormPOV", "Value=[Year]",
ENDITEM,
        "Name=FormPOV", "Value=Periodic",
ENDITEM,
        "Name=FormPOV", "Value=
{department_name}", ENDITEM,
        "Name=FormPOV", "Value=<Entity
Currency>", ENDITEM,
        "Name=FormPOV",
"Value=NET_INCOME_LEGAL", ENDITEM,
        "Name=FormPOV", "Value=[ICP Top]",
ENDITEM,

"Name=MODVAL_19.2007.50331648.1.
{department_id}.14.407.2130706432.4.1.90.0.345
", "Value=<1.7e+3>;;", ENDITEM,

"Name=MODVAL_19.2007.50331648.1.
{department_id}.14.409.2130706432.4.1.90.0.345
", "Value=<1.7e+2>;;", ENDITEM, LAST);
```

If department name is parameterized but department ID isn't, the script won't work properly. You won't get an error, but the information won't be saved. This is an example of a situation that never can happen in real life—users working through GUIs would choose department name from a drop-down box (so it always would be correct) and matching ID would be found automatically. Incorrect parameterization leads to sending impossible combinations of data to the server with unpredictable results. To validate this information, we would check what's saved after the test—if you see your data there, you know the script works.

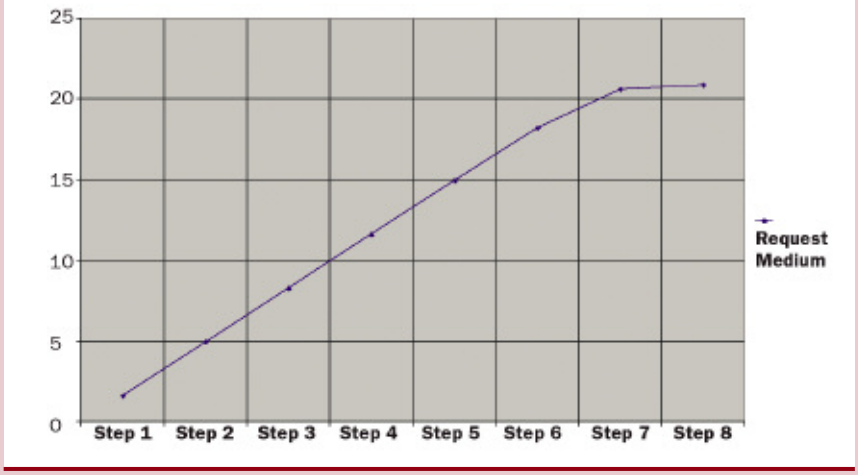
TUNE AND TROUBLESHOOT

Usually, when people talk about performance testing, they don't 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. Some tuning activities are typically necessary at the beginning of testing to be sure the system is tuned properly and the results will be meaningful. In most cases, if a performance problem is found, it should be diagnosed further, up to the point when it's clear how to handle it. Generally speaking, performance testing, tuning, diagnostics and capacity planning are quite different processes, and excluding any one of them from the test plan (if they're assumed) will make the test unrealistic from the beginning.

Both performance tuning and troubleshooting are iterative processes where you make the change, run the test, analyze the results and repeat the process based on the findings. The advantage of performance testing is that you apply the same synthetic load, so you can accurately quantify the impact of the change that was made. That makes it much simpler to find problems during performance testing than to wait until they happen in production, when workload is changing all the time. Still, even in the test environment, tuning and performance troubleshooting are quite sophisticated diagnostic processes usually requiring close collaboration between a performance engineer running tests and developers and/or system administrators making changes. In most cases, it's impossible to predict how many test iterations will be necessary. Sometimes it makes sense to create a shorter, simpler test still exposing the problem under investigation. Running a complex, "real-life" test on each tuning or troubleshooting iteration can make the whole process very long and the problem less evident because of different effects the problem may have on different workloads.

An asynchronous process to fixing defects, often used in functional testing—testers look for bugs and log them into a defect tracking system, then the defects are prioritized and fixed independently by the developers—doesn't work well for performance testing. First, a reliability or performance problem often blocks further performance testing until the problem is fixed or a workaround is found. Second, usually the full setup, which tends to be very sophisticated, should be used to reproduce the problem. Keeping the full setup for a long time can be expensive or even impossible. Third, debugging performance problems is a sophisticated diagnostic process usually requiring close collaboration between a performance engineer running tests and analyzing the results and a developer profiling and altering code. Special tools may be necessary; many tools, such as debuggers, work fine in a single-user environment but do not work in the multi-user environment, due to huge performance overheads. What's usually required is the synchronized work of performance engineering and development to fix the problems and complete performance testing.

