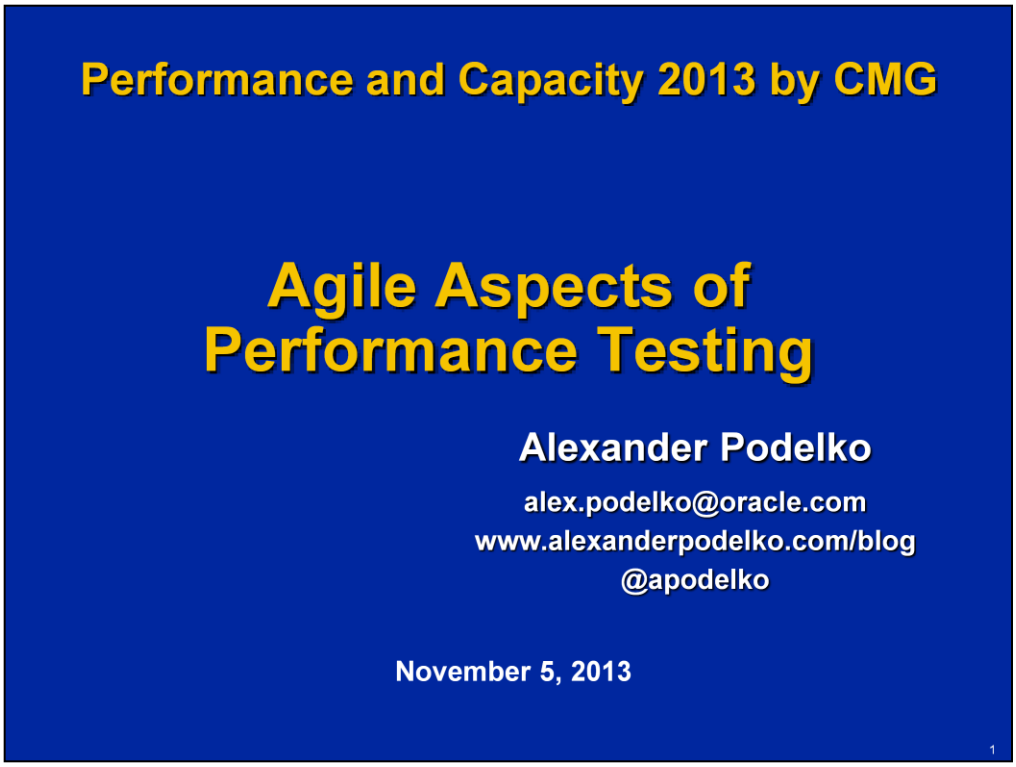
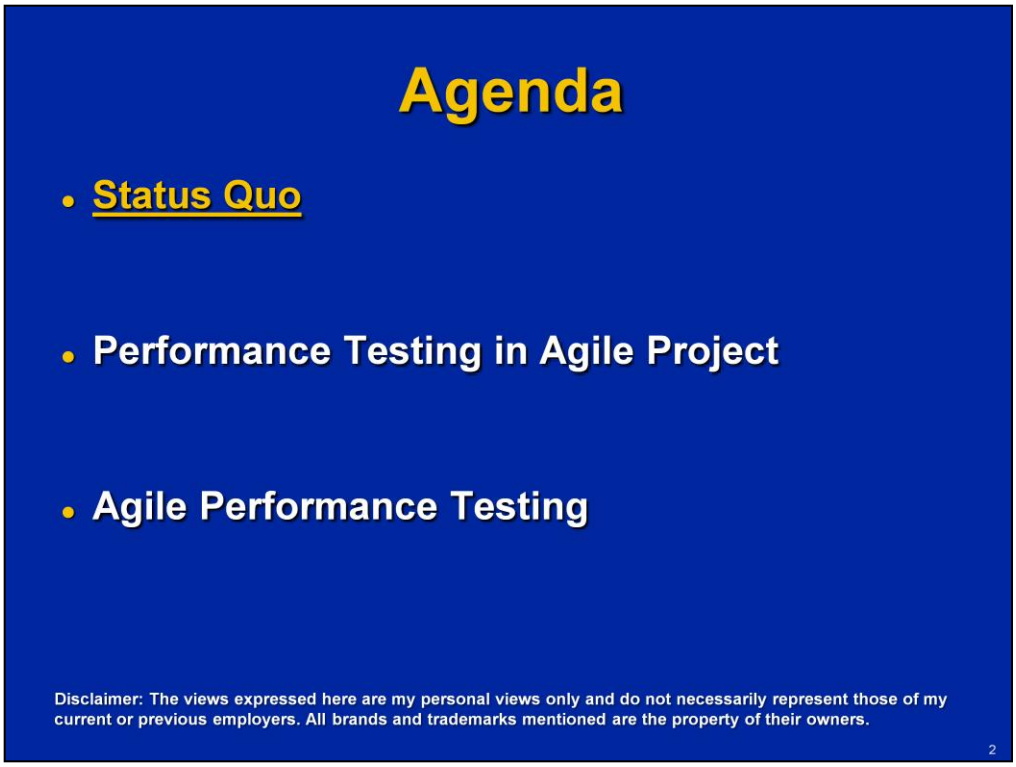


