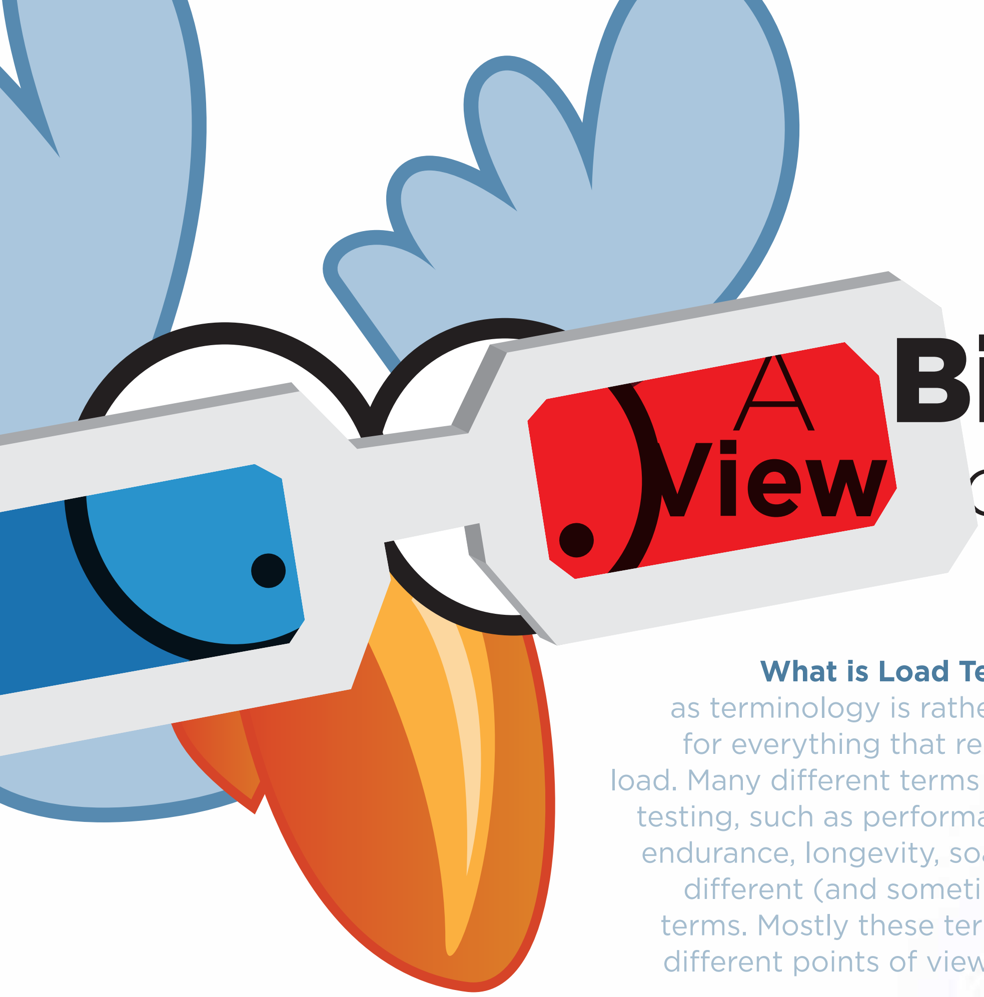


by Alexander PODELKO

A Bird's-Eye View of Load Testing

What is Load Testing? Let's first define load testing as terminology is rather vague here. The term is used here for everything that requires applying multi-user synthetic load. Many different terms are used for such kind of multi-user testing, such as performance, concurrency, stress, scalability, endurance, longevity, soak, stability, reliability, etc. There are different (and sometimes conflicting) definitions of these terms. Mostly these terms describe testing from somewhat different points of view, so they are not mutually exclusive.

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 workload to the system. The term 'load testing' is used further in this article because, by author's opinion, it better contrasts multi-user testing with other performance engineering methods, such as single-user performance testing. Everything mentioned here applies to performance, stress, concurrency, scalability, and other kinds of testing as far as the system is tested by applying multi-user load.

If we define load testing in this way, it becomes evident that it is much wider than the stereotypical waterfall-like last-moment record-and-replay load testing that we often see in large corporations. Unfortunately load testing often became associated with that stereotype that makes it difficult to see a larger picture of load testing as an important and integral part of the performance engineering process.

The recent trends of cloud computing, agile development, DevOps, and web operations are drastically re-defining the IT landscape and to see how they would impact load testing and how load testing could be adjusted, it is important to see a bigger picture. This article will consider load testing from a few different angles, important from the recent trends point of view, without diving into too many details.

Why Do We Need Load Testing?

Load testing is a way to mitigate load- and performance-related risks. There are other approaches and techniques that also alleviate performance risks:

- ***Single-User Performance Engineering***. Profiling, tracking and optimization of single-user performance, Web Performance Optimization (WPO), etc. Everything that helps to ensure that single-user response times, the critical performance path, match our expectations.
- ***Software Performance Engineering (SPE)***. Performance patterns and anti-patterns, scalable architectures, modeling, etc. Everything that helps in selecting appropriate architecture and design and proving that it will scale according to our needs.
- ***Instrumentation / Application Performance Management / Monitoring***. Everything that provides insights in what is going on inside the working system and tracks down performance issues and trends.
- ***Capacity Planning / Management.*** Everything that ensures that we will have enough resources for the system.
- ***Continuous Integration / Deployment.*** Everything allowing quick deployment and removal of changes, decreasing the impact of performance issues.

Every approach or technique mentioned above somewhat mitigates performance risks and improves chances that the system will perform up to expectations; however, none of them guarantees that. And, moreover, none may completely replace the others, as each one addresses different facets of performance.

