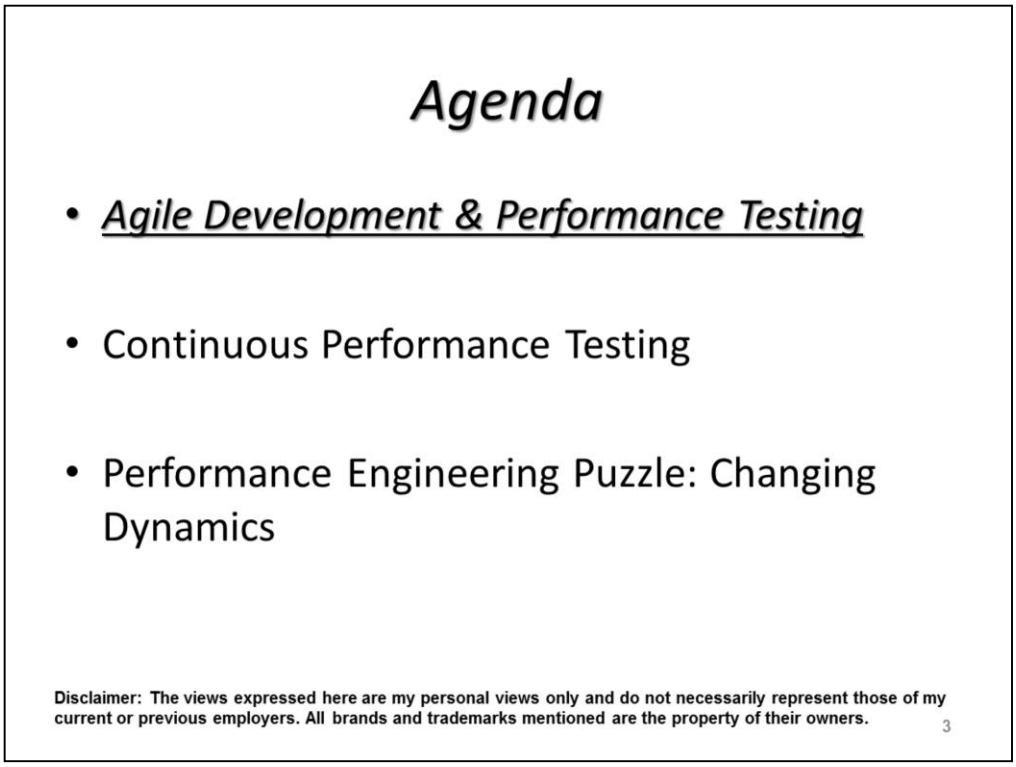
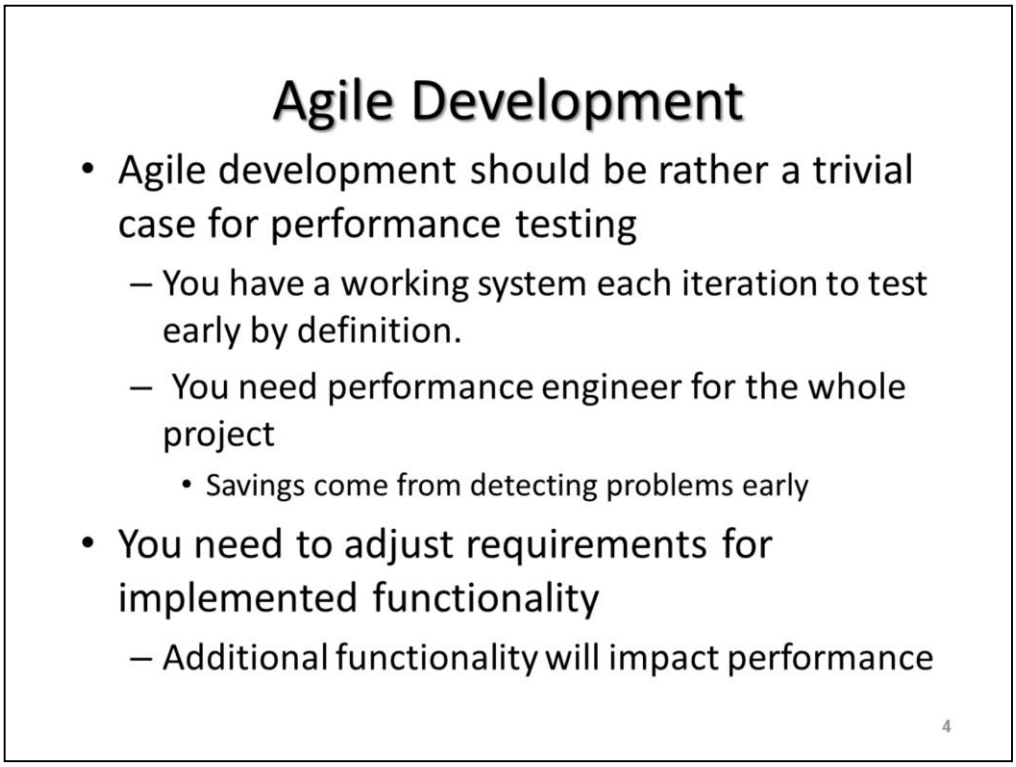
While development process is moving towards all things continuous, performance testing remains rather a gray area. Some continue to do it in the traditional pre-release fashion, some claim 100% automation and full integration into their continuous process. We have a full spectrum of opinions of what, when, and how should be done in regard to performance. The issue here is that context is usually not clearly specified - while context is the main factor here. Depending on context, the approach may (and probably should) be completely different. Full success in a simple (from the performance testing point of view) environment doesn't mean that you may easily replicate it in a difficult environment. The speaker will discuss the issues of making performance testing continuous in detail, illustrating them with personal experience when possible.

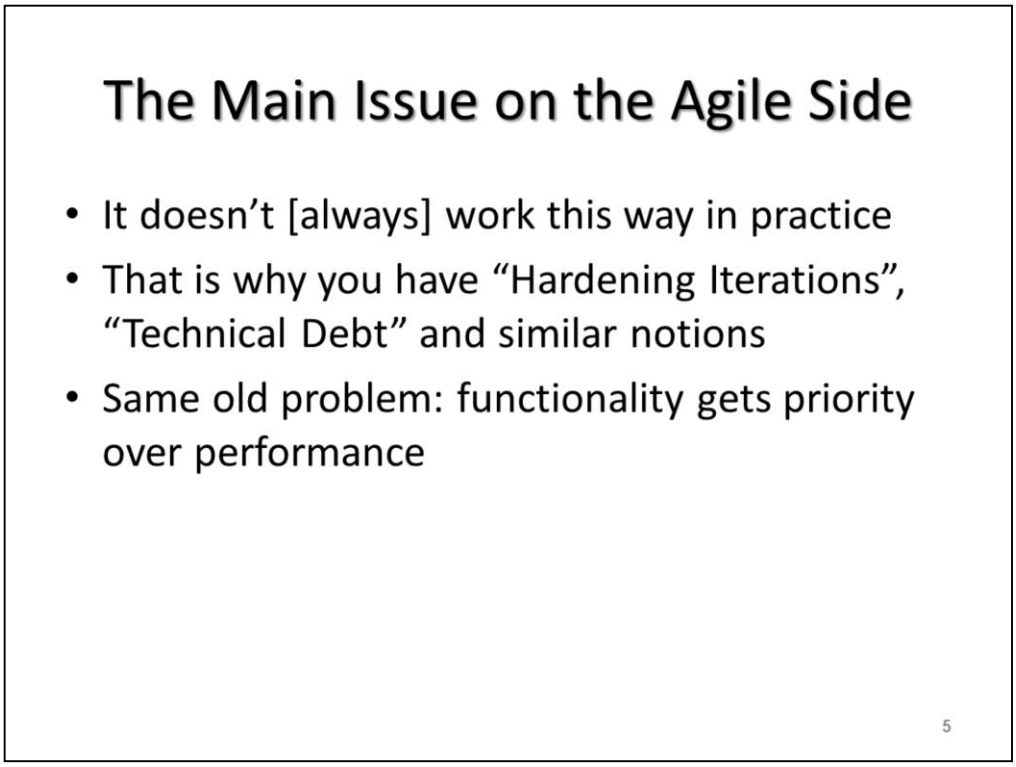
The traditional, stereotypical way of doing load testing is running few tests at the last moment before rolling out the system in production without much instrumentation. It made sense for waterfall development – as not much was available for performance testing until the last moment anyway. And yes, we still need all skills and techniques of traditional performance testing – but now we need much more.



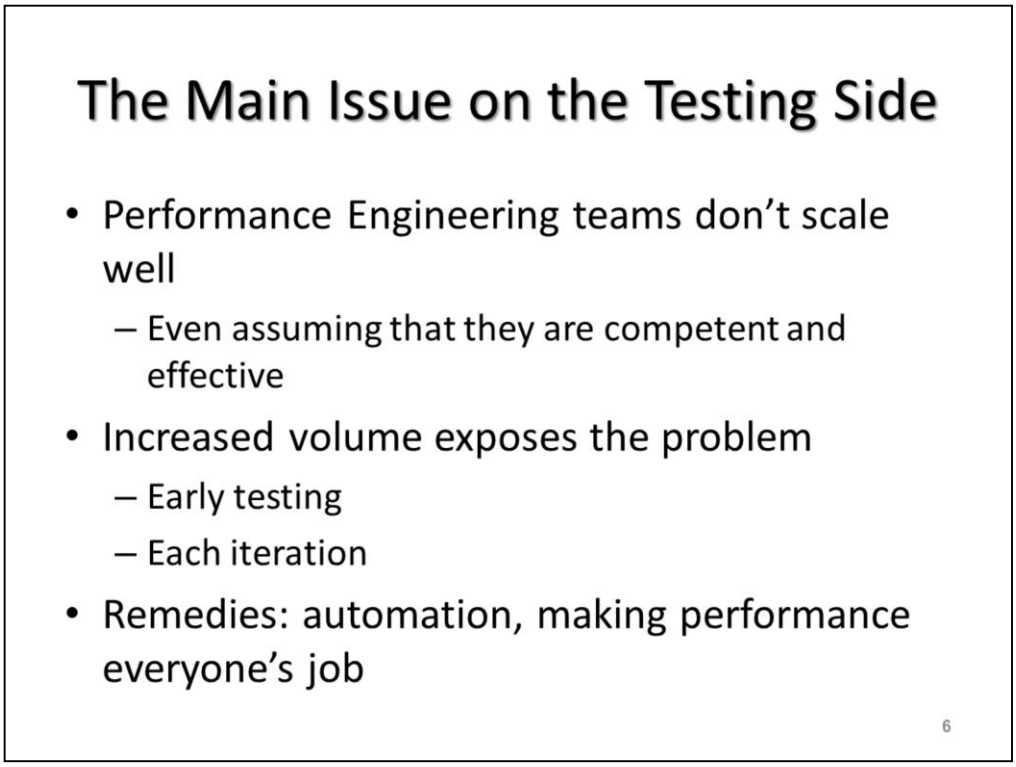
The need in continuous performance testing is coming from needs of agile / iterative development. So it is important to understand what changes in performance testing are triggered by agile development in general – and only then to see where continuous performance testing gets into the picture.



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.

