# Multiple Dimensions of Load Testing

Alexander Podelko
Oracle

*Load testing is an important part of the performance engineering process. It remains the main way to ensure appropriate performance and reliability in production. It is important to see a bigger picture beyond stereotypical, last-moment load testing. There are multiple dimensions of load testing: environment, load generation, testing approach, life-cycle integration, feedback and analysis. This paper discusses these dimensions and how load testing tools support them.*

## *What is Load Testing?*

Let us first define load testing as the terminology is rather vague here [STIR02, MOLY14, PERF07]. The term is used here for everything requiring the application of a multi-user, synthetic load. Many different names may be used for such multi-user testing, such as performance, concurrency, stress, scalability, endurance, longevity, soak, stability, or reliability testing. There are different (and sometimes conflicting) definitions of these terms.

While each kind of performance testing may have different goals and test designs, in most cases they use the same approach: applying multi-user, synthetic workloads to the system. The term "load testing" is used here to better contrast multi-user testing with other performance engineering methods such as single-user performance testing which doesn’t require a load testing tool.

The taxonomy, single-user vs. multi-user performance, may look trivial – but often is not well understood. Both are performance assessment techniques and they share certain areas of common knowledge, but otherwise they actually are different disciplines which require different knowledge, skills, and tools. Single-user performance is performance only, when you may trace down where problems are and how much time you may save fixing them. A kind of ideal scalability is assumed. Profiling or Web Performance Optimization (WPO) falls, for example, into that category. While you need to have a good understanding of the performance aspects of your environment and be able to use appropriate tools such as a profiler, usually you don’t need a lot of performance engineering knowledge or skills and often such tasks are performed by software developers.

With multi-user performance we are getting to the intersection of performance and scalability involving more sophisticated problems often requiring special knowledge, tools, and skills. Load testing remains the main proactive way to assure multi-user performance and usually requires the support of performance testers / engineers / architects. But the area is transforming now – and it is important to understand what options we have here and how load testing may be adjusted to today's industry needs.

## *Five Dimensions to Think About*

The traditional (and stereotypical) way of testing performance was last-minute, pre-production performance validation in a lab with a load testing tool using recording/playback. A typical load testing process (minor variations of which may be found in numerous papers and books) is shown in Figure 1.

However, it shows rather a narrow view representing the traditional approach: several important parts are missed or underrepresented on this picture (some existed in a simplified form; some were introduced with latest industry trends). I will highlight five such elements and name them ‘dimensions’ of load testing to highlight the fact that every one of them provides a continuum of choices you need to make, somewhat independent from other dimensions. The actual load testing is a combination of all these choices.

This approach is used just to present a variety of the options we have and a number of decision we should make instead of blindly following the stereotypical approach. The specific dimensions and their overall number here are just ones that, by author's opinion, often do not getting enough consideration. It is not a complete list and is definitely open for further elaboration.

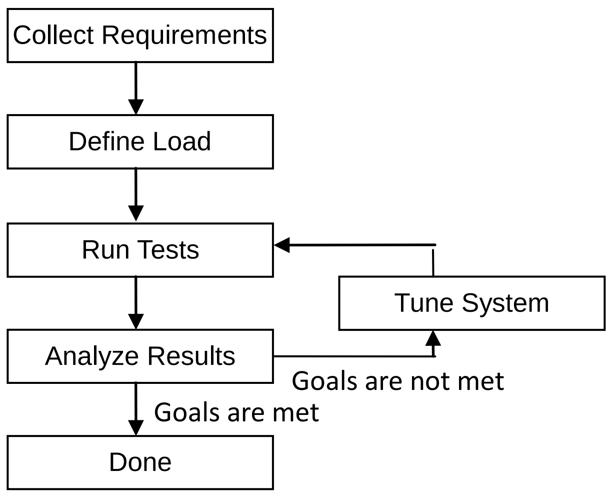

Figure 1 – Load Testing Process

In particular, we see that the following aspects are not well represented on the picture:

- **Environment.** It basically assumes that system and load generators are given. But setting environment properly is one of main challenges – and we have many options here. With cloud and cloud services, you have even more options.