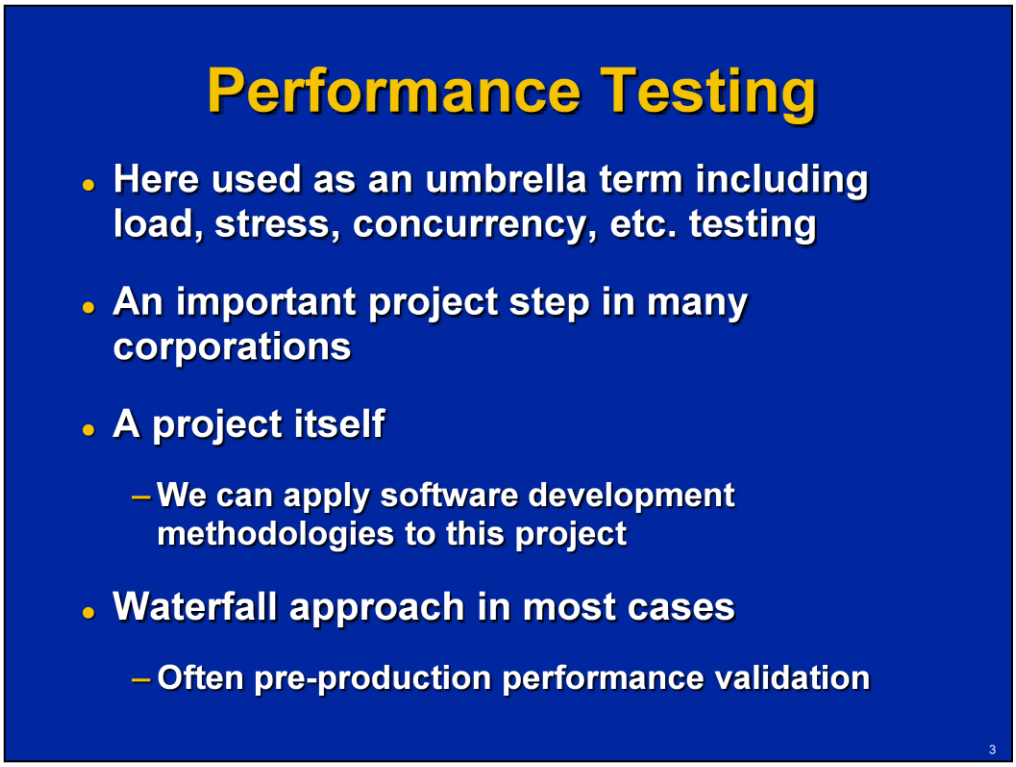
It looks like agile methodologies are somewhat struggling with performance testing. Theoretically it should be a piece of cake: every iteration you have a working system and know exactly where you stand with the system's performance. Unfortunately, it doesn't always work this way in practice. Performance related activities tend to slip toward the end of the project. Another issue is that agile methods are oriented toward breaking projects into small tasks, which is quite difficult to do with performance: performance-related activities usually span the whole project.

From another side, performance testing is rather agile in itself. Approaching performance testing formally, with rigid, step-by-step approach and narrow specialization often leads to missing performance problems or to prolonged agony of performance troubleshooting. With small extra efforts, making the process more agile, efficiency of performance testing increases significantly and these extra efforts usually pay off multi-fold even before the end of performance testing. This paper discusses agile aspects of performance testing in detail, including both performance testing in agile projects and doing performance testing in agile way.



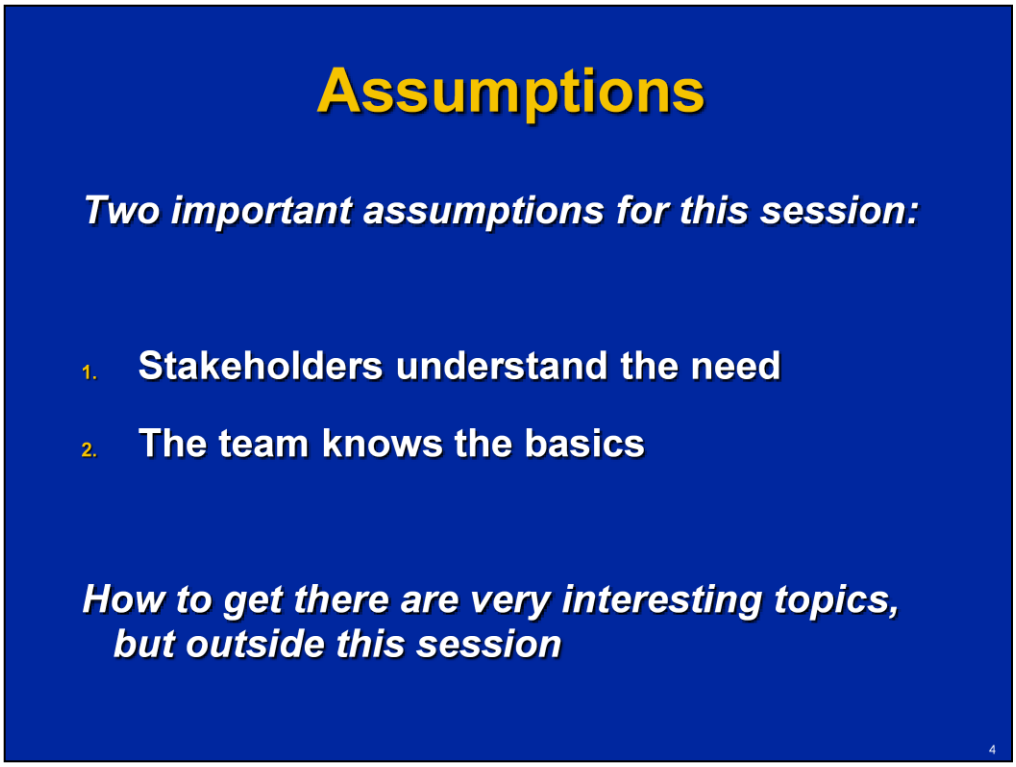
After a short introduction into the subject, two major aspects would be discussed: performance testing in agile projects and doing performance testing in agile way.