In particular, none of the other methods to mitigate performance risks or their combination may completely replace load testing. Yes, they definitely decrease performance risk compared to situations where nothing is done about performance at all until the last moment before rolling out the system in production without any instrumentation, but they still leave risks of crashing and performance degradation under multi-user load. And if its cost is high, you should do load testing.

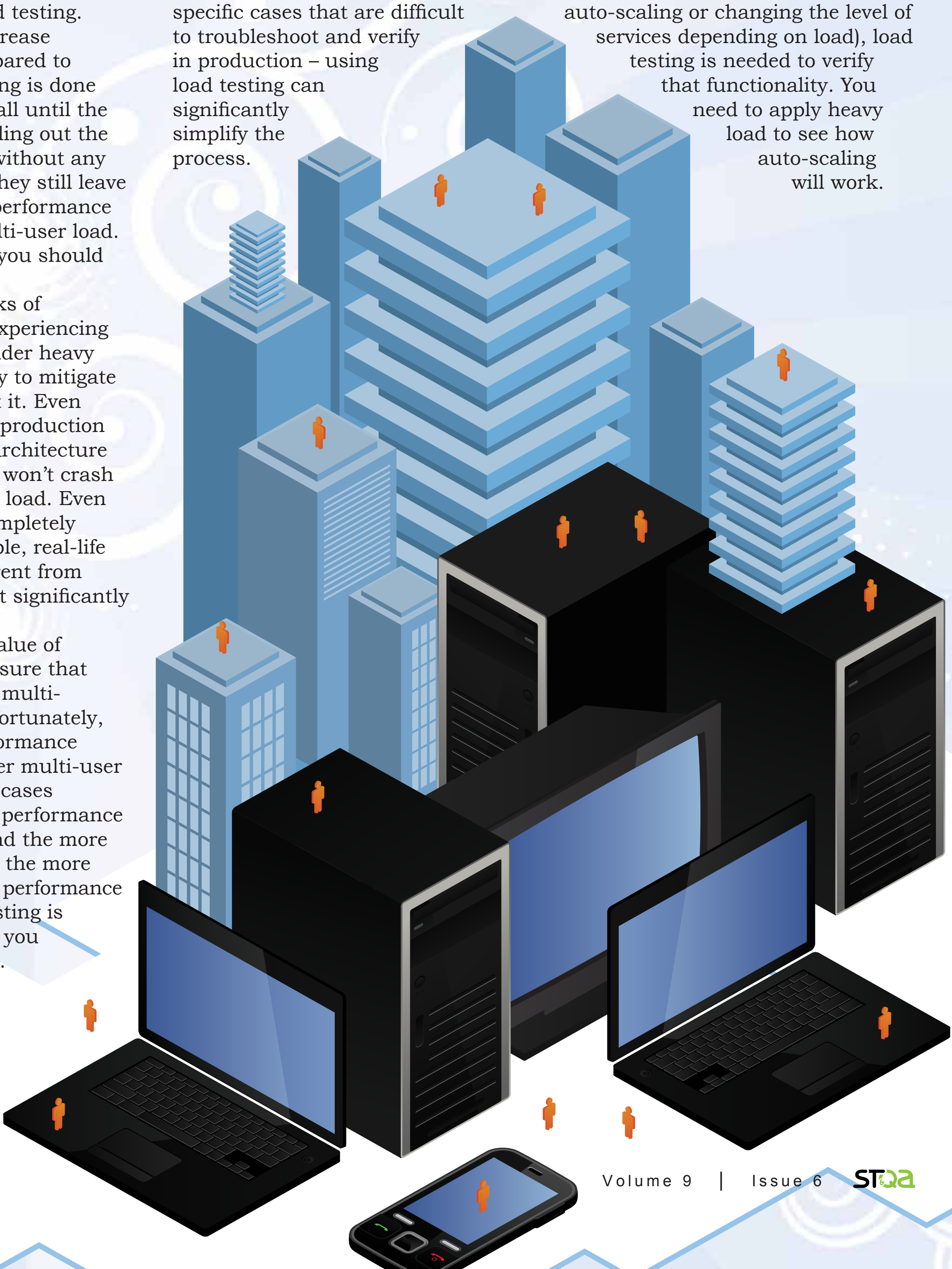
There are always risks of crashing a system or experiencing performance issues under heavy load – and the only way to mitigate them is to actually test it. Even stellar performance in production and a highly scalable architecture don't guarantee that it won't crash under a slightly higher load. Even load testing doesn't completely guarantee it (for example, real-life workload may be different from what was tested), but it significantly decreases the risk.

Another important value of load testing is making sure that changes don't degrade multi-user performance. Unfortunately, better single-user performance doesn't guarantee better multi-user performance. In many cases it improves multi-user performance too, but not always. And the more complex the system is, the more likely exotic multi-user performance issues can be. Load testing is the way to ensure that you don't have such issues.

And when you do performance optimization, you need a reproducible way to evaluate the impact of changes on multi-user performance. The impact of the changes on multi-user performance won't probably be proportional to what you see with single-user performance (even if it would be somewhat correlated). The actual effect is difficult to quantify without multi-user testing. The same with the issues happening only in specific cases that are difficult to troubleshoot and verify in production – using load testing can significantly simplify the process.

It may be possible to survive without load testing by using other ways to mitigate performance risks ***if*** the cost of performance issues and downtime is low. However, it actually means that you use users to test your system, addressing only those issues that pop up; this approach becomes risky once performance and downtime start to matter.