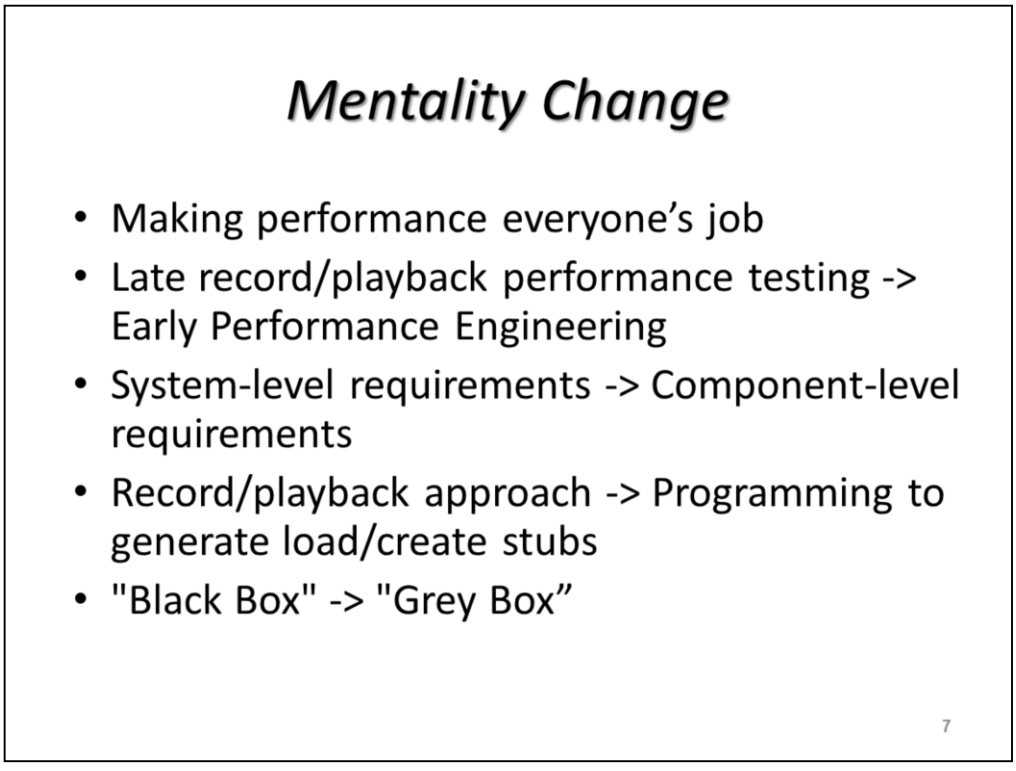


From the agile development side the problem is that, unfortunately, it doesn't always work this way in practice. So such notions as "hardening iterations" and "technical debt" get introduced. Although it is probably the same old problem: functionality gets priority over performance (which is somewhat 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.

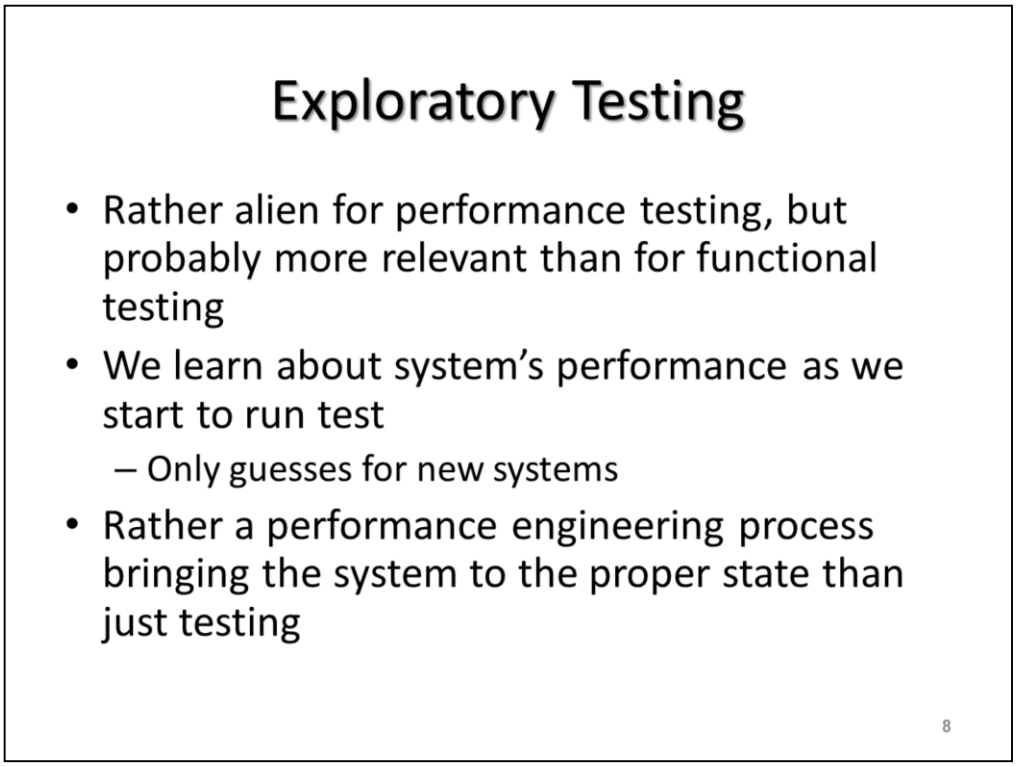


From the performance testing side the problem is that performance engineering teams don't scale well, even assuming that they are competent and effective. At least not in their traditional form. They work well in traditional corporate environments where they check products for performance before release, 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.

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

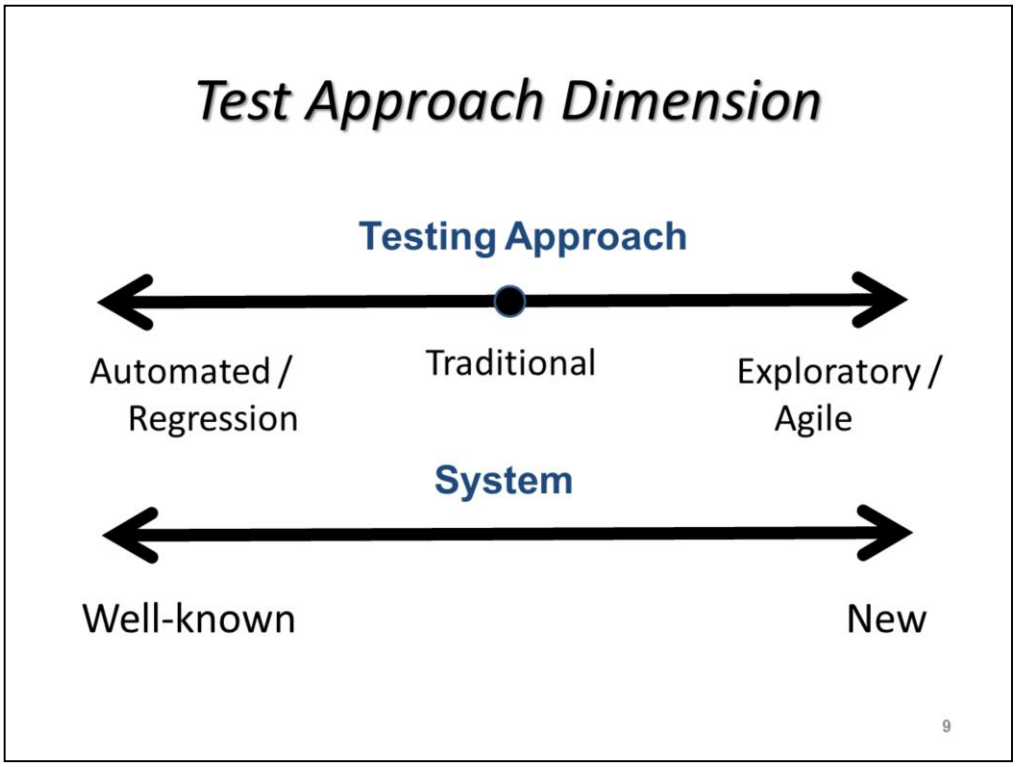


The problem is that early performance testing requires a mentality change from a simplistic "record/playback" performance testing occurring late in the product life-cycle to a performance engineering approach starting early in the product life-cycle. You need to translate "business functions" performed 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. And for all these you need to get all stakeholders involved - making performance everyone's job.



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 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 – 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.

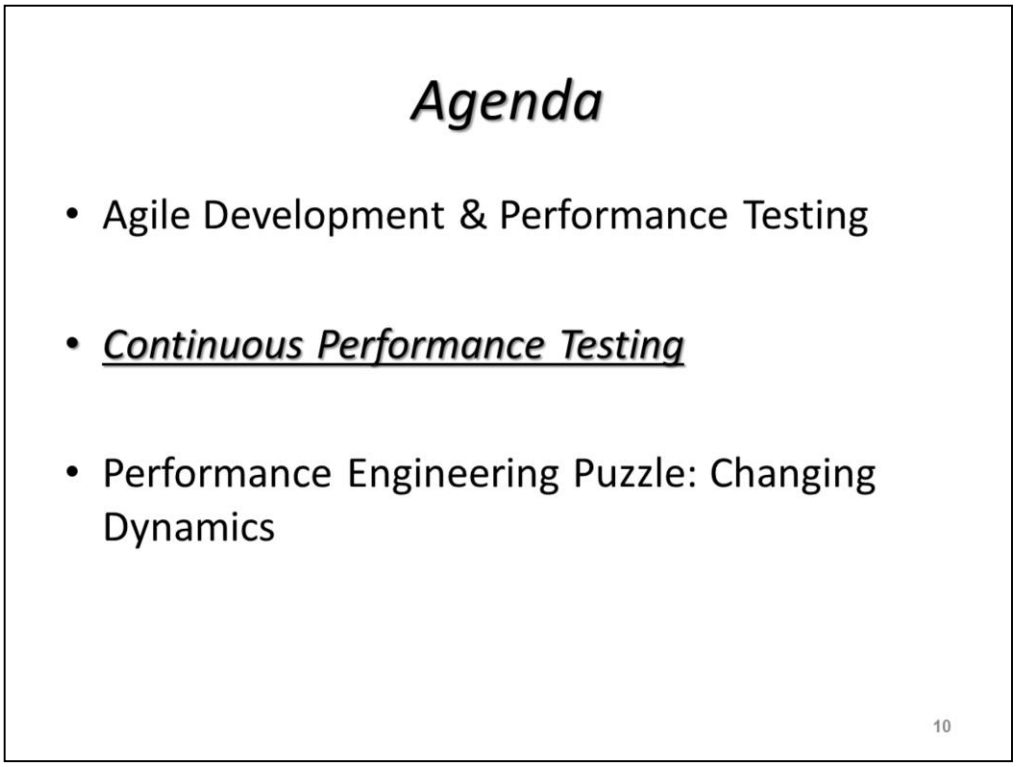
The concept of exploratory performance testing is still rather alien. But the notion of exploring is much more important for performance testing than for functional testing. Functionality of systems is usually more or less defined (whether it is well documented is a separate question) and testing boils 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.