Disclaimer: The views expressed here are my personal views only and do not necessarily represent those of my current or previous employers. All brands and trademarks mentioned are the property of their owners.

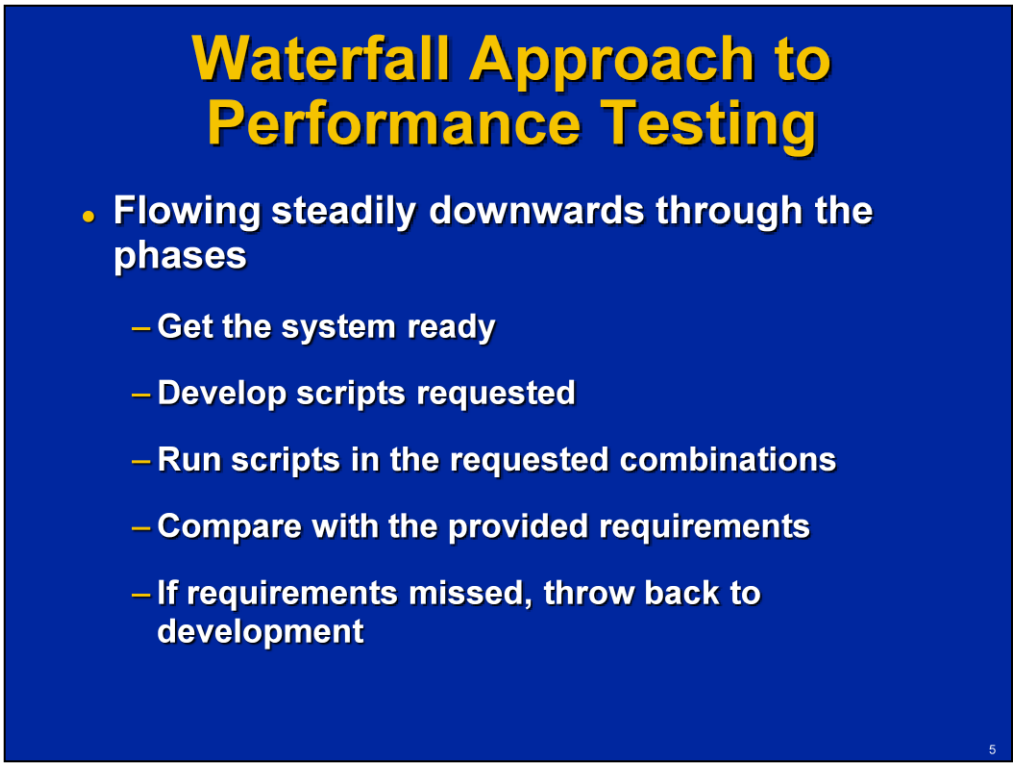


Most large corporations have performance testing / engineering groups today, performance testing becomes a must step to get the system into production. Still, in most cases, it is pre-production performance validation only.

Even if it is only a short pre-production performance validation, performance testing is a project itself with multiple phases of requirement gathering, test design, test implementation, test execution, and result analysis. So in most cases software development methodologies could, with some adjustments, be applicable to performance testing.

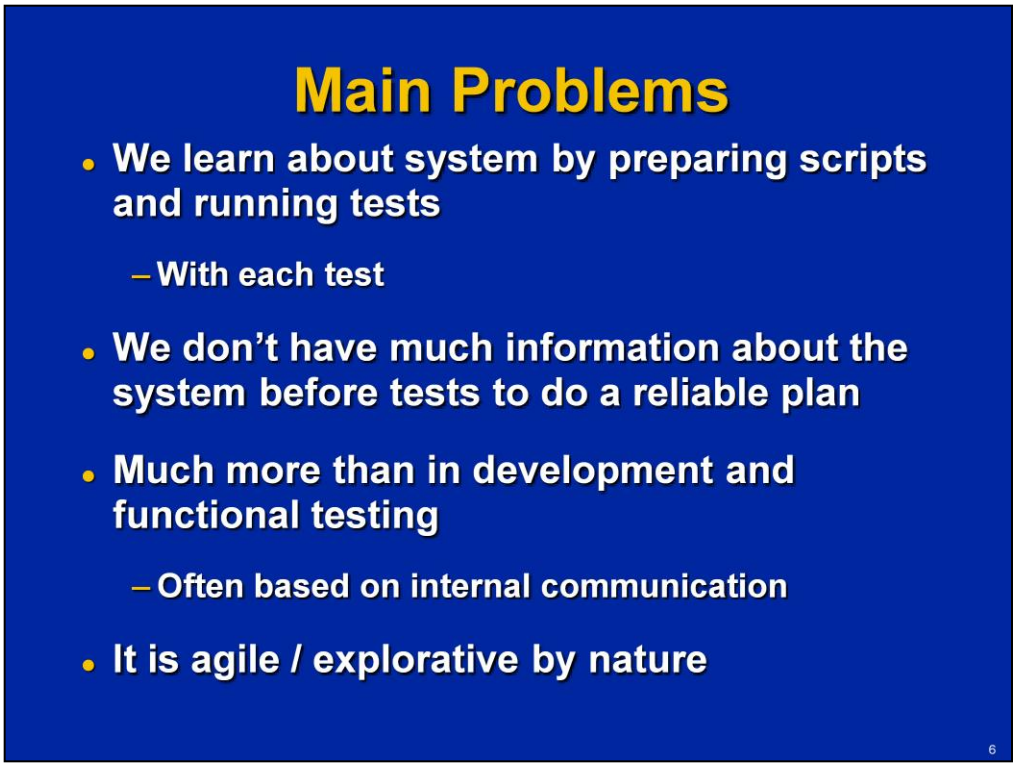


There are two very important assumptions for this discussion: stakeholders understand the need in performance engineering and the team knows the basics. How to get there is a very interesting topic, but it is outside of this session. For those, who are not there yet, the issues described here may sound abstract and they probably have more urgent problems to fight – but it is something you may want to be aware about while you are getting there.



