Reinventing Performance Testing

Alexander Podelko
Oracle

Load testing is an important part of the performance engineering process. However the industry is changing and load testing should adjust to these changes - a stereotypical, last-moment performance check is not enough anymore. There are multiple aspects of load testing - such as environment, load generation, testing approach, life-cycle integration, feedback and analysis - and none remains static. This presentation discusses how performance testing is adapting to industry trends to remain relevant and bring value to the table.

Performance Engineering Puzzle

There are many discussions about performance, but they often concentrate on only one specific facet of performance. The main problem with that is that performance is the result of every design and implementation detail, so you can't ensure performance approaching it from a single angle only.

There are different approaches and techniques to alleviate performance risks, such as:

- **Software Performance Engineering (SPE).** Everything that helps in selecting appropriate architecture and design and proving that it will scale according to our needs. Including performance patterns and anti-patterns, scalable architectures, and modeling.

- **Single-User Performance Engineering.** Everything that helps to ensure that single-user response times, the critical performance path, match our expectations. Including profiling, tracking and optimization of single-user performance, and Web Performance Optimization (WPO).

- **Instrumentation / Application Performance Management (APM)/ 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. Including both people-driven approaches and automatic self-management such as auto-scaling.

- **Load Testing.** Everything used for testing the system under any multi-user load (including all other variations of multi-user testing, such as performance, concurrency, stress, endurance, longevity, scalability, reliability, and similar).

- **Continuous Integration / Delivery / Deployment.** Everything allowing quick deployment and removal of changes, decreasing the impact of performance issues.

And, of course, all the above do not exist not in a vacuum, but on top of high-priority functional requirements and resource constraints (including time, money, skills, etc.).

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 completely replaces the others, as each one addresses different facets of performance.

A Closer Look at Load Testing

To illustrate that point of importance of each approach let's look at load testing. With the recent trends towards agile development, DevOps, lean startups, and web operations, the importance of load testing gets sometimes questioned. Some (not many) are openly saying that they don't need load testing while others are still paying lip service to it – but just never get there. In more traditional corporate world we still see performance testing groups and most important systems get load tested before deployment. So what load testing delivers that other performance engineering approaches don’t?

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 the system. Even stellar performance in production and a highly scalable architecture don't guarantee that it won't crash under a slightly higher load.

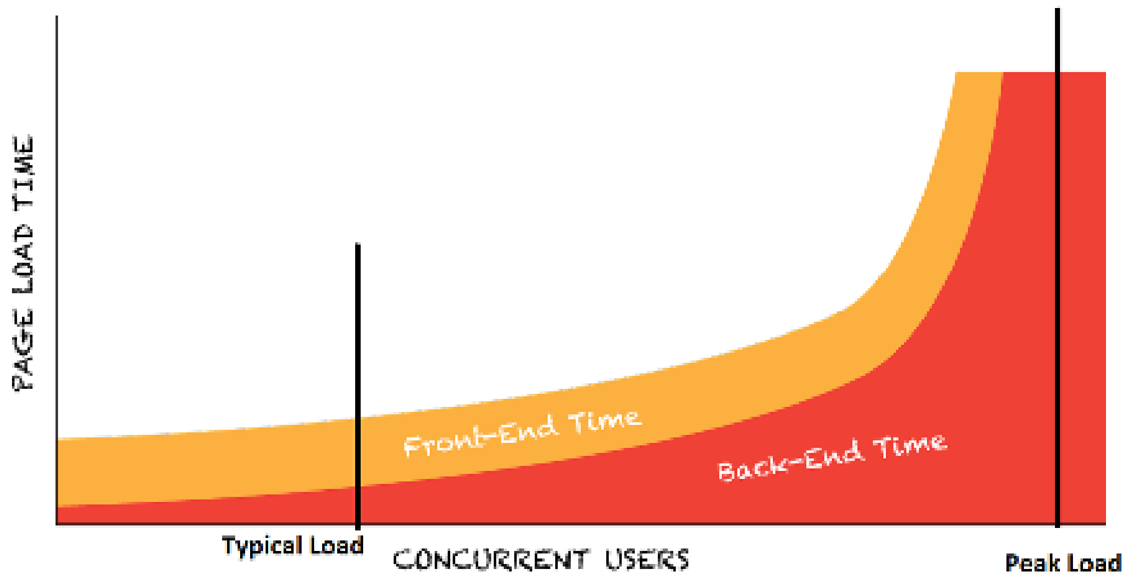

Fig.1. Typical response time curve.