Moreover, with existing trends of system self-regulation (such as auto-scaling or changing the level of services depending on load), load testing is needed to verify that functionality. You need to apply heavy load to see how auto-scaling will work.



So load testing becomes a way to test functionality of the system, blurring the traditional division between functional and non-functional testing.

Load Testing Process Overview

Load testing is emerging as an engineering discipline of its own, based on "classic" testing from one side, and system performance analysis from another side. A typical load testing process is shown in figure 1.

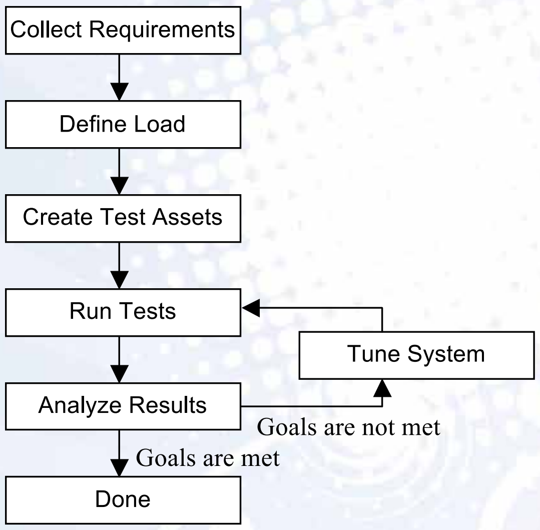


Fig.1 Load Testing Process

We explicitly define two different steps here: 'define load' and 'create test assets'. The 'define load' step (sometimes referred to as workload characterization or workload modeling) is a logical description of the load we want to apply (like "that group of users login, navigate to a random item in the catalog, add it to the shopping cart, pay, and logout with average 10 seconds think time between actions"). The 'create test assets' step is the implementation of this workload, and conversion of the logical description into something that will physically create that load during the 'run tests' step. While for manual testing that can be just the description given to each tester, usually it is something else in load testing – a program or a script.

Quite often load testing goes hand-in-hand with tuning, diagnostics, and capacity planning. They are actually represented by the back loop on Fig.1: if we don't meet our goal, we need to optimize the system to improve performance. Usually the load testing process implies tuning and modification of the system to achieve the goals.

Load testing is not a one-time procedure. It spans through the whole system development life cycle. It may start from technology or prototype scalability evaluation, continue through component / unit performance testing into system performance testing and follow up in production to troubleshoot performance issues and test upgrades / load increases.

Load Generation

Before we can move forward from 'define load' to 'create test assets', we need to decide how we are going to generate that load. Load generation can be a simple technical step when you know how to do it for your system (compared with other non-trivial steps like collecting requirements, defining load, or analyzing results). Unfortunately, quite often it is a very challenging task for a new system, up to being impossible in the given time frame. It is important to understand all possible options; a single approach may not work in all situations. The main choices are to generate workload manually (really an option only if you test few users), use a load testing tool (software or hardware), or create a program to do it. Many tools allow using different ways of recording/playing back and programming. Let's consider different approaches to load generation and what are their pros and cons.

Record and Playback: Protocol Level

The mainstream approach of load testing (at least for business and Internet applications) is recording communication between two tiers of the system and playing back the automatically created script (usually, of course, after proper correlation and parameterization). Tools used for that are usually referred to as "load testing tools" and users simulated by such tools are usually referred as "virtual users". The real client-side software isn't necessary to replay such scripts, so the number of simulated virtual users can be high; it is theoretically limited only by available hardware (each tool has specific hardware requirements depending on the type and complexity of scripts).

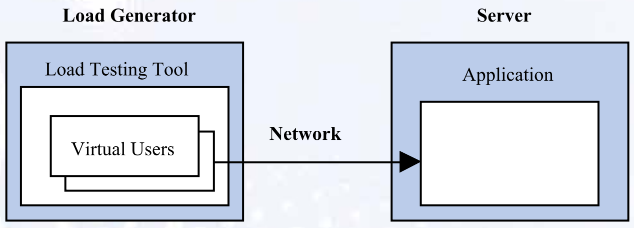


Fig.2 Record and Playback Approach, Protocol Level

Both recording and playback happen between the tiers, so the protocol used between the client and the server is extremely important. Other factors, such as what language was used to develop the system, what platform the server is deployed on, etc. are usually irrelevant for scripting (although they can give some hints about what protocol is used for communication).

The process is reasonably straightforward when you test a simple website or a simple web application with a thin client. Even a beginner in load testing can quickly create a few scripts and run tests. That is one reason why the record and playback approach is so popular. However, there is a trap in that easiness: load testing really embraces much more. Load should be validated for correctness (if you don't see errors in the load testing tool it doesn't always mean that it works properly) and realism (using unrealistic scenarios is the easiest way to get misleading results). Moreover, load generation is only one step in load testing, there are many other important parts (like getting requirements and doing results analysis), as well as other related activities (like tuning or diagnostics).

Unfortunately, scripting can be challenging even for a web application. Recording a script and making it work can be a serious research task, often including many try-and-fail iterations. A good load testing tool can help if it supports your protocol.

The protocol level record and playback approach has several serious limitations:

- It usually doesn't work for testing components and services.
- Each particular load testing tool supports a limited number of technologies.
- Some technologies require very time-consuming correlation and parameterization and some may be not supported at all.
- The workload validity in case of sophisticated logic on the client side is not guaranteed.

These limitations are usually not a problem in the case of simple web applications using a browser as a client, but they become a serious problem when you need to test different protocols across the whole software lifecycle.