The waterfall approach in software development is a sequential process in which development is seen as flowing steadily downwards (like a waterfall) through the phases of requirements analysis, design, implementation, testing, integration, and maintenance [Waterfall]. Being a step on the project plan, performance testing is usually scheduled in the waterfall way, when you need to finish one step to start next. Typical steps could be, for example:

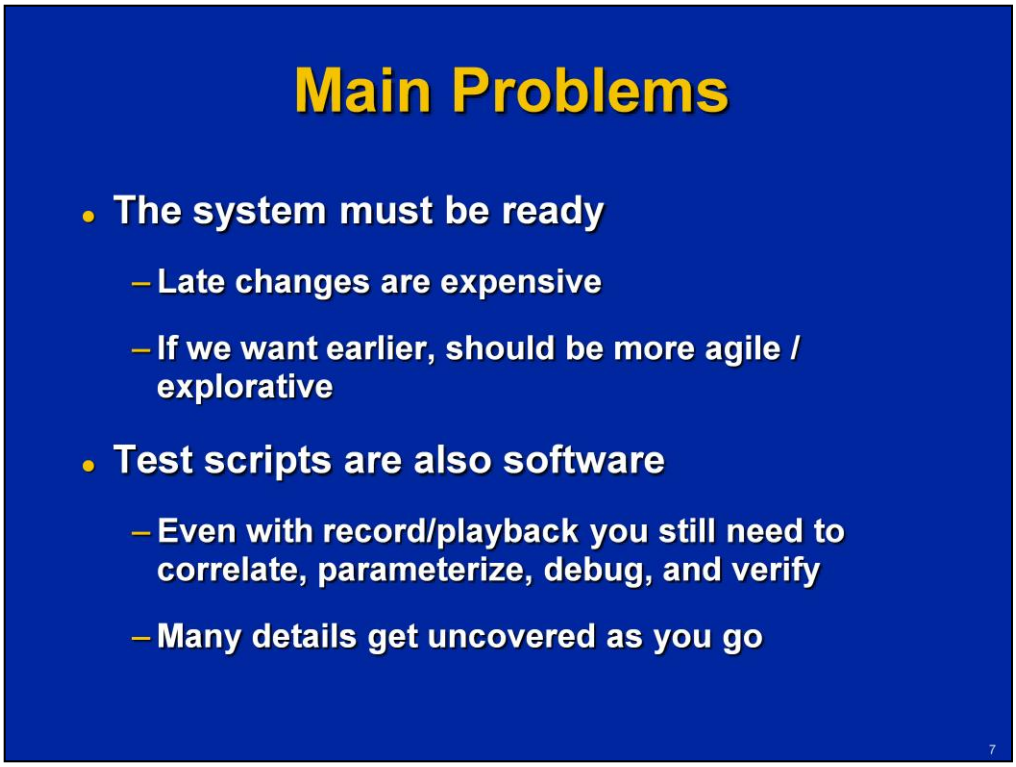
-Get the system ready
-Develop scripts requested (sometimes offshore)
-Run scripts in the requested combinations
-Compare with the requirements provided
-Allow some percentage of errors according to the requirements
-Involve the development team if requirements are missed



The main problem with the waterfall approach is that we learn about system by preparing scripts and running tests. With each test we may get additional information. We don't have much information about the system before tests to do a reliable plan (unless, of course, it is a minor change in a well-known system). This issue is much more challenging than in development and functional testing (where it exists up to a smaller degree – and where agile development and explorative testing are answers to the challenge). Often you need to do some investigation to understand the system before you may create a meaningful performance testing plan.

Performance engineers sometimes have system insights that nobody else has, for example:
•Internal communication between client and server if recording used
•Timing of every transaction (which may be detailed up to specific request and its parameters if needed)
•Resource consumption used by specific transaction or a set of transactions.
This information is actually additional input to test design and is not readily available before performance testing starts. Often the original test design is based on incorrect assumptions and need to be corrected based on the first results.