A typical response time curve is shown on fig.1, adapted from Andy Hawkes’ post discussing the topic [HAWK13]. As it can be seen, a relatively small increase in load near the curve knee may kill the system – so the system would be unresponsive (or crash) under the peak load.

However, load testing doesn't completely guarantee that the system won’t crash: for example, if the real-life workload would be different from what was tested (so you need to monitor the production system to verify that your synthetic load is close enough). But load testing significantly decreases the risk if done properly (and, of course, may be completely useless if done not properly – so it usually requires at least some experience and qualifications).

Another important value of load testing is checking how changes impact multi-user performance. The impact on multi-user performance is not usually proportional to what you see with single-user performance and often may be counterintuitive; sometimes single-user performance improvement may lead to multi-user performance degradation. And the more complex the system is, the more likely exotic multi-user performance issues may pop up.

It can be seen on fig.2, where the black lines represent better single-user performance (lower on the left side of the graph), but worse multi-user load: the knee happens under a lower load and the system won’t able to reach the load it supported before.

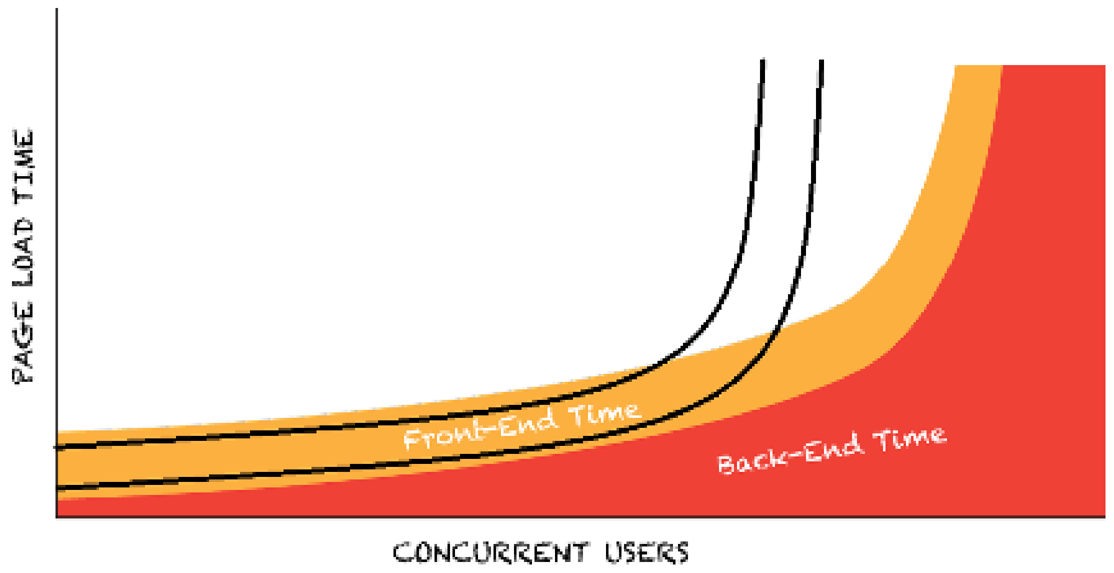

Fig.2 Single-user performance vs. multi-user performance.

Another major value of load testing is providing a reliable and reproducible way to apply multi-user load needed for performance optimization and performance troubleshooting. You apply exactly the same synthetic load and see if the change makes a difference. In most cases you can't do it in production when load is changing – so you never know if the result comes from your code change or from change in the workload (except, maybe, a rather rare case of very homogeneous and very manageable workloads when you may apply a very precisely measured portion of the real workload). And, of course, a reproducible synthetic workload significantly simplifies debugging and verification of multi-user issues.