Each load testing tool supports a limited number of technologies (protocols). New or exotic technologies are not usually on the list. Vendors of load test tools add new supported protocols continually, but we often do not have time to wait for the specific protocol to be added – as soon as we get a new product we need to test it.

For example, back in 1999, we were not able to use recording for the SMB (Server Message Block) protocol, later succeeded by the Common Internet File System (CIFS) protocol, Microsoft DCOM (Distributed Component Object Model), or Java RMI (Remote Method Invocation). While some vendors claimed that their products supported these protocols, it didn't work in all environments.

Later there were issues with Java applets and ActiveX controls, which used serialization, encoding, or even proprietary protocols.

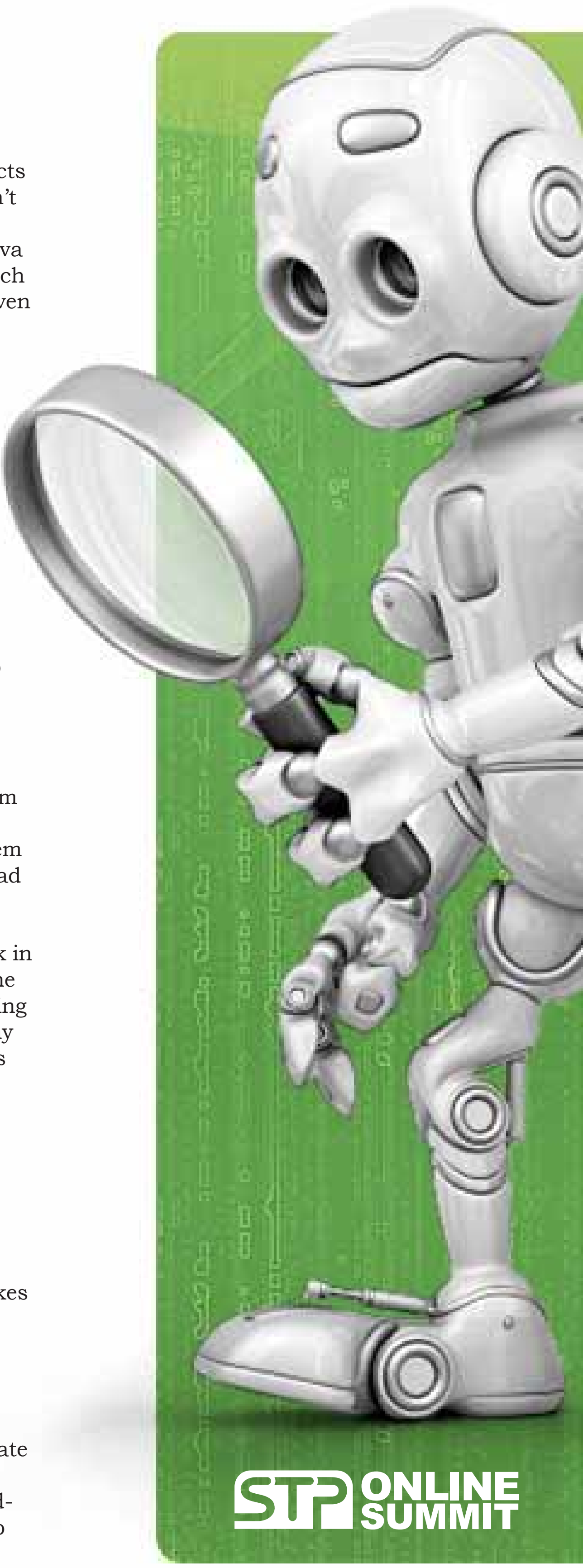
Today we are getting a new generation of Rich Internet Applications (RIA) and new web protocols, bringing these old challenges of protocol level recording back – so some authors started to talk about a crisis of performance testing. Still these issues don't look any more challenging than the issues we had 10-15 years ago – especially considering that many still use underlying standard web protocols, so we at least are able to record the communication.

Even if the protocol is supported, script recording and parameterization often are far from being straightforward and often require a good knowledge of system internals. The question of workload validation is also opened.

So, it is possible that the record and playback approach won't work in your environment, or that using the approach will be too time-consuming and inflexible (as it happened many times for us). When such problems are encountered, it is a good time to check other alternatives and add them to your arsenal.

Record and Playback: UI-Level

Another approach to simulating user activities is to record user interactions with Graphical User Interface (GUI) – such as keystrokes and mouse clicks - and then play them back. Users, simulated by using such approach, are sometimes referred as GUI users. The tools using this approach simulate users in the most accurate way: they take the place of a real user. You are supposed to get end-to-end response times identical to what users would see.



Originally such tools were mostly used for automated functional testing, although the option to use this approach for load testing was available for a long time. For load testing, these GUI tools were usually used in conjunction with the load testing tool from the same vendor, which coordinated execution of multiple GUI scripts and collected results.

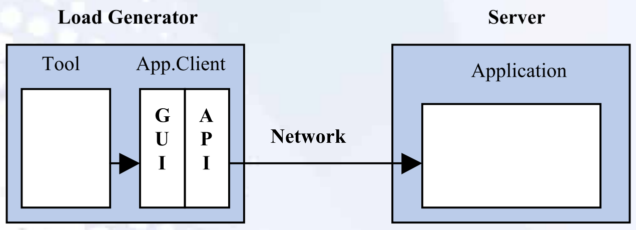


Fig.3 Record and Playback Approach, GUI Users

The main problem with such tools was that these tools drive an instance of client software and require a machine for each user, so it was almost impossible to use them for a large number of simulated users – you need the same number of physical boxes as the number of users being simulated. Some tools have the ability to run one user per Windows Terminal Server session that significantly increases scalability of the solution (probably up to low hundreds of users from a practical point of view).