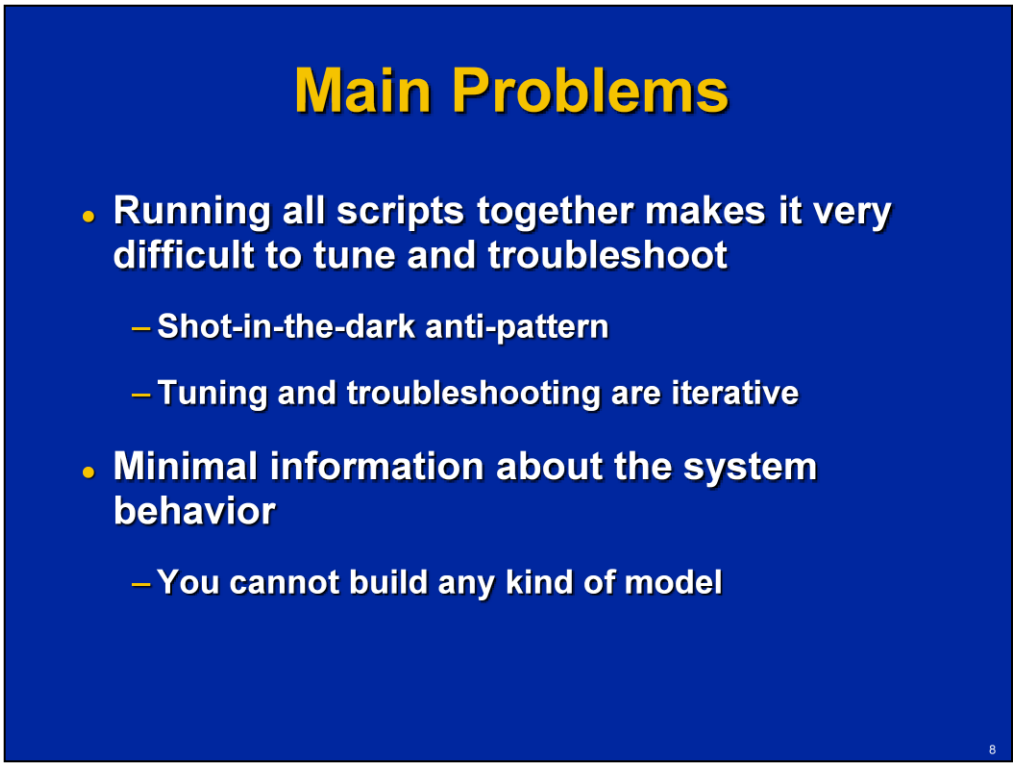
So performance testing of new systems is agile / explorative by nature: you get information about the system as you are running tests and this new information allows you to elaborate further performance testing plan.



At first glance, the waterfall approach to performance testing appears to be a well established, mature process. But there are many serious pitfalls on this way. Here are some of the most significant:

It assumes that the system is completely ready, at minimum, all functionality components included in the performance test. The direct result of waiting until the system is "ready" is that it must occur very late in the development cycle. By that point fixing any problem would be very expensive. It is not feasible to perform such full-scope performance testing early in the development lifecycle. If we want to do something earlier, it should be a more agile/explorative process.

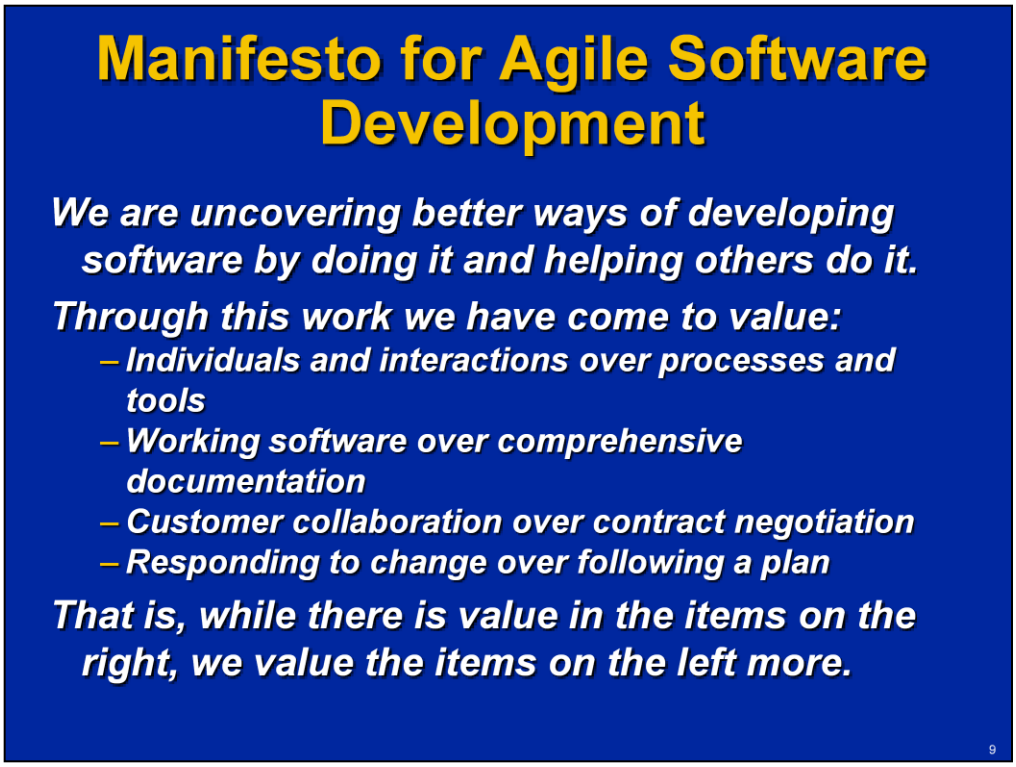
Performance test scripts which are used to create the system load are also software. Record/playback load testing tools may give the tester the false impression that creating scripts is quick and easy. In fact, correlation, parameterization, debugging, and verification may be pretty challenging tasks. Running a script for a single user that doesn't yield any errors doesn't prove much. I have seen large-scale performance testing efforts at a large corporation where none of the script executions actually get through logon (single sign-on token wasn't correlated, so each request returned the login page) – but performance testing was declared successful and the results were reported to management.



Running all scripts together make it very difficult to tune and troubleshoot. It often becomes an illustration to the Shot-in-the-dark anti-pattern [Pepperdine06], "the best efforts of a team attempting to correct a poorly performing application without the benefit of truly understanding *why* things are as they are". Or you need to go back and disintegrate tests to find the exact part causing problems. Moreover, tuning and performance troubleshooting are iterative processes, which difficult to place inside the "waterfall" approach. And, in most cases, it is not something you can do off-line – you need to tune system and fix major problems before results will make sense.