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 nonfunctional testing.

Changing Dynamic

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 customers to test your system, addressing only those issues that pop up; this approach become risky once performance and downtime start to matter.

The question is discussed in detail in Load Testing at Netflix: Virtual Interview with Coburn Watson [PODE14a]. As explained there, Netflix was very successful in using canary testing in some cases instead of load testing. Actually canary testing is the performance testing that uses real users to create load instead of creating synthetic load by a load testing tool. It makes sense when 1) you have very homogenous workloads and can control them precisely 2) potential issues have minimal impact on user satisfaction and company image and you can easily rollback the changes 3) you have fully parallel and scalable architecture. That was the case with Netflix - they just traded in the need to generate (and validate) workload for a possibility of minor issues and minor load variability. But the further you are away from these conditions, the more questionable such practice would be.

Yes, the other ways to mitigate performance risks mentioned above definitely decrease performance risks comparing to situations where nothing is done about performance at all. And, perhaps, may be more efficient comparing with the old stereotypical way of doing load testing – running few tests at 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. So actually you need to have a combination of different performance engineering approaches to mitigate performance risks – but the exact mix depends on your system and your goals. Blindly copying approaches used, for example, by social networking companies onto financial or e-commerce systems may be disastrous.

As the industry is changing with all the modern trends, the components of performance engineering and their interactions is changing too. Still it doesn’t look like any particular one is going away. Some approaches and techniques need to be adjusted to new realities – but there is nothing new in that, we may see such changing dynamic throughout all the history of performance engineering.

Historical View

It is interesting to look how handling performance changed with time. Probably performance engineering went beyond single-user profiling when mainframes started to support multitasking, forming as a separate discipline in 1960-s. It was mainly batch loads with sophisticated ways to schedule and ration consumed resources as well as pretty powerful OS-level instrumentation allowing to track down performance issues. The cost of mainframe resources was high, so there were capacity planners and performance analysts to optimize mainframe usage.

Then the paradigm changed to client-server and distributed systems. Available operating systems didn't have almost any instrumentation or workload management capabilities, so load testing became almost only remedy in addition to system-level monitoring to handle multi-user performance. Deploying across multiple machines was more difficult and the cost of rollback was significant, especially for Commercial Of-The-Shelf (COTS) software which may be deployed by thousands of customers. Load testing became probably the main way to ensure performance of distributed systems and performance testing groups became the centers of performance-related activities in many organizations.

While cloud looks quite different from mainframes, there are many similarities between them [EIJK11], especially from the performance point of view. Such as availability of computer resources to be allocated, an easy way to evaluate the cost associated with these resources and implement chargeback, isolation of systems inside a larger pool of resources, easier ways to deploy a system and pull it back if needed without impacting other systems.

However there are notable differences and they make managing performance in cloud more challenging. First of all, there is no instrumentation on the OS level and even resource monitoring becomes less reliable. So all instrumentation should be on the application level. Second, systems are not completely isolated from the performance point of view and they could impact each other (and even more so when we talk about containers). And, of course, we mostly have multi-user interactive workloads which are difficult to predict and manage. That means that such performance risk mitigation approaches as APM, load testing, and capacity management are very important in cloud.

So it doesn’t look like the need in particular performance risk mitigation approaches, such as load testing or capacity planning, is going away. Even in case of web operations, we would probably see load testing coming back as soon as systems become more complex and performance issues start to hurt business. Still the dynamic of using different approaches is changing (as it was during the whole history of performance engineering). Probably there would be less need for "performance testers" limited only to running tests – due to better instrumenting, APM tools, continuous integration, resource availability, etc. – but I'd expect more need for performance experts who would be able to see the whole picture using all available tools and techniques.

Let’s further consider how today’s industry trends impact load testing.