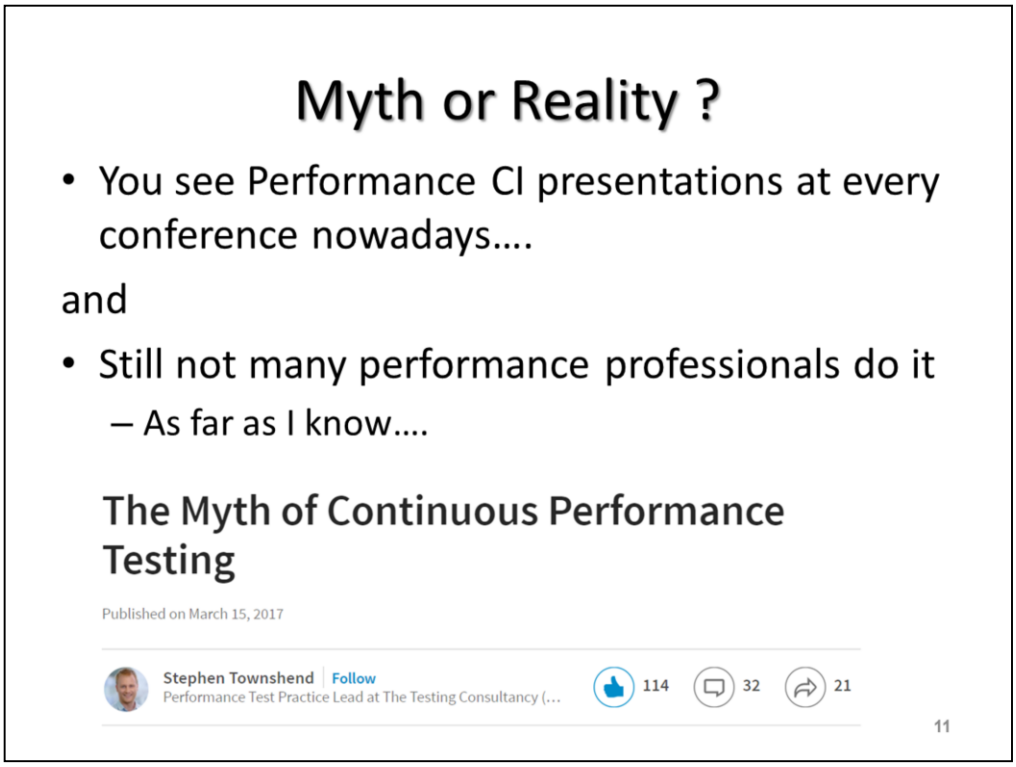
If we have the testing approach dimension, the opposite of exploratory would be regression testing. We want to make sure that we have no regressions as we modify the product – and we want to make it quick and, if possible, automatic. And as soon as we get to an iterative development process where we 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).

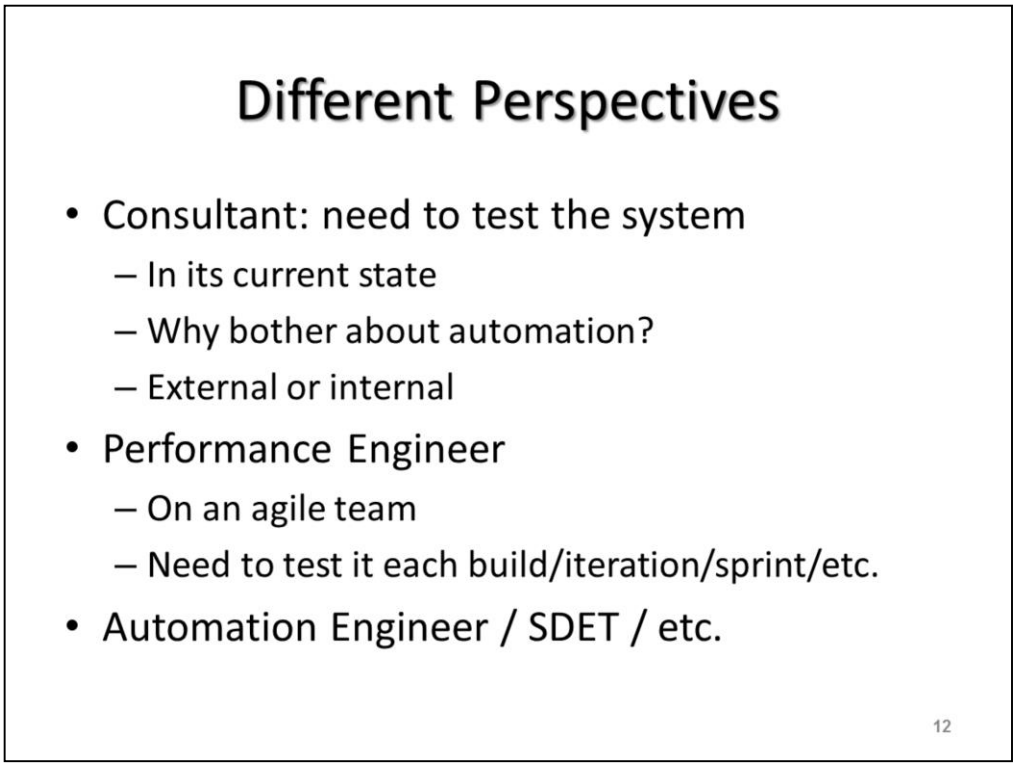
If we see the continuum this way, the question which kind of testing is better looks completely meaningless. You need to use the right combination of approaches for your system in order to achieve better results. Seeing the whole testing continuum between regression and exploratory testing should help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



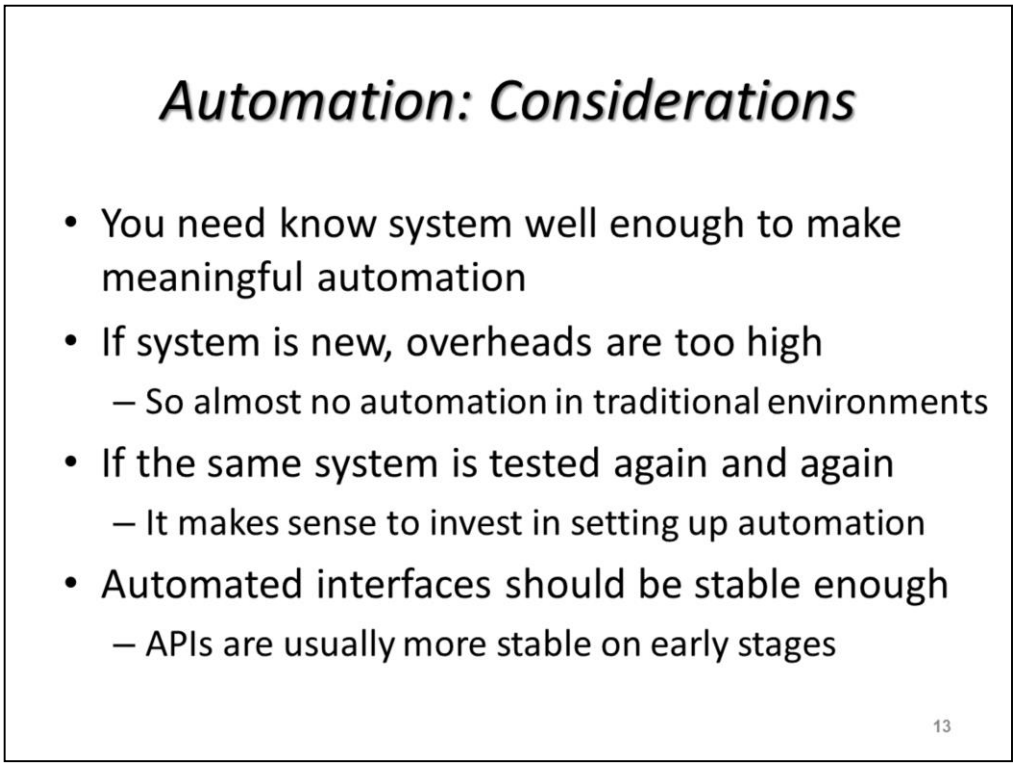
Now let's look into performance testing automation and continuous performance testing.



You see Performance CI presentations at every conference nowadays (for example, at Velocity conferences). Just here at CMG imPACt we have several presentations exactly about this topic. It creates impression that it is a common practice and everybody is doing it (see, for example, [PRAT15]). However still not many performance professionals do it as far as I know – and some even speak up about the topic – as Stephen Townshend in The Myth of Continuous Performance Testing [TOWN17].



Point of view depends on who you are. If you need to verify performance of the system only – why would you bother with automation? If you need to do it each build/iteration/sprint/etc. you start to think about it even knowing what is involved. If you are Automation Engineer / SDET / etc. you do something as it sounds natural to you - and then elaborate what you got…



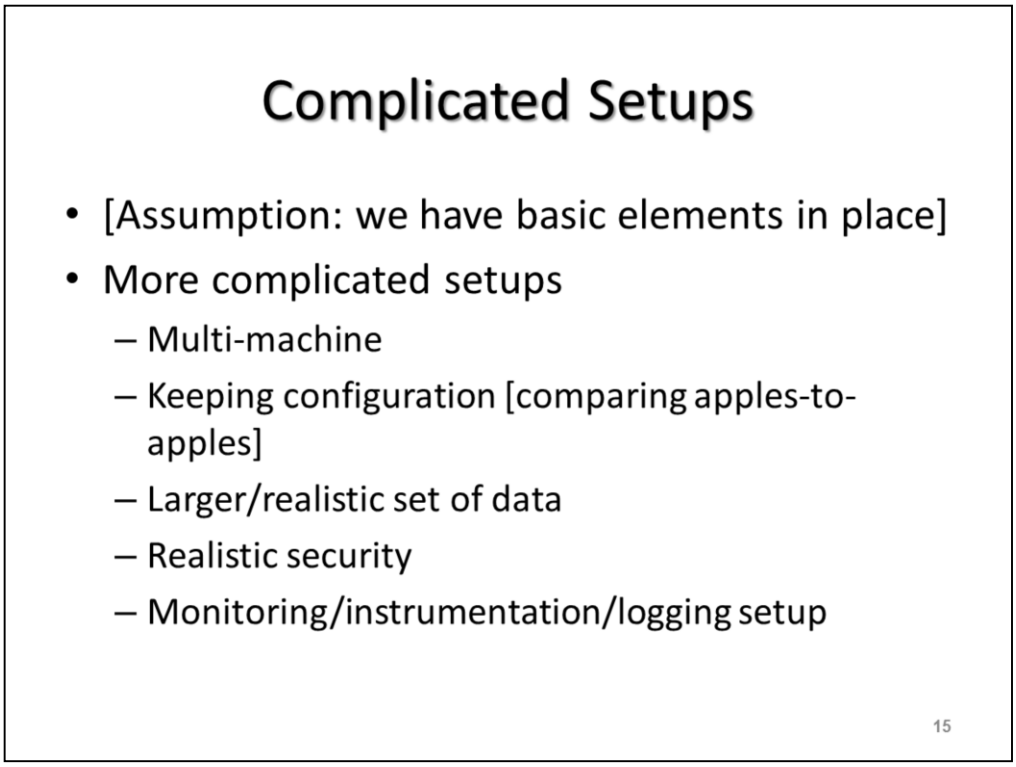
First, we need to understand when automation is really needed – and why we practically didn't have any performance automation in traditional environments. You need know system well enough to make meaningful automation. If system is new, overheads are too high. Automated interfaces should be stable enough (although APIs are usually more stable on early stages). If the same system is tested again and again it does make sense to invest in setting up automation.