Running a single large test (or even a few of them) gives minimal information about the system behavior. You cannot build any kind of model of it (either formal or informal), you will not see relation between workload and system behavior, you will not able to answer what-if sizing and configuration questions. In most cases the workload used in performance tests is only an educated guess, so you need to understand how stable the system would be and how consistent results would be if real workload were somewhat different.

Doing performance testing in a more agile, iterative way may increase its efficiency significantly.



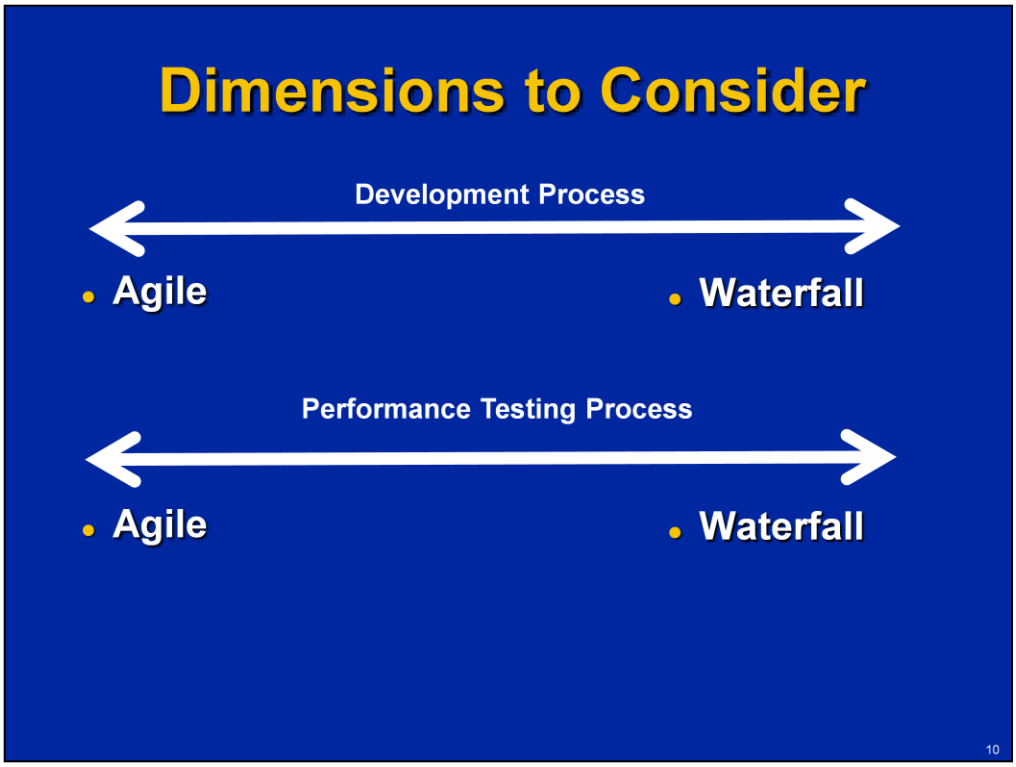
The word agile here doesn't refer to any specific development process or methodology. It rather refers to the original definition of this word as stated in "Manifesto for Agile Software Development" [Manifesto01]:

We are uncovering better ways of developing software by doing it and helping others do it.

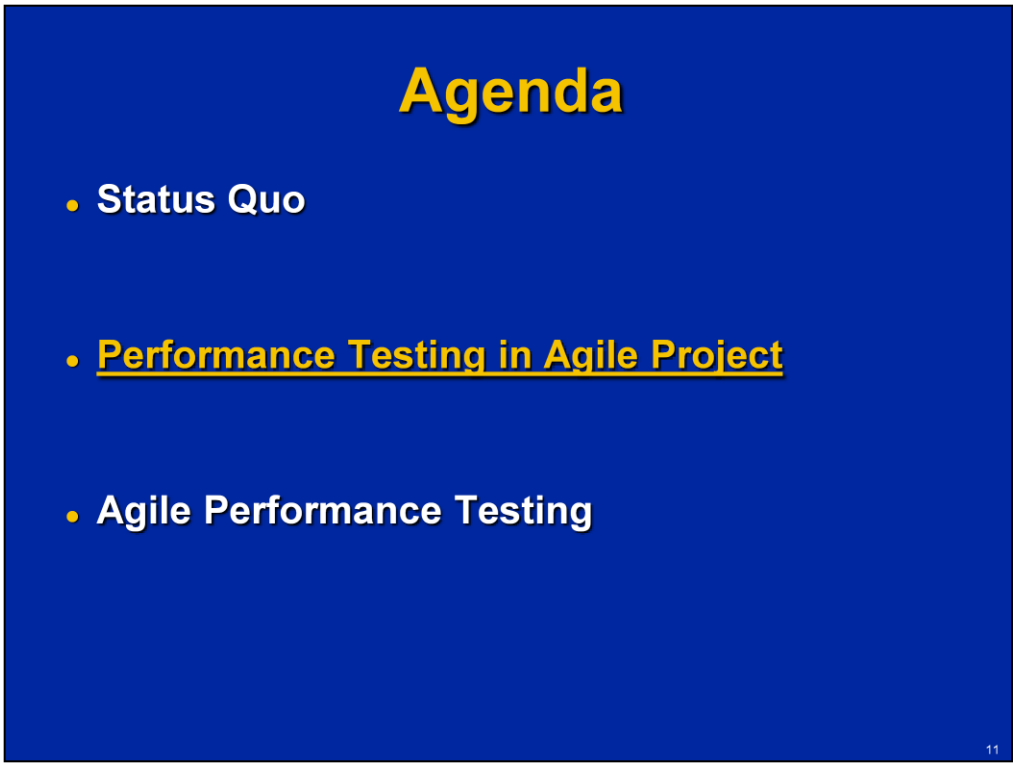
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