- **Load Generation.** After the load is logically defined (sometimes referred to as workload characterization or workload modeling), we need to convert it into something that will physically create that load during the "run tests" step. While often it is considered as a technical and rather straightforward step – it may be a challenge in non-trivial cases.

- **Testing Approach.** Just applying the workload as defined limits the benefits of load testing – and may be not an option early in the project, during agile development or in DevOps environments. Agile and exploratory approaches may greatly increase the possible benefits of load testing.

- **Life-Cycle Integration.** In more and more cases, load testing is not just an independent step of software development life-cycle where you get the system at some point before release. In agile development or DevOps environments, it should be interwoven with the whole development process – and there is no easy answer here.

- **Feedback and Analysis.** You need to decide what information you collect (and how) and how to analyze it. Just comparing with your goals is not enough in most cases – especially in cases where the system under test doesn't match the production system exactly or when the test you are running is not the full projected load.

Let's consider these dimensions one by one to understand the issues and considerations related to each. We still should understand that these dimensions are not completely independent, but rather just different aspects of load testing process – so some decisions may impact several dimensions. But from the didactical point of view, it still makes sense to look into each of them separately.

## *Environment*

With the appearance of cloud and cloud services, we increased the environmental configuration options for both the system under test and the load generators.

We still have the challenge of making the system under test as close to the production environment as possible (in all aspects – hardware, software, data, configuration). One interesting new trend is testing in production. Not that it is something new by itself; what is new is that it is advocated as a preferable and safe (if done properly) way of doing performance testing. As systems become so huge and complex, so that it is extremely difficult to reproduce them in a test setup, people are more ready to accept the issues and risks related to using the production site for testing.

If we create a test system, the main challenge is to make it as close to the production system as possible. In case we can't replicate it, the issue would be to project results to the production environment. And while it is still an option – there are mathematical models that will allow making such a projection in most cases – but the further away is the test system from the production system, the more risky and less reliable would be such projections.

There were many discussions about different deployment models. Options include traditional internal (and external) labs;  cloud as Infrastructure as a Service (IaaS), when some parts of the system or everything are deployed there; and service, cloud as Software as a Service (SaaS), when vendors provide load testing service. There are some advantages and disadvantage of each model. Depending on the specific goals and the systems to test, one deployment model may be preferred over another.

For example, to see the effect of performance improvement (performance optimization), using an isolated lab environment may be a better option to see even small variations introduced by a change. To load test the whole production environment end-to-end just to make sure that the system will handle the load without any major issue, testing from the cloud or a service may be more appropriate. To create a production-like test environment without going bankrupt, moving everything to the cloud for periodical performance testing may be a solution.

For comprehensive performance testing, we may need both lab testing (with reproducible results for performance optimization) and distributed, realistic outside testing (to check real-life issues you can't simulate in the lab). Doing both can be expensive and makes sense only when performance is critical and the system is global.  Even if your systems aren't global as yet, they may end up there eventually. If there is a chance of facing these issues, it would be better to have a tool supporting different deployment models.

The scale also may be a serious consideration. When you have only a few users to simulate, it is usually not a problem. The more users you need to simulate, the more important the right tool becomes. Tools differ drastically on how many resources they need per simulated user and how well they handle large volumes of information. This may differ significantly even for the same tool, depending on the protocol used and the specifics of your script. As soon as you need to simulate thousands of users, it may become a major problem. For a very large number of users, some automation – like automatic creation of a specified number of load generators across several clouds – may be very handy. Cloud services may be another option here.

## *Load Generation*

The "define load" step (sometimes referred to as workload characterization or workload modeling) is a logical description of the load to be applied (e.g. users log in, navigate to a random item in the catalog, add it to the shopping cart, pay, and logout with an average of ten seconds of think time between actions). But in load testing you need another step – we may name it "create test assets" - an implementation of this workload and the conversion of the logical description into something that will physically create that load during the "run tests" step. Manual testing may still be an option in a few cases (when load is low and repeatability is not needed) - then it can be just the description given to each tester. But in all other load testing cases, it should be a program or a script. We need these tests assets to generate load.

While load generation is rather a technical issue, it is the basis for load testing – you can't proceed until you figure out a way to generate load. As a technical issue, it depends heavily on the tools and functionality supported.