Automation: Difficulties

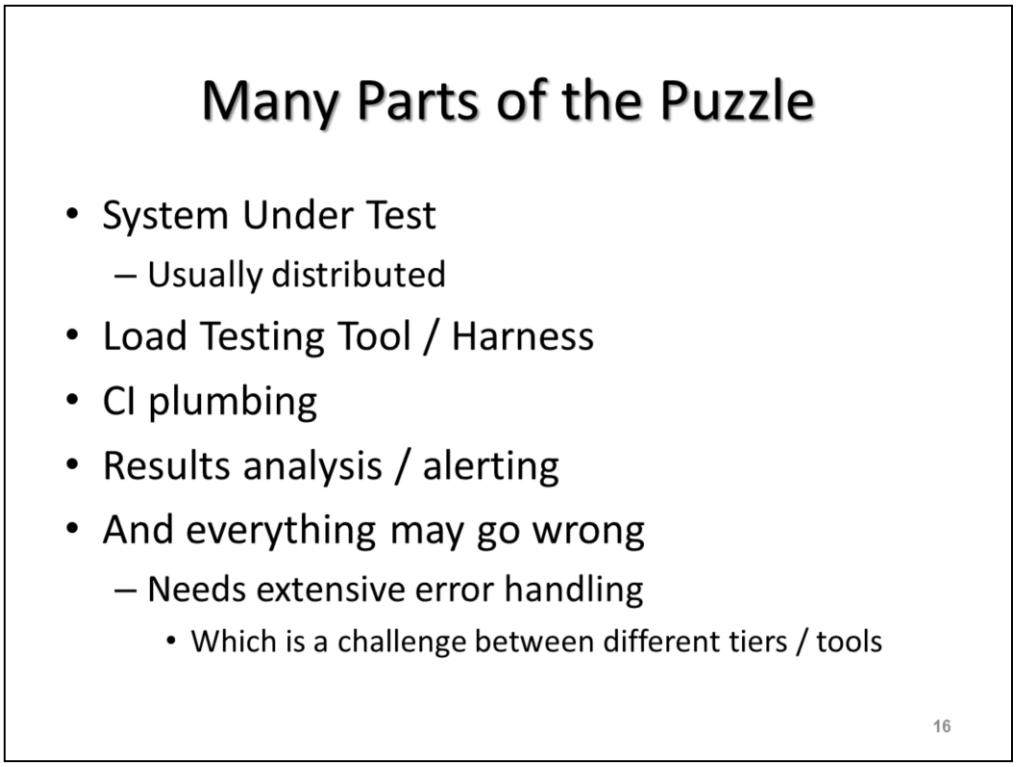
- Complicated setups
- Many parts of the puzzle
  - Long list of possible issues
- Complex results (no pass/fail)
  - Not easy to compare two result sets
- Changing/Fragile Interfaces
- Time / Resources considerations
  - Tests may be long / use a lot of resources

14

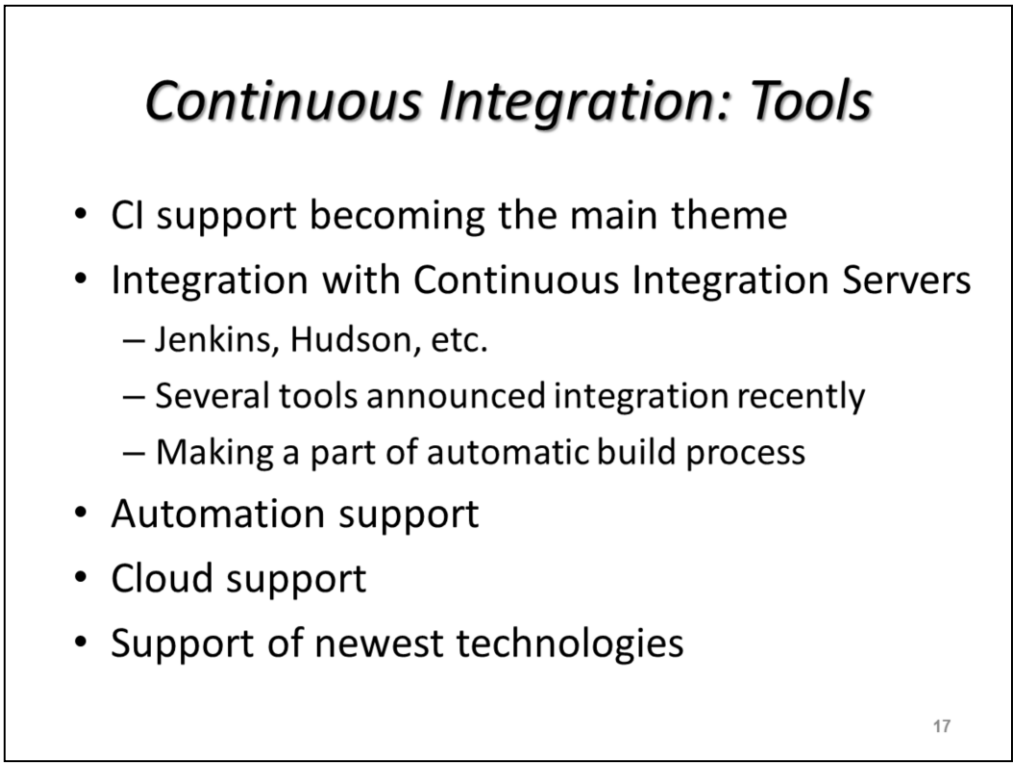
All issues are real and serious – but how challenging is every one depends heavily on context. Let's consider all these concerns separately.



Usually involving performance testing into continuous integration happens when other elements are in place – automatic builds, basic automated deployment and configuration, functional testing. If it is not so, it is a completely different challenge. Still, even everything is in place, it may require significant enhancements for performance testing – such as multi-machine deployments, making sure that configuration stays the same (in whatever way it make sense for the system), deployment meaningful sets of data and security configuration, getting all needed monitoring / instrumentation / logging in place.



We have many parts of the puzzle here – more than in functional testing – and the need to work smooth together. We have the system under test (usually distributed), a load testing tool or harness, CI tools / plumbing, results analysis / alerting… And everything may go wrong – so it needs extensive error handling, which is a challenge between different tiers / tools.

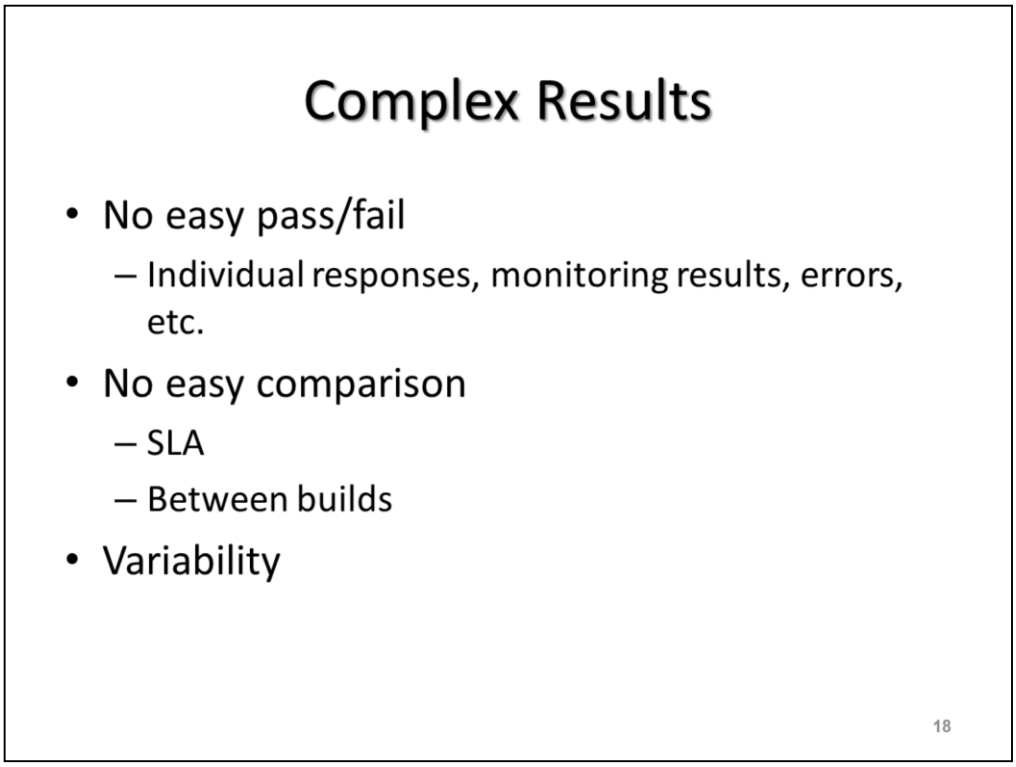


In more and more cases, performance testing should not be just an independent step of the software development life-cycle when you get the system shortly before release. In agile development / DevOps environments it should be interwoven with the whole development process. There are no easy answers here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

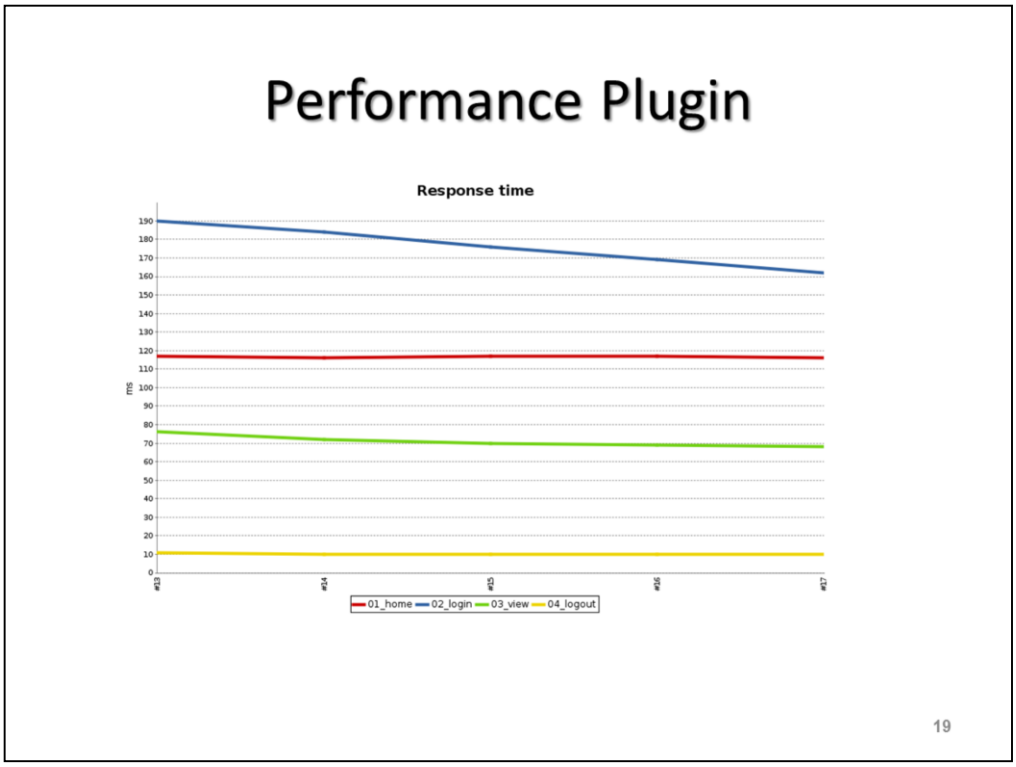
What makes agile projects really different is the need to run a large number of tests repeatedly, resulting in the need for tools to support performance testing automation. The situation started to change recently as agile support became the main theme in load testing tools [LOAD14]. Several tools recently announced integration with Continuous Integration Servers (such as Jenkins or Hudson). While initial integration may be limited, 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.



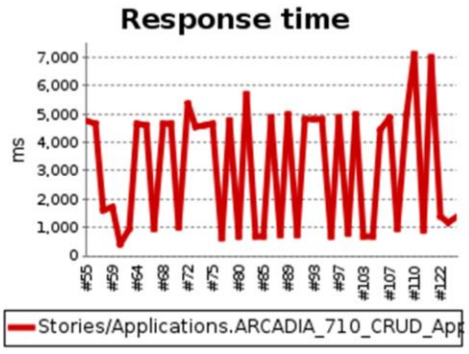
Most load testing tools compare results to SLAs – but it is not very useful for continuous integration when we want to see the change.



Even if there is some comparisons – they are rather simple. Here is an example for Jenkins Performance Plugin – taking input from JMeter, Junit, Taurus and few more tools. It is easy to get something if you have Jenkins and Jmeter – you will find quite a few instructions how to do it. But any step you do further – you are basically on your own. It would probably result at least in some sophisticated plumbing, if not significant custom development.