Another known option was, for example, using the low-level graphical Citrix or Remote Desktop protocols – which always were the last resort when nothing else was working, but were notoriously tricky to playback. It works fine when you indeed use Citrix or Remote Desktop. But using it as a workaround means that you test a significantly different setup than you use in real life (with multiple clients parts running on a server) that may undermine the value of testing.

Nowadays most applications have web-based interface and a new generation of UI-level tools for browsers extend possibilities of the UI-level approach allowing to run multiple browsers per machine (so scalability is limited by the machine resources available to run browsers). Perhaps we can refer to users simulated by such tools as browser users (because low-level browser control is usually used).

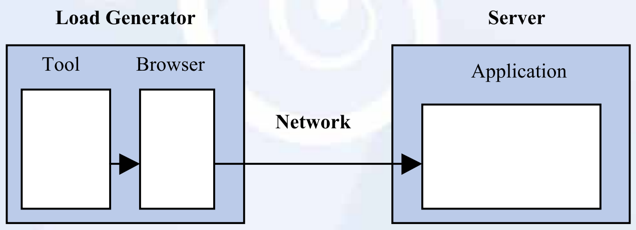


Fig.4 Record and Playback Approach, Browser Users

Moreover, UI-less browsers were created, such as HtmlUnit or PhantomJS, which require significantly less resources than real browsers. This drastically increased scalability of the UI-level approach and made it much more viable for load testing now, but the approach still remains less scalable than the protocol-level approach just because all these browsers (even the light-weight ones) still need to be run and all client-side application code be executed on the load generator machine.

Using the UI-level approach for load testing sounds very promising: we get end-user timing and do not depend on intricacies of the client-server communication. However, questions of supported technologies, scalability, and timing accuracy remain largely undocumented, so the approach requires evaluation in every non-trivial case. So far the approach is mostly used to re-use existing functional testing scripts or when it is impossible to use protocol-level scripts.

Manual

Manual load generation isn't a real option if we want to simulate a large number of users. Still, in some cases, it can be a good option when we need load from a few users and don't have proper tools available or face serious issues with scripting. Sometimes a manual test can be a good option in earlier stages of testing to verify that the system can support concurrent work or to diagnose, for example, locking problems.

One of the concerns with manual testing is that even when each user has an exact scenario, time variations can occur; so the tests are not exactly reproducible due to variations in human input times. Such an approach hardly can be recommended as a long term solution, even with few users.

It still could be useful to run one or few users manually in parallel to simulated virtual users' workload to better understand what real users would experience. That is a good way to verify test results: if manual response times match what you see for the scripts, it is an indication that your scripts are correct.

Programming: Custom Test Harness

Programming is another approach to load generation. A straightforward way to create a multi-user workload is to develop a special program to generate workload. This program requires access to the Application Programming Interface (API) or source code and some programming work. It is often used to test components. No special testing tool is necessary (although some tools are available that can simplify work).

In some simple cases it could be the best solution (from a cost perspective, especially if there is no purchased load testing tool). A starting version could be quickly created by a programmer familiar with the API. A simple test harness, for example, could spawn several threads and each thread, simulating a real user, could include the same sequence of API calls as the real software for that use case. No need to worry about what protocol is used for communication.

We successfully used this approach for component load testing in several projects (and, of course, this approach is widely used by developers). However, efforts to update and maintain the harness increase drastically as soon as you need to add such features as, for example:

- Complex user scenarios
- Centralized test management and result analysis
- Coordinated test execution from several computers

If you have numerous products, you really need to create something like a commercial load testing tool to assure all necessary performance and reliability testing. It probably isn't the best choice for a small group of testers.

Programming: Using Load Testing Tools
Many advance load testing tools support one (or several) scripting languages allowing you to program scripts in whatever way is necessary while using the tool to manage scripts executions, collect and analyze the results. It may be direct programming of server requests, using web services, or using API. If using API, the approach may need lightweight custom software clients (client stubs) to create the correct workload.