There are three main approaches to workload generation [PODE12] and every tool may be evaluated on which of them it supports and how.

### Protocol-level recording/playback

**T**his is the mainstream approach to load testing: recording communication between two tiers of the system and playing back the automatically created script (usually, of course, after proper correlation and parameterization). As far as no client-side activities are involved, it allows the simulation of a large number of users. Such tool can only be used if it supports the specific protocol used for communication between two tiers of the system.

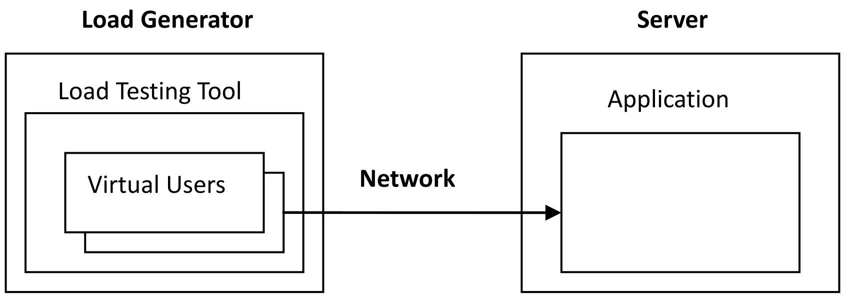

Fig.2 Record and playback approach, protocol level

With quick internet growth and the popularity of browser-based clients, most products support only HTTP or a few select web-related protocols. To the author's knowledge, only HP LoadRunner and Borland SilkPerformer try to keep up with support for all popular protocols (other products claiming support of different protocols usually use only UI-level recording/playback, described below). Therefore, if you need to record a special protocol, you will probably end up looking at these two tools (unless you find a special niche tool supporting your specific protocol). This somewhat explains the popularity of LoadRunner at large corporations because they usually using many different protocols. The level of support for specific protocols differs significantly, too. Some HTTP-based protocols are extremely difficult to correlate if there is no built-in support, so it is recommended that you look for that kind of specific support if such technologies are used. For example, Oracle Application Testing Suite may have better support of Oracle technologies (especially new ones such as Oracle Application Development Framework, ADF).

Quite often the whole area of load testing is reduced to pre-production testing using protocol-level recording/playback [BUKSH12]. While it was (and still is) the mainstream approach to testing applications, it is definitely just one type of load testing using only one type of load generation; such equivalency is a serious conceptual mistake, dwarfing load testing and undermining performance engineering in general [SMITH02].

**UI-level recording/playback**

This option has been available for a long time, but it is much more viable now. For example, it was possible to use Mercury/HP WinRunner or QuickTest Professional (QTP) scripts in load tests, but a separate machine was needed for each virtual user (or at least a separate terminal session). This drastically limited the load level that could be achieved. Other known options were, for example, Citrix and Remote Desktop Protocol (RDP) protocols in LoadRunner – which always were the last resort when nothing else was working, but were notoriously tricky to play back [PERF].

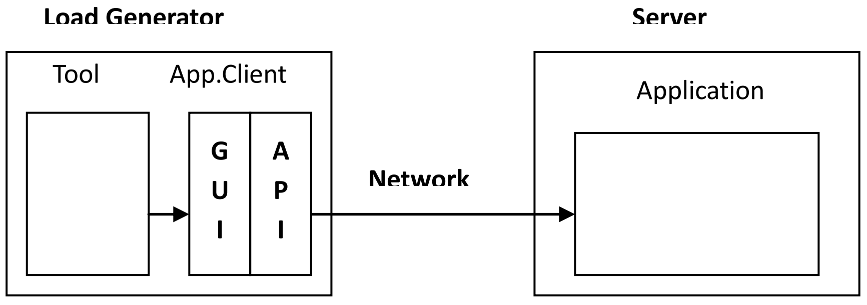

Fig.3 Record and playback approach, GUI users

New UI-level tools for browsers, such as Selenium, have extended the possibilities of the UI-level approach, allowing running of multiple browsers per machine (limiting scalability only to the resources available to run browsers). Moreover, UI-less browsers, such as HtmlUnit or PhantomJS, require significantly fewer resources than real browsers.