Variability

- Inherent to test / environment
- Due to difference in environments



20

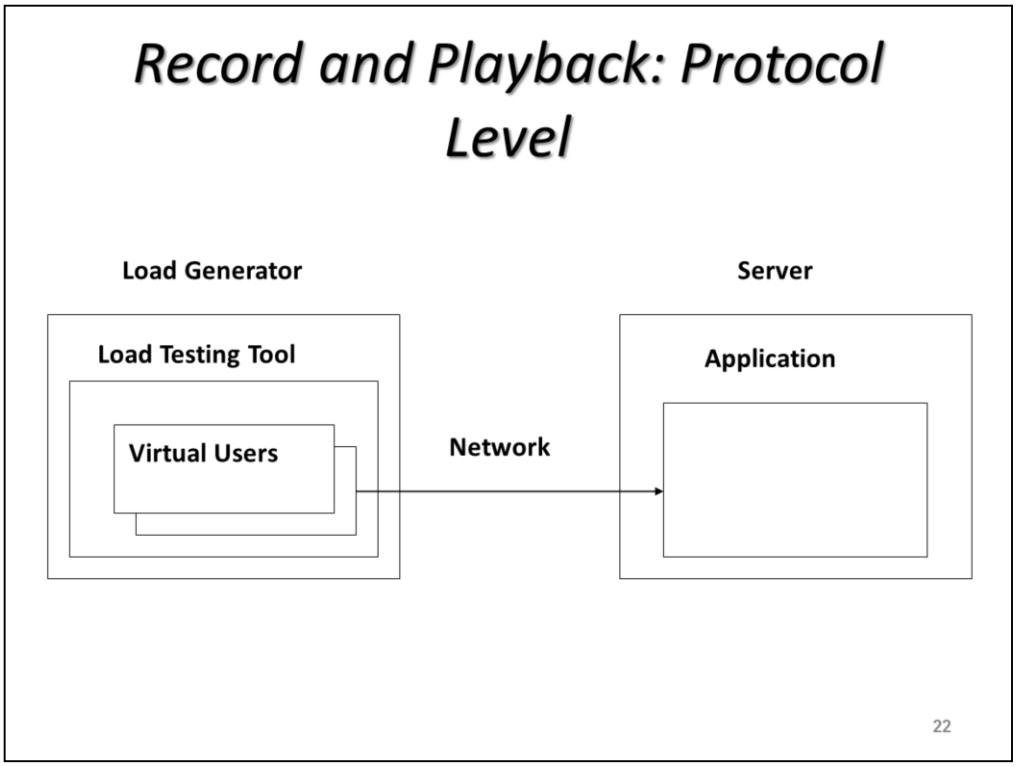
We have two parts here: variability inherent to test / environment (which we need to separate from changes in performance) and variability due to difference in environments (which is quite often is the case when environments are allocated automatically). Here is a rather extreme case when app server environment may be allocated in the same or in another data center as database environment (so we see the difference between about 1 sec and about 5-7 sec).

Changing Interfaces

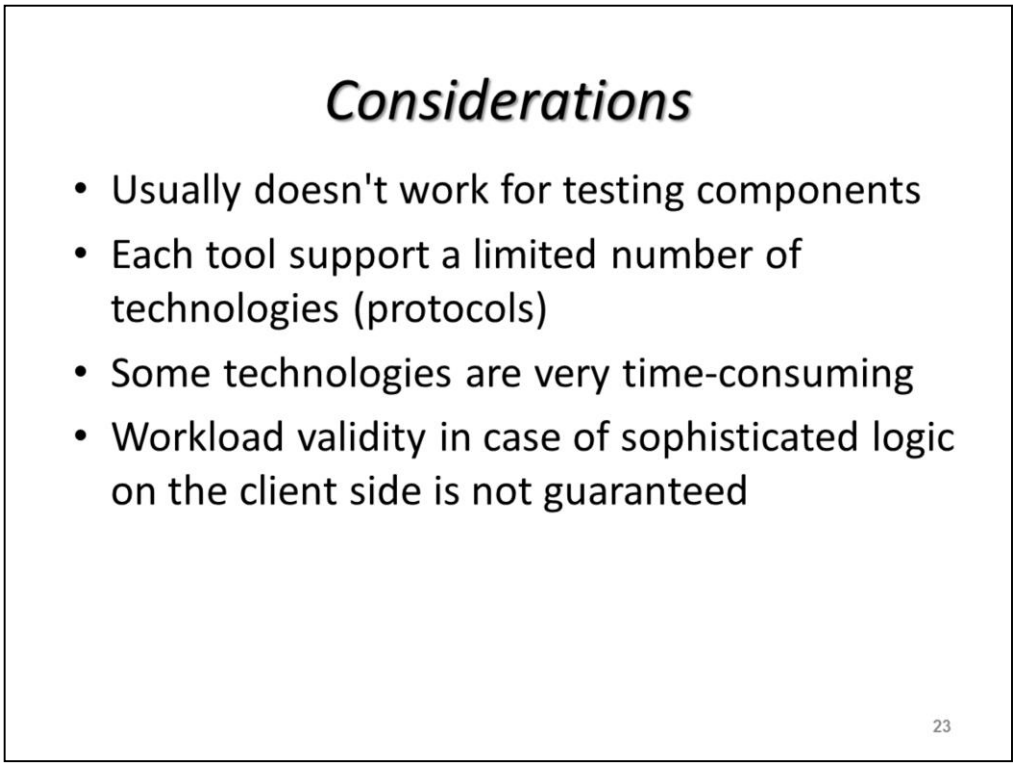
- If use recording, minor changes may break scripts
  - And you may even don't know that
  - Interface should be mature enough
- Not just protocol-level recording
  - GUI
  - API / Programming

21

Traditional approach to performance testing – recording/playback on the protocol level (and on GUI level too) - is notoriously prone to change / fragile, especially during early stages of system's lifecycle. It adds a lot of overheads maintaining the scripts and the need to add sophisticated logic to avoid false negative and positive results, especially in case of Continuous Integration. Using APIs is usually more robust when you have APIs available and know well how it is used – but it often not the case.

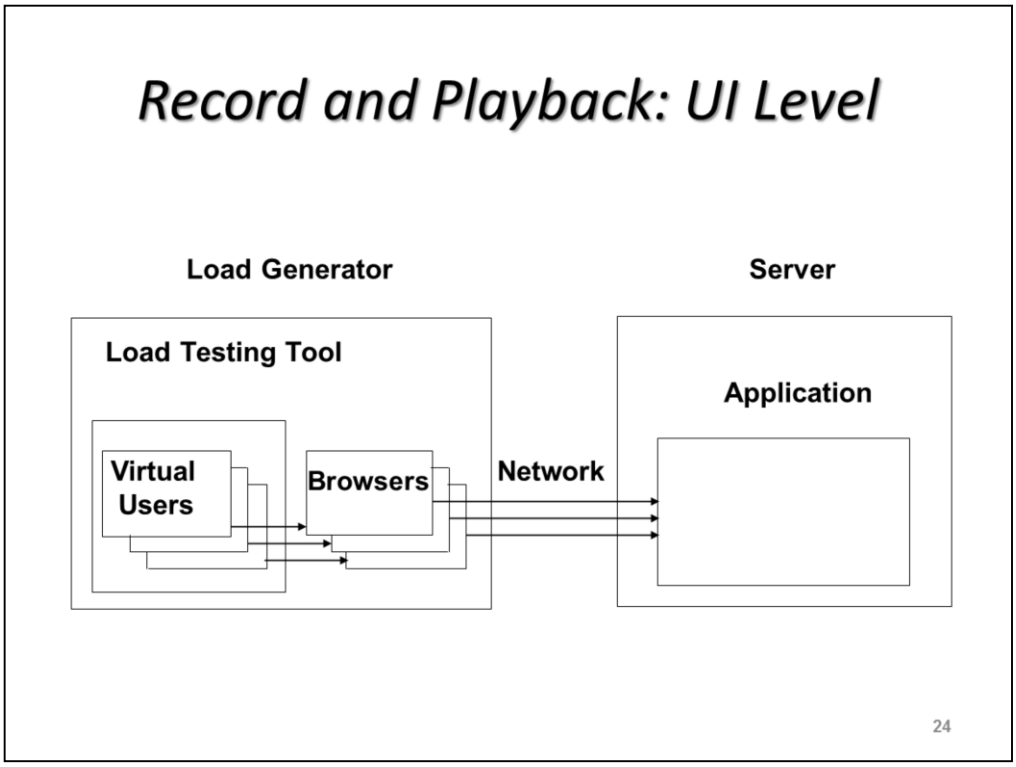


Record and playback on the protocol level is the mainstream approach to load testing: recording communication between two tiers of the system and playing back the automatically created script (usually, of course, after proper correlation and parameterization). As far as no client-side activities are 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.

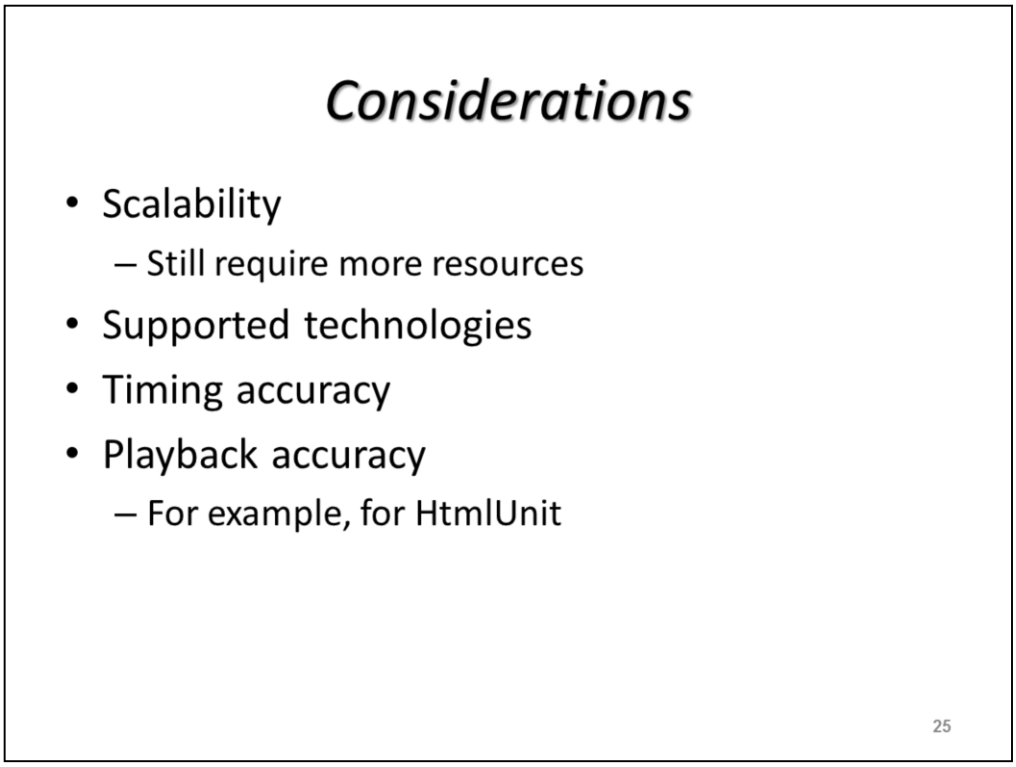
With quick internet growth and the popularity of browser-based clients, most products support only HTTP or a few select web-related protocols. To the author's knowledge, only MicroFocus LoadRunner and SilkPerformer try to keep up with support for all popular protocols (other products claiming support 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).