Cloud

Cloud and cloud services significantly increased a number of options to configure for both the system under test and the load generators. Cloud practically eliminated the lack of appropriate hardware as a reason for not doing load testing and significantly decreased cost of large-scale load tests as it may provide large amount of resources for a relatively short period of time.

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 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, you probably need to use several approaches - for example, lab testing (for performance optimization to get reproducible results) and distributed, realistic outside testing (to check real-life issues you can't simulate in the lab). Limiting yourself to one approach limits the risks you will mitigate.

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.

Agile Development

Agile development eliminates the main problem of tradition development: you need to have a working system before you may test it, so performance testing happened at the last moment. While it was always recommended to start performance testing earlier, it was usually rather few activities you can do before the system is ready. Now, with agile development, we got a major "shift left", allowing indeed to start testing early.

In practice it is not too straightforward. Practical agile development is struggling with performance in general. Agile methods are oriented toward breaking projects into small tasks, which is quite difficult to do with performance (and many other non-functional requirements) – performance-related activities usually span the whole project.

There is no standard approach to specifying performance requirements in agile methods. Mostly it is suggested to present them as user stories or as constraints. The difference between user stories and constraints approaches is not in performance requirements per se, but how to address them during the development process. The point of the constraint approach is that user stories should represent finite manageable tasks, while performance-related activities can't be handled as such because they usually span multiple components and iterations. Those who suggest to use user stories address that concern in another way – for example, separating cost of initial compliance and cost of ongoing compliance [HAZR11].

And practical agile development is struggling with performance testing in particular. Theoretically it should be rather straightforward: every iteration you have a working system and know exactly where you stand with the system's performance. You shouldn't wait until the end of the waterfall process to figure out where you are – on every iteration you can track your performance against requirements and see the progress (making adjustments on what is already implemented and what is not yet). Clearly it is supposed to make the whole performance engineering process easier and solve the main problem of the traditional approach that the system should be ready for performance testing (so it happened very late in the process when the cost of fixing found issues is very high).

From the agile development side the problem is that, unfortunately, it doesn't always work this way in practice. So such notions as “hardening iterations” and “technical debt” get introduced. Although it is probably the same old problem: functionality gets priority over performance (which is somewhat explainable: you first need some functionality before you can talk about its performance). So performance related activities slip toward the end of the project and the chance to implement a proper performance engineering process built around performance requirements is missed.

From the performance testing side the problem is that performance engineering teams don't scale well, even assuming that they are competent and effective. At least not in their traditional form. They work well in traditional corporate environments where they check products for performance before release, but they face challenges as soon as we start to expand the scope of performance engineering (early involvement, more products/configurations/scenarios, etc.). And agile projects, where we need to test the product each iteration or build, expose the problem through an increased volume of work to do.

Just to avoid misunderstandings, I am a strong supporter of having performance teams and I believe that it is the best approach to building performance culture. Performance is a special area and performance specialists should have an opportunity to work together to grow professionally. The details of organizational structure may vary, but a center of performance expertise (formal or informal) should exist. Only thing said here is that while the approach works fine in traditional environments, it needs major changes in organization, tools, and skills when the scope of performance engineering should be extended (as in the case of agile projects).

Remedies recommended are usually automation and making performance everyone jobs (full immersion) [CRIS09, BARB11]. However they haven't yet developed in mature practices and probably will vary much more depending on context than the traditional approach.

Automation means here not only using tools (in performance testing we almost always use tools), but automating the whole process including setting up environment, running tests, and reporting / analyzing results. Historically performance testing automation was almost non-existent (at least in traditional environments). Performance testing automation is much more difficult than, for example, functional testing automation. Setups are much more complicated. A list of possible issues is long. Results are complex (not just pass/fail). It is not easy to compare two result sets. So it is definitely much more difficult and may require more human intervention. And, of course, changing interfaces is a major challenge. Especially when recording is used to create scripts as it is difficult to predict if product changes break scripts.