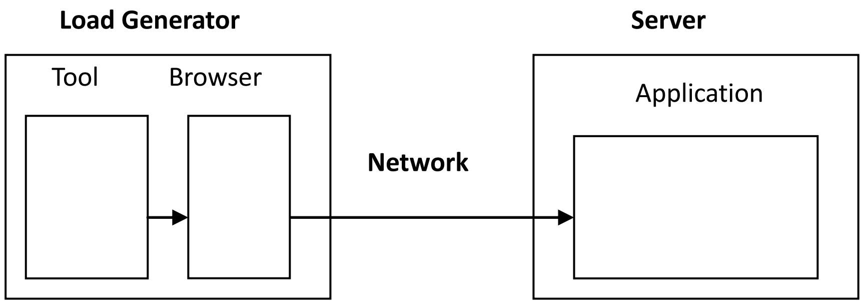

Fig.4 Record and playback approach, browser users

Today there are multiple tools supporting this approach, such as Appvance, which directly harnesses Selenium and HtmlUnit for load testing; or LoadRunner TruClient protocol and SOASTA CloudTest, which use proprietary solutions to achieve low-overhead playback. Nevertheless, questions of supported technologies, scalability, and timing accuracy remain largely undocumented, so the approach requires evaluation in every specific case.

**Programming**

There are cases when recording can't be used at all, or when it can, but with great difficulty. In such cases, API calls from the script may be an option. Often it is the only option for component performance testing. Other variations of this approach are web services scripting or use of unit testing scripts for load testing. And, of course, there is a need to sequence and parameterize your API calls to represent a meaningful workload. The script is created in whatever way is appropriate and then either a test harness is created or a load testing tool is used to execute scripts, coordinate their executions, and report and analyze results.

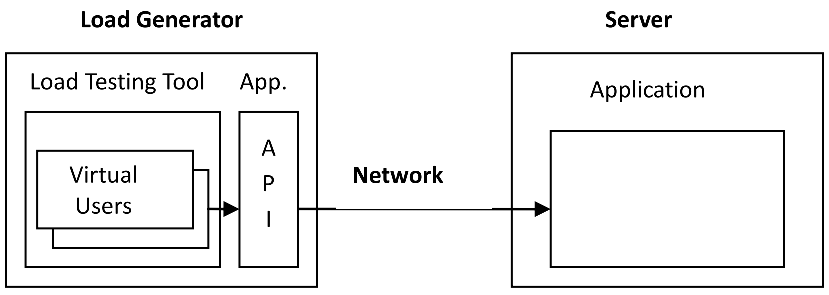

Fig 5. Programming API using a Load Testing Tool.

To do this, the tool should have the ability to add code to (or invoke code from) your script. And, of course, if the tool's language is different from the language of your API, you would need to figure out a way to plumb them. Tools, using standard languages such as C (e.g. LoadRunner) or Java (e.g. Oracle Application Testing Suite) may have an advantage here. However, you need to understand all of the details of the communication between client and server to use the right sequences of API calls; this is often the challenge.

**Special cases**

There are special cases which should be evaluated separately, even if they use the same generic approaches as listed above. The most prominent special case is mobile technologies. While the existing approaches remain basically the same, there are many details on how these approaches get implemented that need special attention. The level of support for mobile technologies differs drastically: from the very basic ability to record HTTP traffic from a mobile device and play it back against the server up to end-to-end testing for native mobile applications and providing a "device cloud".

## *Testing Approach*

Many people associate performance testing with the traditional corporate approach to load testing - last-minute (just before release, when everything is already implemented and functionally tested) performance validation according to given scenarios and requirements. The concept of exploratory performance testing, for example, is still rather alien.

But the notion of exploring is much more important for performance testing than for functional testing. Functionality of systems is usually more or less defined (whether it is well documented is a separate question) and testing boils down to validating if it works properly. In performance testing, you won't have a clue how the system would behave until you try it. Having requirements – which in most cases are goals you want your system to meet – doesn't help you much here because actual system behavior may be not even close to them. It is rather a performance engineering process (with tuning, optimization, troubleshooting and fixing multi-user issues) eventually bringing the system to the proper state than just testing.