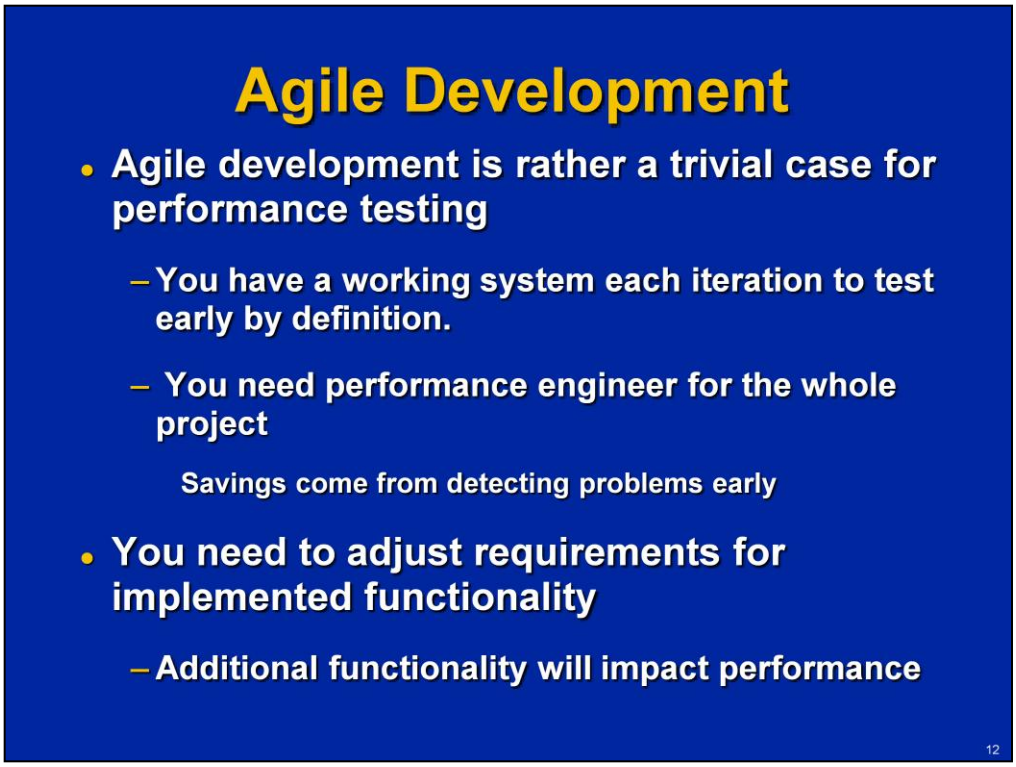
That is, while there is value in the items on the right, we value the items on the left more.



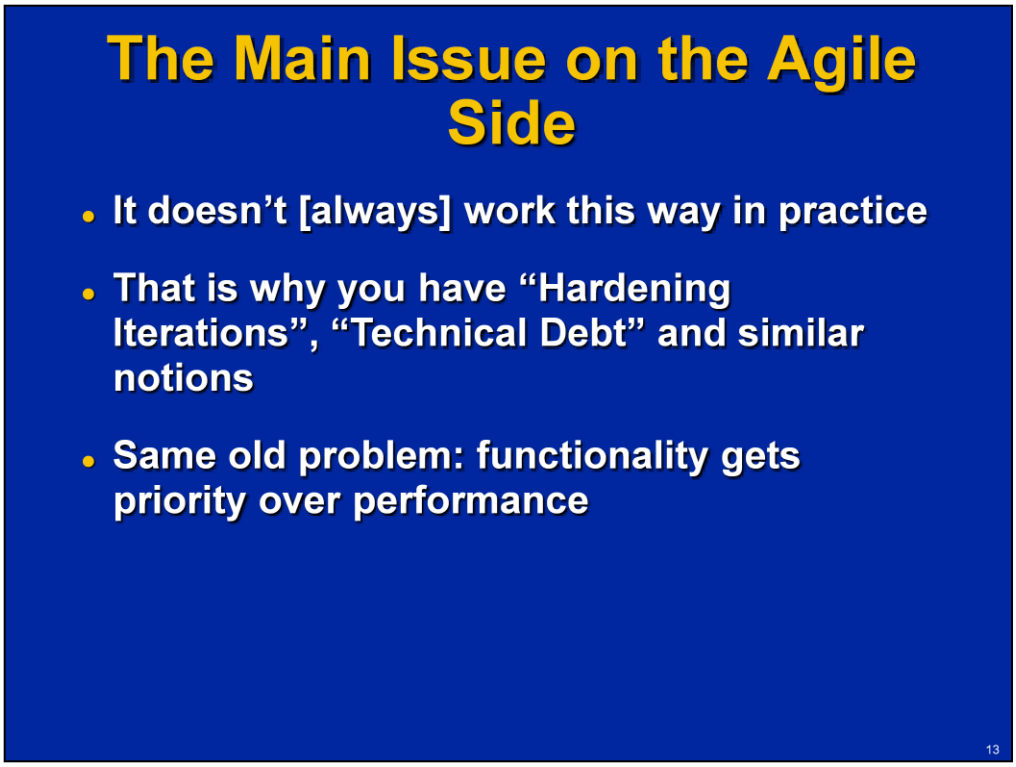
We have two different dimensions here. The first is the process used for the whole project. The second is the approach used to performance testing as a project (independently from the process used for the whole project).