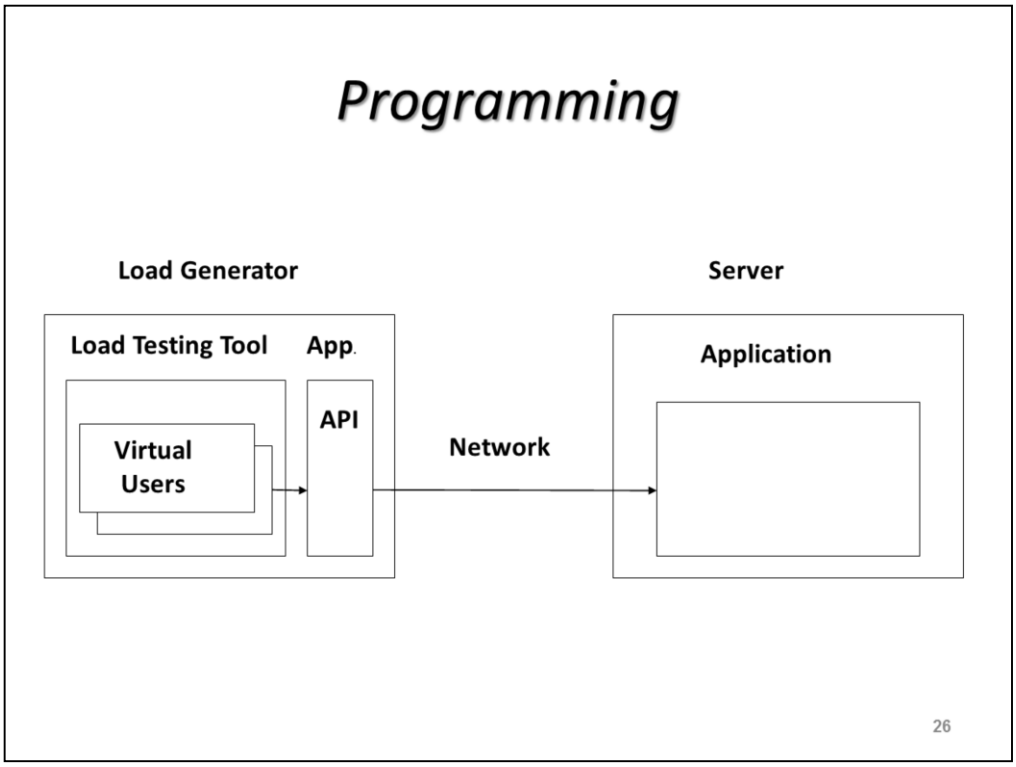
This option has been available for a long time, but it is much more viable now. For example, it was possible to use Mercury/HP WinRunner or QuickTest Professional (QTP) scripts in load tests, but a separate machine was needed for each virtual user (or at least a separate terminal session). This drastically 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.

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.

Today there are multiple tools supporting this approach, such as Appvance, which directly harnesses Selenium and HtmlUnit for load testing, or LoadRunner TruClient protocol, which use proprietary solutions to achieve low-overhead playback.

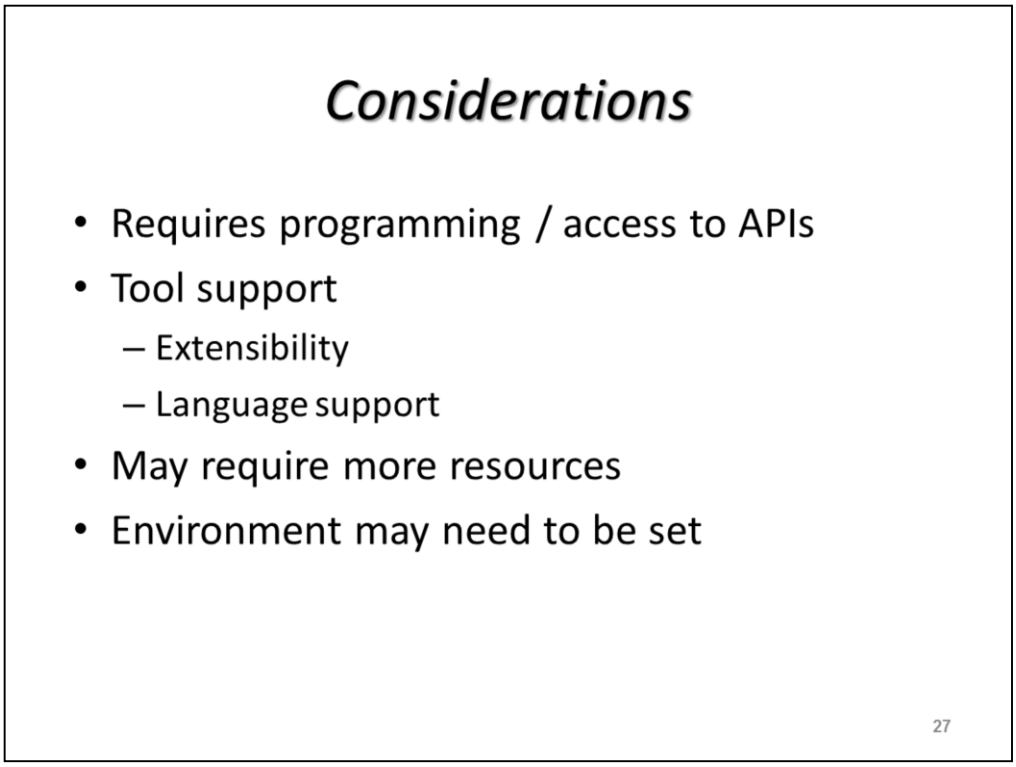


Nevertheless, questions of supported technologies, scalability, and timing accuracy remain largely undocumented, so the approach requires evaluation in every specific case.



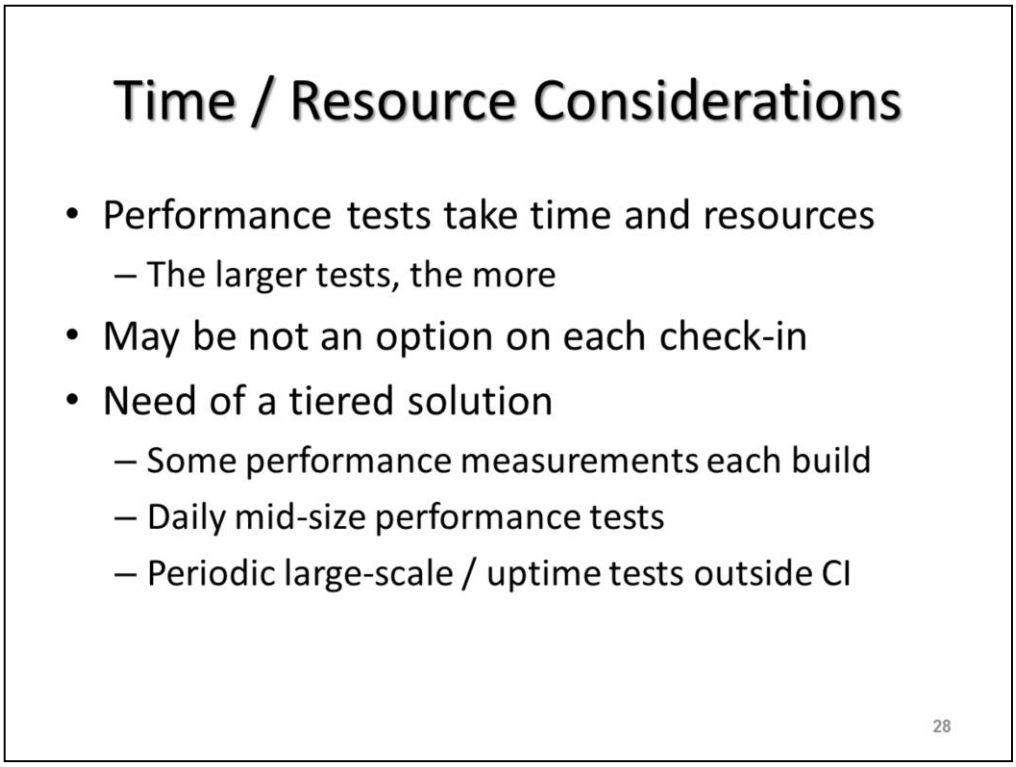
There are cases when recording can't be used at all, or when it can, but with great difficulty. In such cases, API calls from the script may be an option. Often it is the only option for component performance testing. Other variations of this approach are web services scripting or use of unit testing 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.

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.

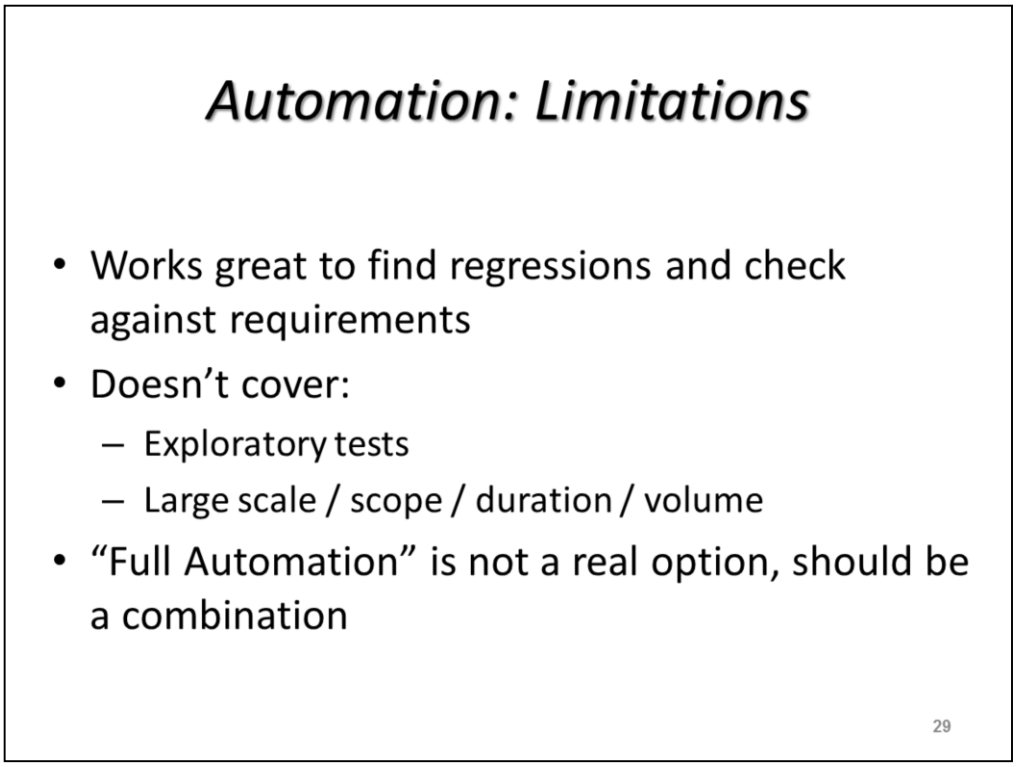


To use a load testing tool with APIs, it should have the ability to add code to (or invoke code from) your script. And, of course, if the tool's language is different from the language of your API, you would need to figure out a way to plumb them. Tools, using standard languages such as C (e.g. LoadRunner) or Java (e.g. Oracle Application Testing Suite) may have an advantage here.