If we have the testing approach dimension, the opposite of exploratory would be regression testing. We want to make sure that we have no regressions as we modify the product – and we want to make it quick and, if possible, automatic. And as soon as we get to an iterative development process where we have product changing all the time - we need to verify that there is no regression all the time. It is a very important part of the continuum without which your testing doesn't quite work. You will be missing regressions again and again going though the agony of tracing them down in real time. Automated regression testing becomes a must as soon as we get to iterative development where we need to test each iteration.

So we have a continuum from regression testing to exploratory testing, with traditional load testing being just a dot on that dimension somewhere in the middle. Which approach to use (or, more exactly, which combination of approaches to use) depends on the system. When the system is completely new, it would be mainly exploratory testing. If the system is well known and you need to test it again and again for each minor change – it would be regression testing and here is where you can benefit from automation (which can be complemented by exploratory testing of new functional areas – later added to the regression suite as their behavior become well understood).

If we see the continuum this way, the question which kind of testing is better looks completely meaningless. You need to use the right combination of approaches for your system in order to achieve better results. Seeing the whole testing continuum between regression and exploratory testing should help in understanding what should be done.

Of course, after deciding what to do on a conceptual level, there are numerous technical details to be decided so that you can implement tests properly. In more sophisticated cases, for example, coordination between virtual users is needed, such as synchronization points, data exchange during execution, or sophisticated scheduling.

Accuracy of user simulations is very important too. The test results may vary depending on specific browser and browser's settings, protocol used, network connection details, etc. Details of implementation of browser's cache and threading may have a major impact. Sophisticated tools allow creation of very complicated test scenarios and configure each group of users in the way required.

Other important aspects are, for example, network simulations (simulating different network conditions for different groups of users) or IP spoofing ( where every virtual user pretends to have its own IP address, which may be important for proper work of load balancers) allowing to create a more realistic load.

## *Life-Cycle Integration*

In more and more cases, it should not be just an independent step of the software development life-cycle when you get the system at some point before release. In agile development / DevOps environments it should be interwoven with the whole development process. There are no easy answers here that fit all situations. While agile development / DevOps become mainstream nowadays, their integration with performance testing is just making first steps.

Integration support becomes increasingly important as we start to talk about continuous integration (CI) and agile methodologies. Until recently, while there were some vendors claiming their load testing tools better fit agile processes, it usually meant that the tool is a little easier to handle (and, unfortunately, often just because there is not much functionality offered).

What makes agile projects really different is the need to run large number of tests repeatedly, resulting in the need for tools to support performance testing automation. Unfortunately, even if a tool has something that may be used for automation, like starting with a command line and parameters, it may be difficult to discover. If continuous integration is on the horizon (to whatever degree), it is important to understand what the tool provides to support CI.

The situation started to change recently as agile support became the main theme in load testing tools [LOAD14]. Several tools recently announced integration with Continuous Integration Servers (such as Jenkins or Hudson). While initial integration may be minimal, it is definitively an important step toward real automation support.

It doesn't looks like we may have standard solutions here, as agile and DevOps approaches differ significantly and proper integration of performance testing can't be done without considering such factors as development and deployment process, system, workload, ability to automate and automatically analyze results. You would probably need a combination of shorter automated tests inside CI with periodic longer tests outside or, maybe, in parallel to critical CI path.

The continuum here would be from old traditional load testing (which basically means no real integration: it is a step in the project schedule to be started as soon as system would be ready, but otherwise it is executed separately as a sub-project) to full integration into CI when tests are run and analyzed automatically for every change in the system.

While already mentioned above, cloud integration and support of new technologies are important for integration. Cloud integration, including automated deployment to public clouds (almost all major load testing tools), private cloud automation (Oracle Testing as a Service - TaaS), and cloud services simplify deployment automation. Support of newest technologies used (such as WebSocket or SPDY by Neoload or ADF in Oracle Load Testing) eliminates time-consuming and error-prone manual scripting and streamlines automation.