The cost of performance testing automation is high. You need to know system well enough to make meaningful automation. Automation for a new system doesn't make much sense - overheads are too high. So there was almost no automation in traditional environment (with testing in the end with a record/playback tool). When you test the system once in a while before a next major release, chances to re-use your artifacts are low.

It is opposite when the same system is tested again and again (as it should be in agile projects). It makes sense to invest in setting up automation. It rarely happened in traditional environments – even if you test each release, they are far apart and the difference between the releases prevents re-using the artifacts (especially with recorded scripts – APIs, for example, is usually more stable). So demand for automation was rather low and tool vendors didn't pay much attention to it. Well, the situation is changing – we may see more automation-related features in load testing tools soon.

While automation would take a significant role in the future, it addresses one side of the challenge. Another side of agile challenge is usually left unmentioned. The blessing of agile development – allowing to test the system early – highlights that for early testing we need another mindset and another set of skills and tools. Performance testing of new systems is agile and exploratory in itself and can't be replaced by automation (well, at least not in the foreseen future). Automation would complement it – together with additional input from development offloading performance engineers from routine tasks not requiring sophisticated research and analysis. But testing early – bringing most benefits by identifying problems early when the cost of their fixing is low – does require research and analysis, it is not a routine activity and can't be easily formalized.

It is similar to functional testing where both automated regression testing and exploratory testing are needed [BACH16] – with the difference that tools are used in performance testing in any case and setting up continuous performance testing is much more new and challenging.

The problem is that early performance testing requires a mentality change from a simplistic "record/playback" performance testing occurring late in the product life-cycle to a performance 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 and end-user requirements into component/unit-level requirements. You need to go from the record/playback approach to utilizing 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 "grey box", understanding the architecture of the system and how your load impact.

The concept of exploratory performance testing 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 through 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.

Continuous Integration

In more and more cases, performance testing should not be just an independent step of the software development life-cycle when you get the system shortly 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 a large number of tests repeatedly, resulting in the need for tools to support performance testing automation. 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.

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.

Automation means here not only using tools (in performance testing tools are used in most cases), but automating the whole process including setting up environment, running tests, and reporting / analyzing results. However “full performance testing automation” doesn’t look like a probable option in most cases. Using automation in performance testing helps with finding regressions and checking against requirements only – and it should fit the CI process (being reasonable in the length and amount of resources required). So large-scale, large-scope, and long-length tests would not probably fit, as well as all kinds of exploratory tests. What would be probably needed is a combination of shorter automated tests inside CI with periodic larger / longer tests outside or, maybe, in parallel to the critical CI path as well as exploratory tests.

While already mentioned above, cloud integration and support of new technologies are important for integration. Cloud integration, including automated deployment to public clouds, private cloud automation, and cloud services simplify deployment automation. Support of new technologies minimizes amount of manual work needed.

New Architectures

Cloud seriously impacts system architectures that has a lot of performance-related consequences.

First, we have a shift to centrally managed systems. ‘Software as a Service’ (SaaS) basically are centrally managed systems with multiple tenants/instances. Mitigating performance risks moves to SaaS vendors. From one side, it makes it easier to monitor and update / rollback systems that lowers performance-related risks. From another side, you get much more sophisticated systems when every issue potentially impacts a large number of customers, thus increasing performance-related risks.

To get full advantage of cloud, such cloud-specific features as auto-scaling should be implemented. Auto-scaling is often presented as a panacea for performance problems, but, even if it is properly implemented (which is, of course, better to be tested), it just assign a price tag for performance. It will allocate resources automatically – but you need to pay for them. And the question is how effective is the system – any performance improvement results in immediate savings.

Another major trend is using multiple third-party components and services, which may be not easy to properly incorporate into testing. The answer to this challenge is service virtualization, which allow to simulate real services during testing without actual access.

Cloud and virtualization triggered appearance dynamic, auto-scaling architectures, which significantly impact getting and analyzing feedback. System’s configuration is not given anymore and often can’t be easily mapped to hardware. 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 is the results of monitoring your system (such as response times, errors, and resources your system consumes).