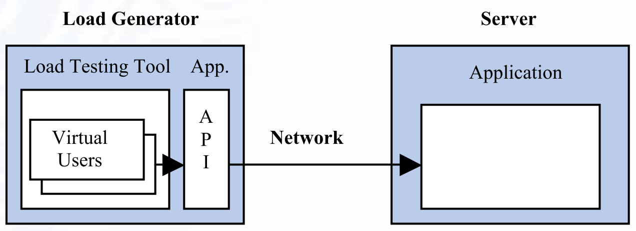


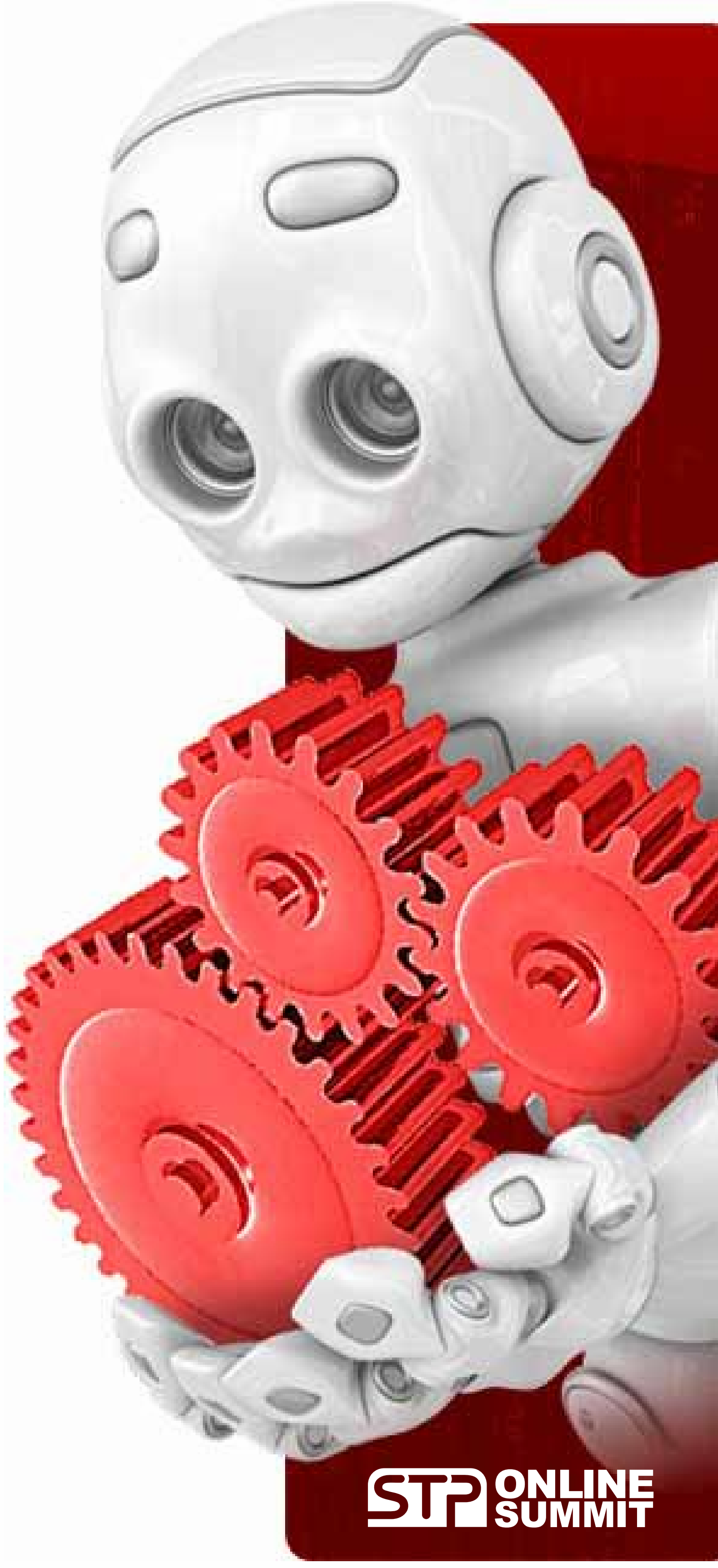
Fig 5. Programming API Using a Load Testing Tool.

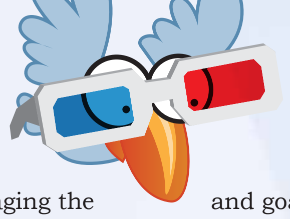
The implementation of this approach (we called it custom load generation) depends on the particular load testing tool. The original way was to create an external C dll (or shared library for UNIX) and then call functions defined in the dll from the tool's native script language.

Another way to implement this approach appeared in the later versions of load testing tools: creating a script in a programming language (such as Java or Visual Basic) with the help of templates and special tool-supplied functions.

These are significant advantages of this custom load generation approach:

- It eliminates dependency on the third-party tool to support specific protocols.
- It leverages all the features of existing load testing tools and allows use of them as a test harness.
- It takes away the need to implement multi-user support, data collection and analysis, reporting, scheduling, etc. This is inherent in the third-party tool.
- It ensures that performance testing of current or future applications can be done for any protocol used to communicate among different tiers





Custom load generation may allow managing the workload in a more user-friendly way by simplifying parameterization.

For example, if you record socket-level traffic, recording and parameterization could take a lot of time. And if you need to change the workload (for example, use new queries), it is almost impossible to change the parameterized script to reflect the new workload. You probably need to re-record and re-parameterize the script.

When you implement custom load generation, the real query could be read from an input file. Changing the query becomes very easy: you just change the input file without any changes in the script.

The same is true if different builds of the software are tested. Small changes could impact a low-level protocol script, but the API is usually more stable. Just install the new build and run the test. There is no new recording and parameterization needed.

But, of course, there are some considerations to keep in mind for the custom load generation approach:

- It requires access to API or source code.
- It requires additional programming work.
- It requires an understanding of internals (to re-create the sequence used by real users).
- The client environment should be set up on all load generator machines.
- It requires commercial tool licenses for the necessary number of virtual users.
- It usually requires more resources on client machines (since there is some custom software).
- The results should be carefully interpreted (to insure that there is no contention between client stubs).

Programming may be a better solution in many cases, but it is not a full replacement of recording approaches. In cases when recording works well, it usually provides better and more efficient solutions. One of important advantages of recording is that that the tool records exactly whatever communication happens between user and server – while with programming it is often what the person creating scripts *think* the communication is. Unfortunately communication between user and server is often very complicated and difficult to reproduce programmatically. So the tools that support only programming and does not support recording have a rather limited area of application.

Environments

There is always a lot of discussion on what test environments should be. Most experts agree that it should be as close to production as possible, but what it exactly means and what to do when, due to different limitations, it is impossible to have it similar to production, is always a topic for discussions.

The cloud introduced new opportunities and challenges to performance testing, but specific pros and cons vary significantly depending on your environment and goals. The term cloud is overused and covers a lot of different options. If we want to understand how cloud may impact performance testing, we should consider all these options separately as they bring a completely different performance testing context.