## *Feedback and Analysis*

Another dimension is getting and analyzing feedback. As already mentioned, performance testing is rather a performance engineering process (with tuning, optimization, troubleshooting and fixing multi-user issues) eventually bringing the system to the proper state rather than just testing. And the main feedback you get during your testing (in addition to system's configuration and load applied, which you know) is the results of monitoring your system (such as response times, errors, and resources your system consumes).

While, of course, it always was an important part of load testing and represents the feedback loop on the fig. 1, its role there was often limited to comparing results against goals. It is not enough anymore in most cases. Even the cases when the system under test doesn't match the production system exactly or when the test you are running is not the full projected load require a much more sophisticated analysis. It becomes even a bigger challenge when such tests would be a part of CI, where analysis should be done automatically.

As we get to discussing load testing tools, environmental monitoring and results analysis are two very important sets of functionality. While it is theoretically possible to do them both using separate tools, it significantly degrades productivity and may require building some plumbing infrastructure. So while these two areas are sometimes presented as optional for load testing tools, integrated and powerful monitoring and results analysis are both very important. The more complex the system measured and the tests performed, the more important feedback and analysis become. A possibility to analyze monitoring results and test results together helps a lot.

Talking about integrated monitoring, it is important to understand what kind of information is available and what mechanisms are behind it. While for Windows there is usually the Performance Monitor behind the scene, for (L)UNIXes it may differ a lot. Getting information from application (via, for example, JMX) and database servers is important. Many tools recently announced integration with Application Performance Management / Monitoring (APM) tools, such as AppDynamics, New Relics, or Compuware Dynatrace. If using such tools is an option, it definitely opens new opportunities to see what is going on inside the system under load and what needs to be optimized. One thing to keep in mind is that older APM tools and profilers may be not appropriate to use under load due to the high overheads they introduce.

## *Summary*

There are many more ways of doing performance testing than just the old, stereotypical approach of last-minute pre-production performance validation. You actually have several almost independent dimensions of load testing – and for every one you may select one or multiple points of how to exactly address that side of load testing. The combination of all choices will define what you are doing.

The industry is rapidly changing (just think cloud, agile, CI, DevOps, etc.) – and to keep up performance testing should be changing too. It is not that we just need to find a new recipe – it looks like we would never get to that point again - we need to adjust on the fly to every specific situation to remain relevant.

Good tools can help you here – and not so good tools may limit you in what you can do. And having so many dimensions / options, we can't just rank tools on a simple better/worse scale. It may be the case that a simple tool will work quite well in a particular situation. A tool may be very good in one situation and completely useless in another. The value of the tool is not absolute; rather it is relative to *your* situation.

## References

[BUKSH12] J. Buksh. Performance Testing is hitting the wall. (2012). http://www.perftesting.co.uk/performance-testing-is-hitting-the-wall/2012/04/11/

[LOAD14] Load Testing at the Speed of Agile. A Neotys White Paper. (2014). http://www.neotys.com/documents/whitepapers/whitepaper_agile_load_testing_en.pdf

[MOLY14] I. Molyneaux. The Art of Application Performance Testing. O'Reilly (2014).

[OPEN] Open Source Performance Testing Tools. http://www.opensourcetesting.org/performance.php

[PERF] Performance Testing Citrix Applications Using LoadRunner: Citrix Virtual User Best Practices, Northway white paper. http://northwaysolutions.com/our-work/downloads

[PERF07] Performance Testing Guidance for Web Applications. (2007). http://perftestingguide.codeplex.com/

[PODE12] A.Podelko. Load Testing: See a Bigger Picture. CMG (2012).

[PODE14] A.Podelko. Adjusting Performance Testing to the World of Agile. CMG (2014).

[SEGUE05] Choosing a Load Testing Strategy, Segue white paper (2005). http://bento.cdn.pbs.org/hostedbento-qa/filer_public/smoke/load_testing.pdf

[SMITH02] C.U. Smith, L.G.Williams, “Performance Solutions”, Addison-Wesley (2002).

[STIR02] S.Stirling, Load Testing Terminology, Quality Techniques Newsletter (2002). http://blogs.rediff.com/sumitjaitly/2007/06/04/load-test-article-by-scott-stirling/

*All mentioned brands and trademarks are the property of their owners.