The dynamic architectures represent a major challenge for both monitoring and analysis. It makes it difficult to analyze results as the underling system is changing all the time. Even before it often went beyond comparing results against goals – for example, when the system under test didn't match the production system exactly or when tests didn't represent the full projected load. It becomes even a much more serious challenge when configuration is dynamic – for both monitoring and analysis. Another challenge is when tests are a part of CI, where all monitoring and analysis should be done automatically. The more complex the system, the more important feedback and analysis become. A possibility to analyze monitoring results and test results together helps a lot.

Traditionally monitoring was on the system level. Due to virtualization system-level monitoring doesn't help much anymore and may be misleading – so getting information from application (via, for example, JMX) and database servers becomes very important. Many load testing tools recently announced integration with Application Performance Management / Monitoring (APM) tools, such as AppDynamics, New Relics, or 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.

With really dynamic architectures, we have a great challenge here to discover configuration automatically, collect all needed information, and they properly map the collected information and results onto changing configuration and system components in a way to highlight existing and potential issues, and, potentially, make automatic adjustments to avoid them. It would require very sophisticated algorithms (including machine learning) and potentially creates real Application Performance Management (the word "Management" today is rather a promise than the reality).

In additions to new challenges in monitoring and analysis, virtualized and dynamic architectures open a new applications for performance testing: to test if the system is dynamically changing under load in a way it is supposed to change.

New Technologies

Quite often the whole area of load testing is reduced to pre-production testing using protocol-level recording/playback. Sometimes it even lead to conclusions like "performance testing hitting the wall" [BUKSH12] just because load generation may be a challenge. While protocol-level recording/playback 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].

Well, the time when all communication between client and server was using simple HTTP is in the past and the trend is to provide more and more sophisticated interfaces and protocols. 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

This 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.

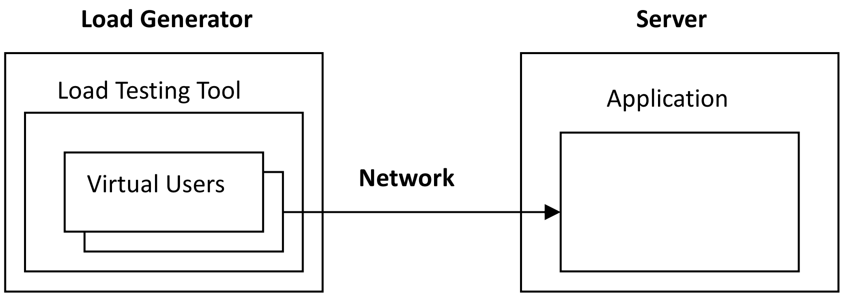

Fig.3 Record and playback approach, protocol level

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).

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].

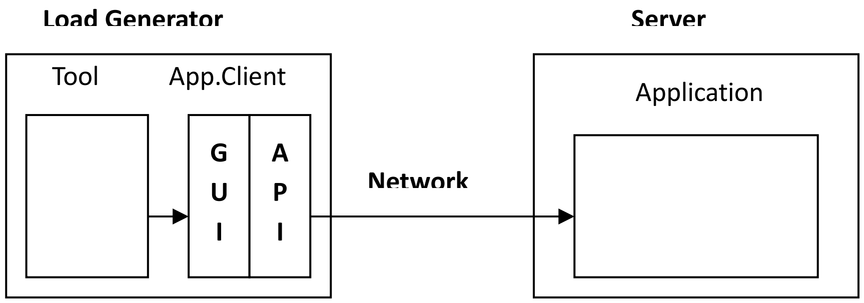

Fig.4 Record and playback approach, GUI users

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.