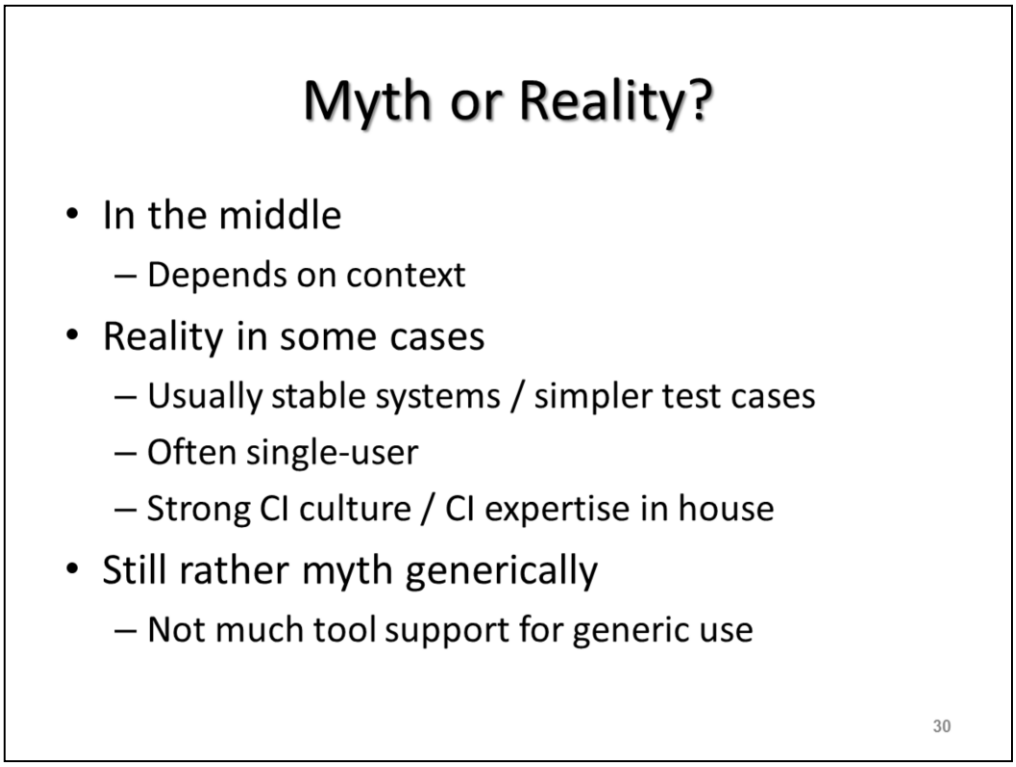
In any case, you need to understand all of the details of the communication between client and server to use the right sequences of API calls; this is often the challenge.



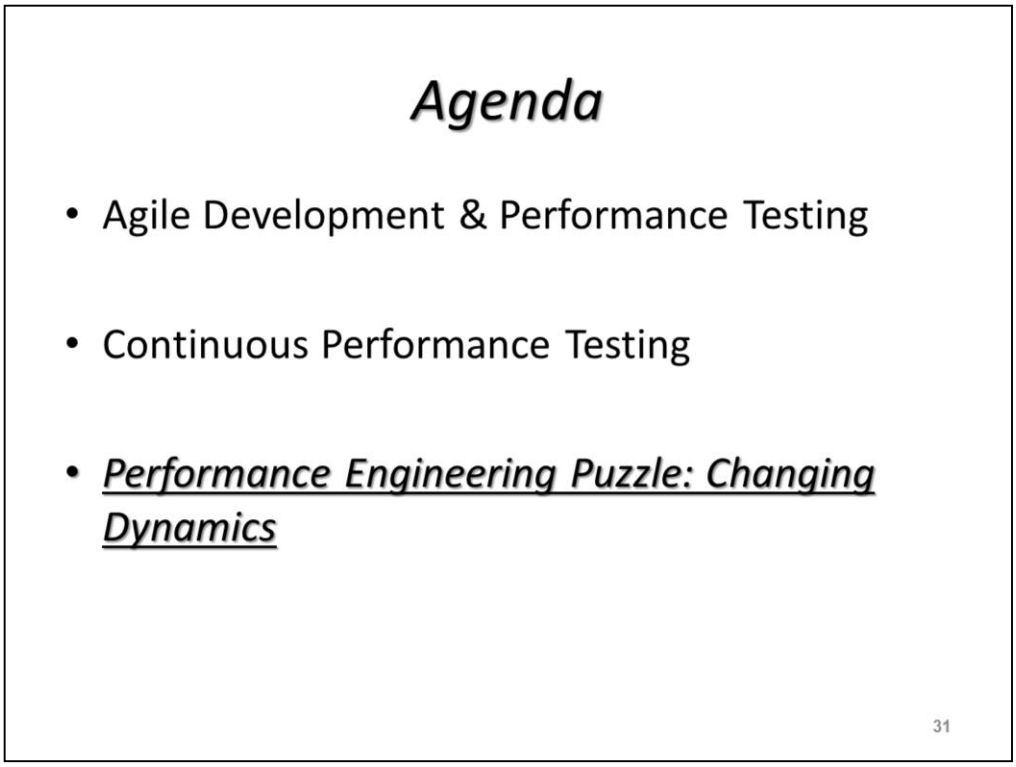
Performance tests take time and resources. The larger tests, the more. Running full-scale realistic tests on each check-in is usually not an options. So a tiered solution is probably needed – and details, of course, would depend on context. It may be, for example:
-Some simple performance measurements each build.
-Daily mid-size performance tests.
-Periodic large-scale / uptime tests outside CI.



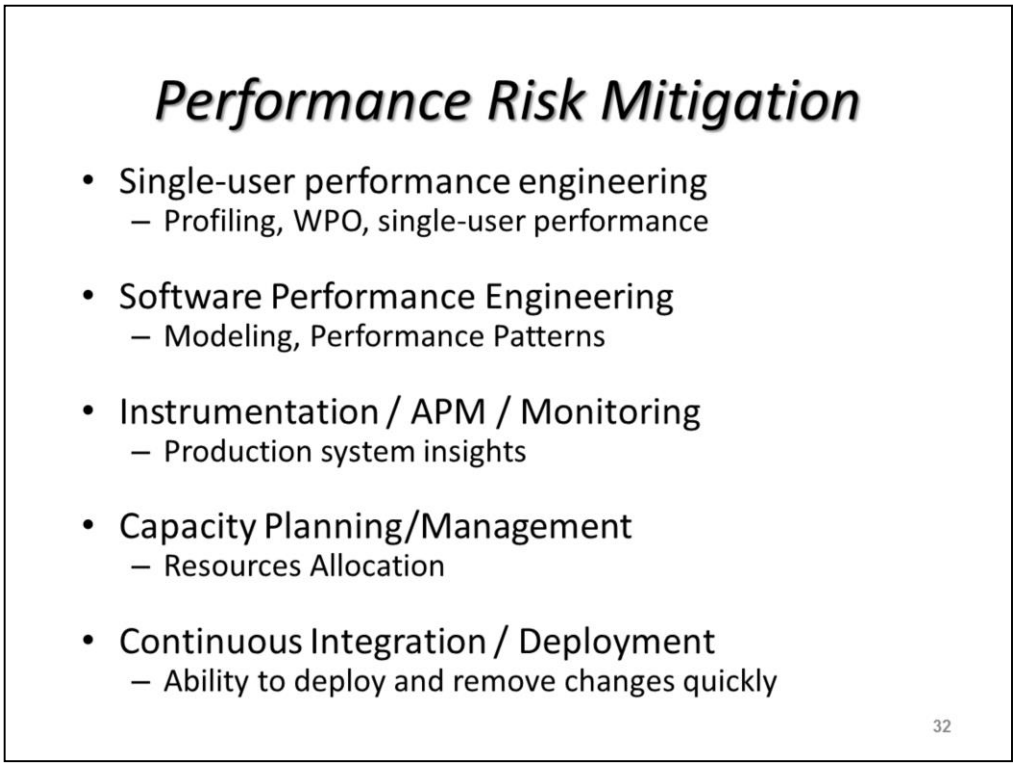
Automation means here not only using tools (in performance testing tools are used in most cases), but automating the whole process including setting up environment, running tests, and reporting / analyzing results. However "full performance testing automation" doesn't look like a probable option in most 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.



The truth is, as usual, in the middle. The answer heavily depends on context. It is a reality in some cases: usually stable systems / simpler test cases; often single-user; in places with strong CI culture / CI expertise in house. But it is still rather myth generically as there is not much tool support for generic use. It is changing, but it still a long way before it becomes easy and straightforward….



To properly place continuous performance testing, it may be worth time to discuss the place of performance testing in general in performance engineering.



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.

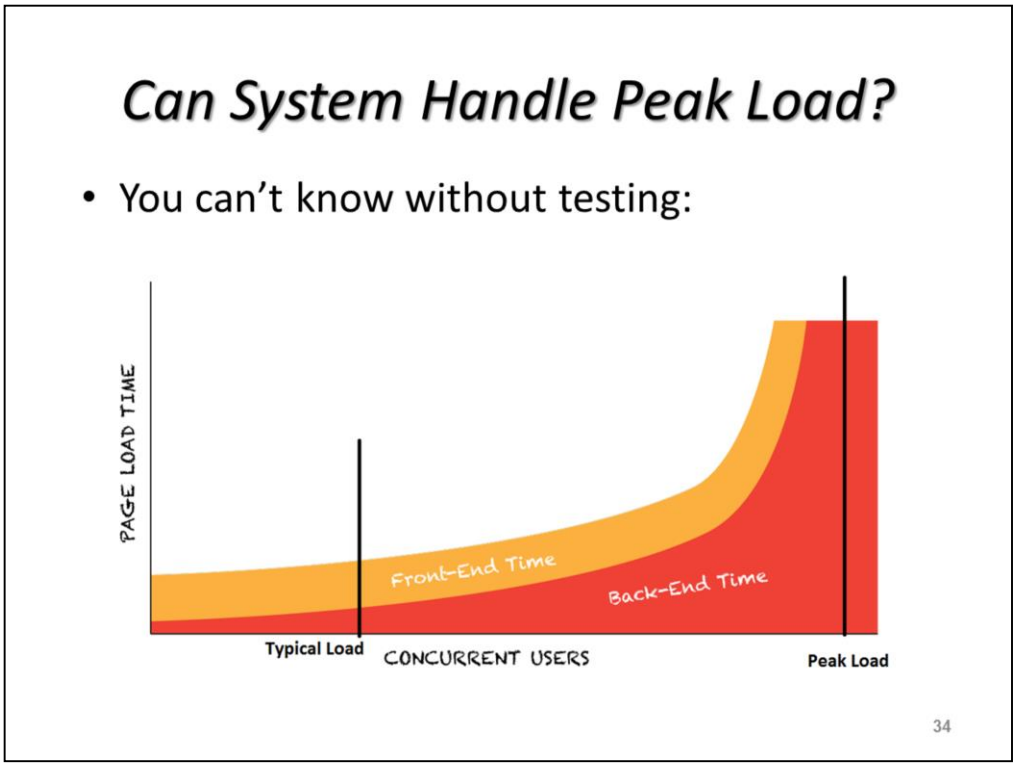
Load Testing. Everything used for testing the system under any multi-user load (including all other variations of multi-user testing, such as performance, concurrency, stress, endurance, longevity, scalability, reliability, and similar).

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.

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?



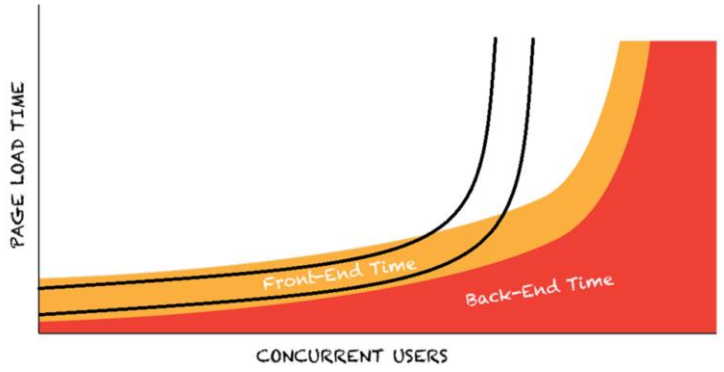
There are always risks of crashing a system or experiencing performance issues under heavy load – and the only way to mitigate them is to actually test the system. Even stellar performance in production and a highly scalable architecture don't guarantee that it won't crash under a slightly higher load.

A typical response time curve is shown on the slide, 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).