In performance testing we have two main components: the system under test and load generators (we may have other components for monitoring, results analysis, etc., but they are not so important in the context of this discussion).

When we talk about load generators, we have three main options:

- Have them locally, the traditional option (for example, in a test lab).
- Have them as a service. This option existed for a long time (for example, load testing services provided by Gomez, Keynote, and other companies). While we can refer to it as a SaaS (Software as a Service) cloud now, the only real change is that we have more such companies (and, respectively, more choices) because it is easier to start such service using cloud to provide infrastructure.
- Have them in IaaS (Infrastructure as a Service) clouds. This is a new option and it makes it easy to get a large number of remote load generators. It was always possible to have a load generator on a remote machine, but now it is much easier to get it. Some tools provide help with cloud deployments, which may be very handy when you need a large number of load generators for a large-scale test.

When we talk about the system under test, in addition to having the system locally (which may be anything from a development machine to the production system), we may deploy it in a cloud now. It helps to overcome one of the main reasons of not testing full-scale setups, lack of hardware resources: now you can get as much hardware as you want when you are ready for that. However it may be not exactly the same kind of hardware and software that you use in your production system, so getting closer to the scale of the system you may be farther away in details of the environment.

What configuration would be better for you depends on what are the goals of performance testing. Performance testing in the cloud (or from the cloud) makes sense for certain types of performance testing. For example, it should work great if we want to test how many users the system supports, would it crash under load of X users, or how many servers we need to support Y users, but when we are not too concerned with exact numbers or variability of results (or even want to see some real-life variability).

Even in this case the assumptions are that we don't introduce any bottleneck using the cloud (for example, saturating network bandwidth between the load generators and the system under test) and leave to the cloud provider to care that our tests don't impact other cloud tenants.

However it doesn't work well for performance optimization, when we make a change in the system and want to see how it impacts performance. Testing in a cloud with other tenants intrinsically has some results variability as far as we don't control other activities in the cloud and in most cases don't even know the exact hardware configuration. The effects may be even more sophisticated in case of Platform as a Service (PaaS) or SaaS clouds. So when we talk about performance optimization, we may still need an isolated environment.

One interesting case is when the system is created to be used in a cloud, which probably would be more and more common with time. The first thought would be that it simplifies the choice, you just test it in the cloud where it is supposed to be deployed. Still it won't work too well if you need to do performance optimization or troubleshooting and want tests to be completely reproducible. In this case you may need something like an isolated private cloud with hardware and software infrastructure similar to the target cloud and monitoring access to the underlying hardware to see how the system maps to hardware resources and if it works as expected. Real-world network emulators may be used to make sure that performance testing is representative of how the system would be used in production – otherwise we don't take into account such factors as network latency, bandwidth, jitter, etc. So if we need optimization for cloud software, we may still need a lab – but the lab should be more sophisticated to emulate the cloud environment and real-world network conditions. An ultimate example of such lab is the lab Microsoft created for testing IE, described at http://blogs.msdn.com/b/b8/archive/2012/02/16/internet-explorer-performance-lab-reliably-measuring-browser-performance.aspx.

Thus we have different options for the system and load generator deployments, and what option (or combination of options) would be the best depends on the goals of performance testing. For example, some typical performance testing scenarios may be:

- System validation for high load. Outside load (service or cloud) against the production system may be the best option here. We have a wider scope of testing, but lower repeatability.
- Performance optimization / troubleshooting. An isolated environment may be the best option here. We have a limited scope, but high repeatability.
- Testing in Cloud. It may be the best option for periodic tests to lower costs. We have a limited scope and low repeatability.

So by factoring in the cloud into performance testing, we have at least two major alternatives (with a variety of more subtle options): coarse performance testing in or from the cloud with inherent variability (and probably some savings on hardware and configuration costs) or granular performance testing and optimization in an isolated environment (thus avoiding variability with probably higher hardware and configuration costs). For comprehensive performance testing you may even need both lab testing (with reproducible results for performance optimization) and realistic outside testing from around the globe (to check real-life issues that you can't simulate in the lab). Doing both would be expensive and makes sense only when performance really matters – but if you are not there yet, you may get there eventually.

Automation

One of the main trends in software development now is automation. The whole DevOps trend is, in a way, based on automation. And load testing is trailing far behind here. There are, of course, objective reasons for that: it is just much more difficult to automate than, for example, functional testing: you usually need a more sophisticated setup, have many more factors that may impact tests, and results are complex and difficult to interpret as pass/fail.

While, of course, load testing is more difficult to automate than other activities such as building software, functional testing or deployment, it is not impossible. It is surprising that tool vendors don't provide much functionality yet to support such activities.

There were several good presentations sharing experience of single-user performance test automation, in some cases performance information was collected in parallel to performance tests. It looks like a good first step in the right direction, but the author hasn't heard yet about good examples of multi-user load test automation. However, it is not something that can be done without tools support – either they would be created from a scratch, or existing tools would add such functionality.

There is not much to discuss in this section now, but we may see significant developments in that area in the foreseeable future.

Selecting Load Testing Tools

Classifying and evaluating load testing tools is not easy as they include different sets of vaguely defined functionality, often over-embellished by vendors. In most cases, available classifications are either an oversimplification (which in some cases still may be useful) or a marketing trick to highlight advantages of specific tools. There are many criteria to use to differentiate load testing tools and it is probably better to evaluate tools on each criterion separately. While the considerations below may look somewhat generic, the author explicitly decided not to mention any specific tool due to limited space and to prevent potential vendor complaints.