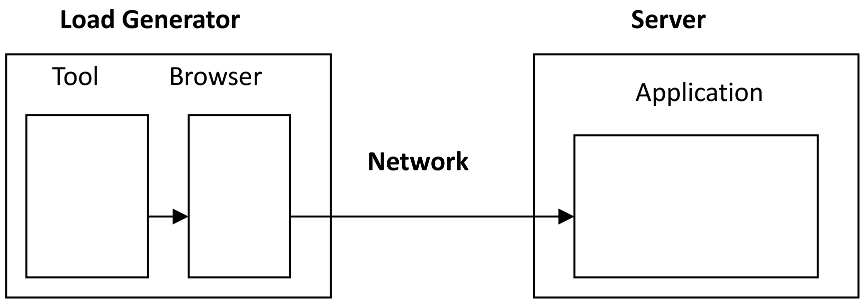

Fig.5 Record and playback approach, browser users

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.

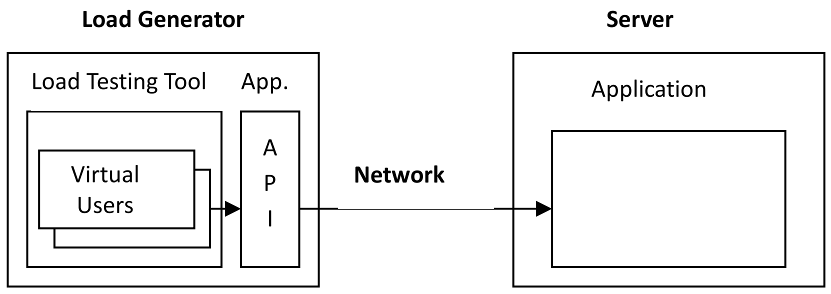

Fig 6. Programming API using a Load Testing Tool.

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.

The importance of API programming increases in agile / DevOps environments as tests are run often during the development process. In many cases APIs are more stable than GUI or protocol communication – and even if something changed, the changes usually can be localized and fixed – while GUI- or protocol-based scripts often need to be re-created.

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".

Summary

There are many more ways of doing performance testing than just the old, stereotypical approach of last-minute pre-production performance validation. While all these ways are not something completely new – the new industry trends push them into the mainstream.

The industry is rapidly changing (cloud, agile, continuous integration, DevOps, new architectures and technologies) – and to keep up performance testing should reinvent itself to become a flexible, context- and business-driven discipline. 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.

Performance testing should fully embrace [at last!] early testing (with exploratory approaches and “shift left” to performance engineering) as well as agile / iterative development (with regressions performance testing and getting to the next level of automation).

Good tools can help you here – and not so good tools may limit you in what you can do. And getting so many options in performance testing, 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

[BACH16] Bach, J., Bolton, M. A Context - Driven Approach to Automation in Testing. 2016.
http://www.satisfice.com/articles/cdt-automation.pdf

[BARB11] Barber, S. Performance Testing in the Agile Enterprise. STP, 2011.
http://www.slideshare.net/rsbarber/agile-enterprise

[BUKS12] Buksh, J. Performance Testing is hitting the wall. 2012.
http://www.perftesting.co.uk/performance-testing-is-hitting-the-wall/2012/04/11/

[CRIS09] Crispin, L., Gregory, J. Agile Testing: A Practical Guide for Testers and Agile Teams. Pearson Education, 2009.

[EIJK11] Eijk, P. Cloud Is the New Mainframe. 2011.
http://www.circleid.com/posts/cloud_is_the_new_mainframe

[HAZR11] Hazrati, V. Nailing Down Non-Functional Requirements. InfoQ, 2011.
http://www.infoq.com/news/2011/06/nailing-quality-requirements

[HAWK13] Andy Hawkes, A. When 80/20 Becomes 20/80.
http://www.speedawarenessmonth.com/when-8020-becomes-2080/

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

[MOLU14] Molyneaux, I. The Art of Application Performance Testing. O'Reilly, 2014.

[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] Podelko, A. Load Testing: See a Bigger Picture. CMG, 2012.

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

[PODE14a] Podelko, A. Load Testing at Netflix: Virtual Interview with Coburn Watson. 2014.
http://alexanderpodelko.com/blog/2014/02/11/load-testing-at-netflix-virtual-interview-with-coburn-watson/

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

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

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