Verify Multi-User Performance

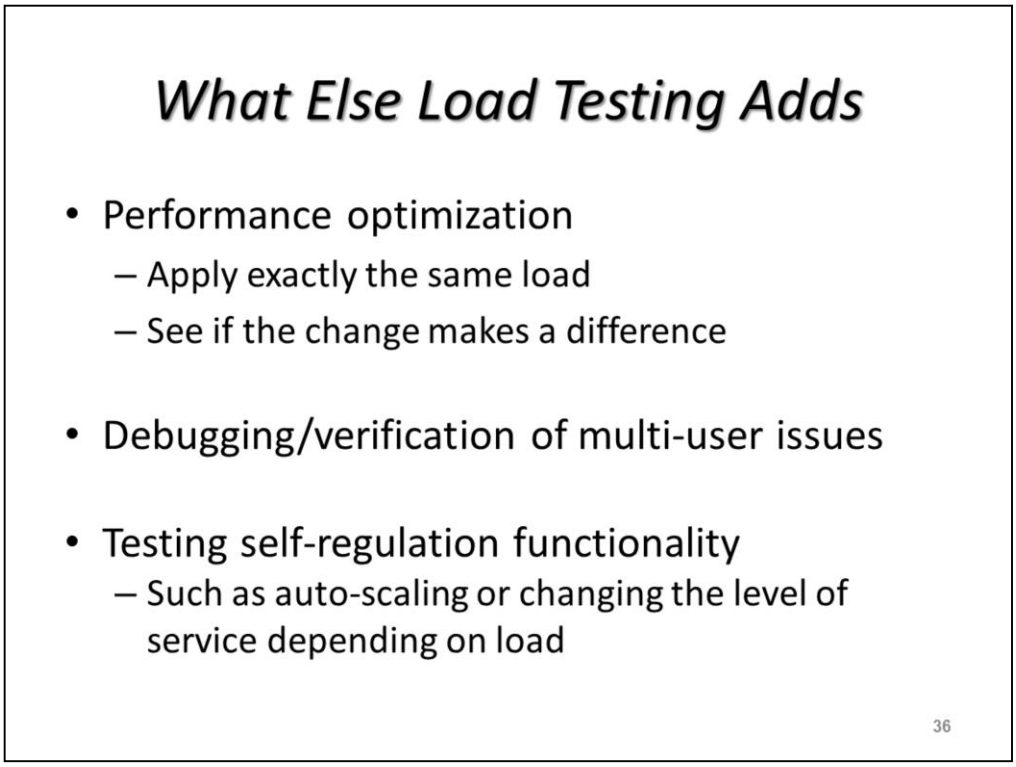
- Single-user improvement may lead to multi-user performance degradation



35

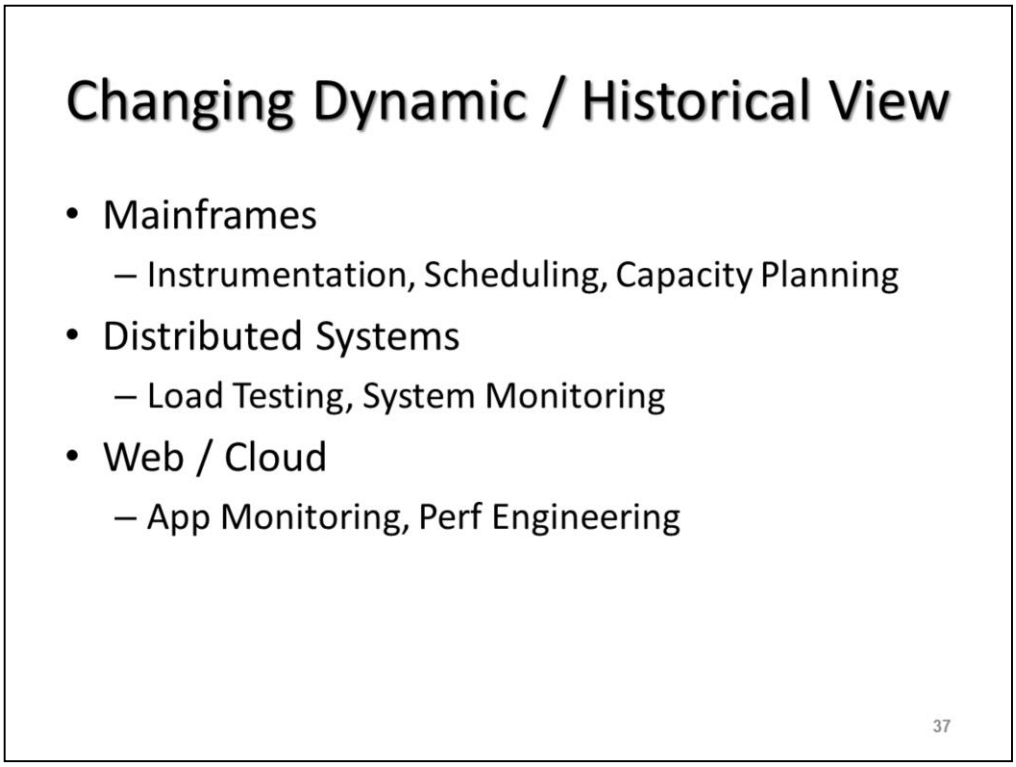
Another important value of load testing is checking how changes impact multi-user performance. The impact on multi-user performance is not usually proportional to what you see with single-user performance and often may be counterintuitive; sometimes single-user performance improvement may lead to multi-user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

As it can be seen on the slide, 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.



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.

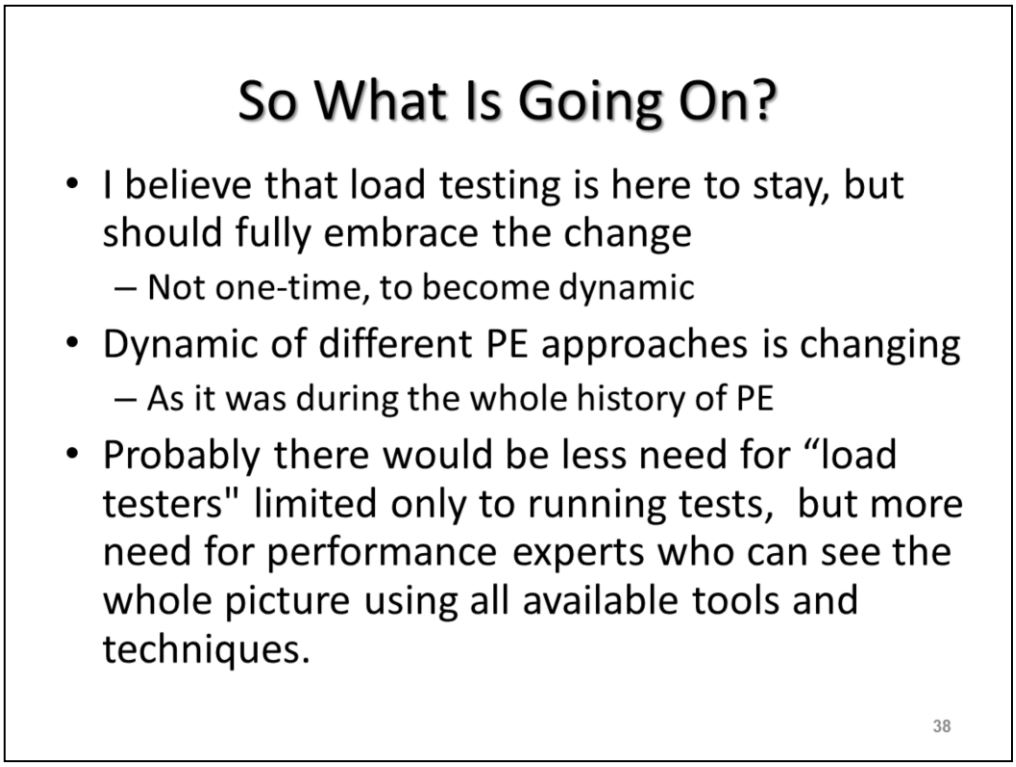
Moreover, with existing trends of system self-regulation (such as auto-scaling or changing the level of services depending on load), load testing is needed to verify that functionality. You need to apply heavy load to see how auto-scaling will work. So load testing becomes a way to test functionality of the system, blurring the traditional division between functional and nonfunctional testing.



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]. Netflix was very successful in using canary testing –the performance testing that uses real users to create load instead of creating synthetic load. 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.

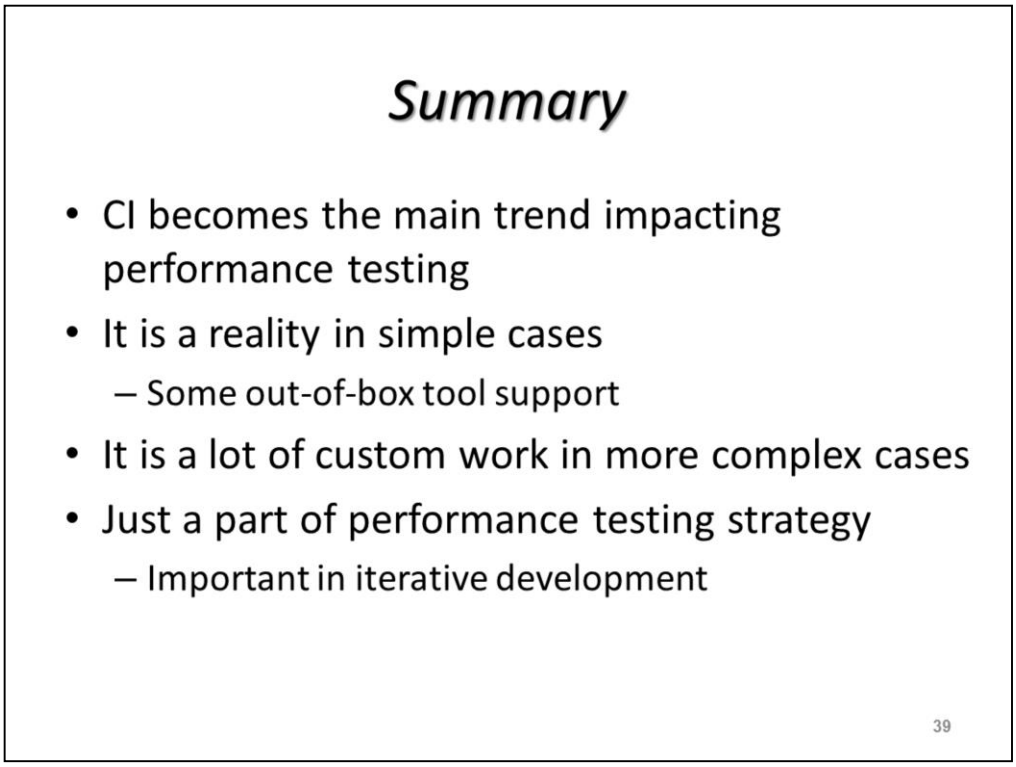
Yes, the other ways to mitigate performance risks mentioned above definitely decrease performance risks comparing to situations where nothing is done about performance at all. And, perhaps, may be more efficient comparing with the old stereotypical way of doing load testing – running few tests just before 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 a combination of different approaches is needed 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.



While cloud looks quite different from mainframes, there are many similarities between them, 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.



Continuous Integration becomes the main trend impacting performance testing. It is a reality in simple cases: there is basic out-of-box tool support (in many tools) allowing quickly put together simple continuous performance test. However what you can do is rather limited (and differs depending on specific tool ecosystem). And as soon as you need more, you basically need to implement it yourself. Or wait until tools extend their support into the area you need – that may be awhile, considering that it would be quite difficult to implement it well in a generic way.

In more complex cases, it probably would be a lot of custom work. So its expedience would rather depend on existing risks and available resources. And it is important to understand that it is just a part of performance testing strategy. Rather important in iterative development – but still a lot should be done outside CI even with running tests, not to mention other performance engineering activities starting from any non-trivial performance analysis.

References

[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/

[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

[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/

[PODE16] Podelko, A. Reinventing Performance Testing. CMG, 2016.

[PRAT15] Prather, D. The Power of Continuous Performance Testing. https://www.stickyminds.com/article/power-continuous-performance-testing

[TOWN17] Townshend, S. The Myth of Continuous Performance Testing. https://www.linkedin.com/pulse/myth-continuous-performance-testing-stephen-townshend