FIG. 1: THROUGHPUT

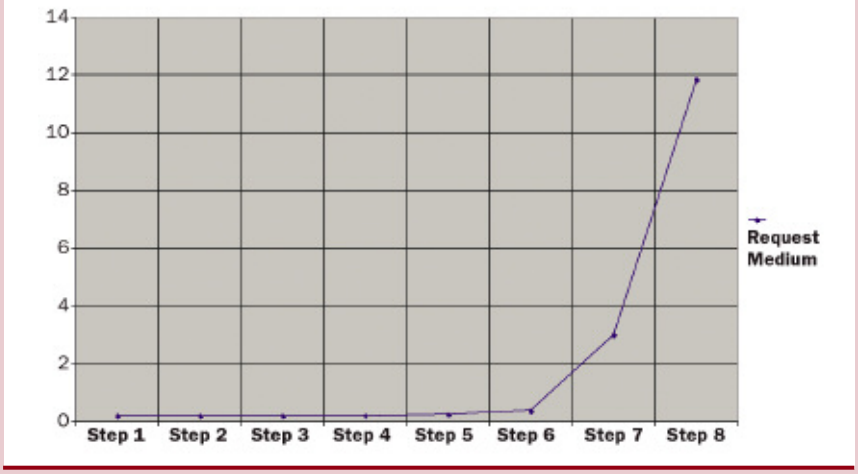


BUILD A MODEL

Creating a model of the system under test significantly increases the value of performance testing. First, it's one more way to validate test correctness and help to identify system problems—deviations from expected behavior might signal issues with the system or with the way you create workload. Second, it allows you to answer questions about the sizing and capacity planning of the system.

Most performance testing doesn't require a formal model created by a sophisticated modeling tool—it may

FIG. 2: RESPONSE TIME



involve just simple observations of the amount of resources used by each system component for the specific workload. For example, workload A creates significant CPU usage on server X while server Y is hardly touched. This means that if you increase workload A, the lack of CPU resources on server X will create a bottleneck. As you run increasingly complex tests, you verify results you get against your "model"—your understanding of how the system behaves. If they don't match, you need to figure out what's wrong.

Modeling often is associated with queuing theory and other sophisticated mathematical constructs. While queuing theory is a great mechanism to build sophisticated computer system models, it's not required in simple cases. Most good performance engineers and analysts build their models subconsciously, without even using such words or any formal efforts. While they don't describe or document their models in any way, they take note of unusual system behavior—*i.e.*, when system behavior doesn't match the model—and can make some simple predictions ("It looks like we'll need X additional resources to handle X users," for example).

The best way to understand the system is to run independent tests for each business function to generate a workload resource usage profile. The load should not be too light (so resource usage will be steady and won't be distorted by noise) or too heavy (so it won't be distorted by nonlinear effects).

Considering the working range of processor usage, linear models often can be used instead of queuing models for the modern multiprocessor machines (less so for single-processor machines). If there are no bottlenecks, throughput (the number of requests per unit of time), as well as processor usage, should increase proportionally to the workload (for example, the number of users) while response time should grow insignificantly. If we don't see this, it means there's a bottleneck somewhere in the system and we need to discover where it is.

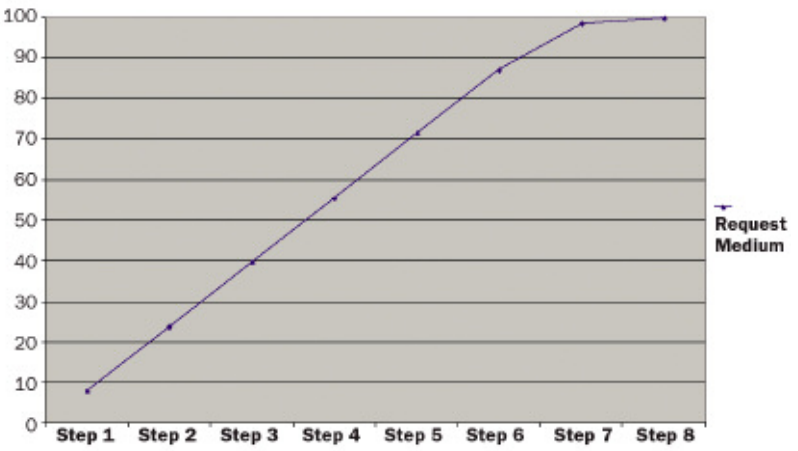
For example, let's look at a simple queuing model I built using TeamQuest's modeling tool for a specific workload executing on a four-way server. It was simulated at eight different load levels (step 1 through step 8, where step 1 represents a 100-user workload and 200 users are added for each step thereafter, so that step 8 represents 1,500 users). Figures 1 through 3 show throughput, response time and CPU usage from the modeling effort.

An analysis of the queuing model results shows that the linear model accurately matches the queuing model through step 6, where the system CPU usage is 87 percent. Most IT shops don't want systems loaded more than 70 percent to 80 percent.

That doesn't mean we need to discard queuing theory and sophisticated modeling tools; we need them when systems are more complex or where more detailed analysis is required. But in the middle of a short-term performance engineering project, it may be better to build a simple, back-of-the-envelope type of model to see if the system behaves as expected.

Running all scripts simultaneously makes it difficult to build a model. While you still can make some predictions for scaling the overall workload proportionally, it won't be easy to find out where the problem is if something doesn't behave as expected. The value of modeling increases drastically when your test environment differs from the production environment. In that case, it's important to document how the model projects testing results onto the production system.

FIG. 3: CPU USAGE



THE PAYOFF

The ultimate goal of applying agile principles to software performance engineering is to improve efficiency, ensuring better results under tight deadlines and budgets. And indeed, one of the tenets of the "Manifesto for Agile Software Development" (http://agilemanifesto.org/) is that responsiveness to change should take priority over following a plan. With this in mind, we can take performance testing of today's increasingly complex software to new levels that will pay off not just for testers and engineers but for all stakeholders.