Load Generation

As it was discussed in the Load Generation section, there are three main approaches on how tools may generate load and every tool may be evaluated on which of them it supports and how it performs:

- Protocol-level recording and the list of supported protocols
- UI-level recording
- Programming

Supported Environments

As was discussed in the Environments section, it is important to understand what environments the tool is supporting and how well. Depending on the goals of load testing, you may need support of one or several types of environments.

Whether it is lab or cloud, an important question is what kind of software / hardware / cloud the tool requires. Many tools use low-level system functionality, so there may be unpleasant surprises when the platform of your choice or your corporate browser standard is not supported.

Scaling

When you have a few users to simulate, it usually is not a problem. The more users you need to simulate, the more important it becomes. Tools differ drastically on how many resources they need per simulated user and how well they may handle large volumes of information. It may differ significantly even for a specific tool depending on protocol used and specifics of your script. As soon as you get to 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. Load testing appliances can be useful for simulating a large number of simple Web users, but scripting is usually limited.

Monitoring and Result Analysis

These two very important sets of functionality are often an indicator of how mature the tool is. While theoretically it is possible to do both using other tools (and it is usually suggested by the vendors who don't have such functionality built-in), it significantly degrades productivity and may require building some plumbing infrastructure. So while these two areas may look optional, integrated and powerful monitoring and result analysis are very important. And the more complex the system and tests are, the more important they are.

Teamwork Support

Performance is mostly a team exercise and you would need to share artifacts and results with other members of the team. In some cases concurrent access to running tests and result analysis may be needed. For licensed tools, how licenses may be shared may have significant financial consequences.

Automation Support

As discussed in the Automation section, while it is rarely clearly spelled out and difficult to formalize, automation support would probably become an important criterion in the near future. Whatever features are needed, they should be explicitly checked – many tools may lack even very basic features like scheduling a test run.

Of course, non-technical criteria are important too:

Cost/Licensing Model

There are commercial tools (and license costs differ drastically) and free tools. And there are some choices in between when a limited edition is available for free and the full version may be purchased. There are many free tools (a few are mature and well-known) and many inexpensive tools, but most of them are very limited in functionality. Switching tools in the future is possible, but has notable costs associated with it (efforts to make the current tool work, the learning curve, re-doing jobs already done with the old tool), so it makes sense to make sure that the chosen tool(s) will support performance engineering activities for the foreseeable future.

Skills
Considering a large number of tools and a relatively small number of people working in the area, there is a kind of labor market only for the most popular tools. Even for the second-tier tools there are few people around and few positions available. So if you don't choose the market leaders, you can't be certain that you will find people with this tool experience. Of course, an experienced performance engineer will learn any tool – but it may take some time until productivity gets to the expected level.

Support
Recording and load generation has a lot of sophistication in the background and issues may happen in every area. Availability of good support may significantly improve productivity.

This is, of course, not a comprehensive list of criteria – rather a few starting points. Unfortunately, in most cases you can't just rank tools on the better/worse scale. It may be that a simple tool will work quite well in your case. If your business is built around a single web site, it doesn't use sophisticated technologies, and load is not extremely high – almost every tool will work for you. The further you are from this state, the more challenging it will be to pick up the right tool. And it even may be that you need several tools.

And while you may evaluate tools with the above mentioned criteria, it is not guaranteed that a specific tool will work with your specific product (unless it uses a well-known and straightforward technology). That actually means that if you have a few systems to test, you need to evaluate the tools you consider using your systems and see if the tools can handle them. If you have many, choosing a tool supporting multiple load generation options is probably a good idea (and, of course, check it with at least the most important systems).

Summary
Load testing is an important way to mitigate performance risks and should be an integral part of the system lifecycle. While we have other ways to mitigate performance risks, they can't completely replace load testing but rather complement it. Maybe there would be less need for simplistic load testing due to better instrumenting, APM tools, continuous integration, etc. – but we may expect more need for performance experts that would be able to see the whole picture using all available tools and techniques.

There is no best approach to load generation, setting test environments, test creating and execution, plus, there is no best load testing tool for every scenario. Some approaches or tools may be better in a particular context. It is quite possible that a combination of tools and approaches would be necessary in complex environments. Choosing the right strategy in load testing may be a challenging task. While digging deeply into details of particular projects and tools may be needed, it is good to see a bigger picture of what approaches and tools are available and what are their advantages and disadvantages.

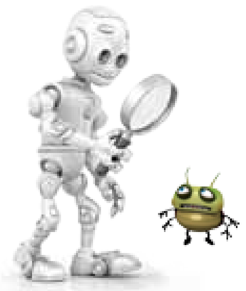
About The Author

Alexander Podelko *is Consulting Member of Technical Staff at Oracle.*

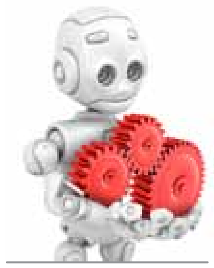
STP ONLINE SUMMIT

STP Online Summits are online events that are delivered for three and a half hours per day over three consecutive days. Attend all sessions, interact with the presenters and network with your peers right from the comfort of your home or office! If you can't attend the live sessions – don't worry! – you will have access to all of the session recordings.

2013 STP ONLINE SUMMIT SCHEDULE



Finding Defects:
Tips, Tools & Techniques
MARCH
12 – 14, 2013



Test Automation:
Anatomy, Architecture & Approaches
MAY
14 – 16, 2013

Performance Testing:
Defining, Executing & Interpreting
AUGUST
20 – 22, 2013

Mobile App Testing:
Scope, Environments & Devices
NOVEMBER
12 – 14, 2013

WWW.SOFTWARETESTPRO.COM/ONLINESUMMIT