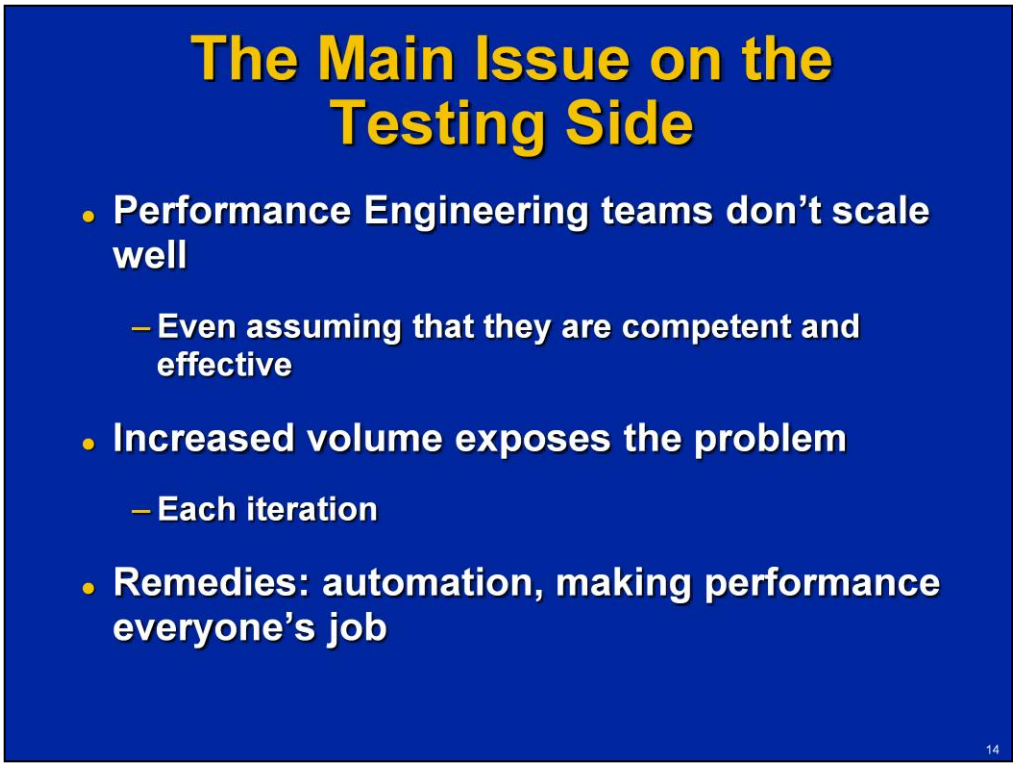
First we discuss performance testing in software development projects using agile methodologies.



Practical agile development is struggling with performance in general. Theoretically it should be a piece of cake: 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 much more straightforward and solve the first problem of traditional approach that the system should be ready for performance testing (so it usually happened very late in the process).



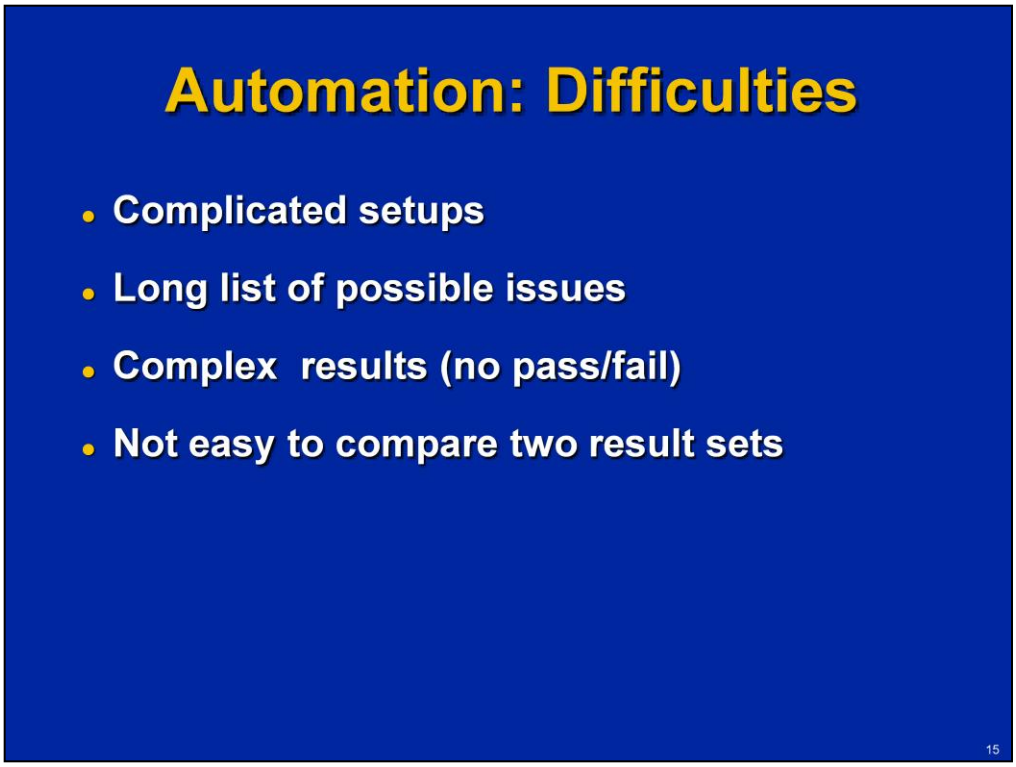
Unfortunately, it looks like it doesn't always work this way in practice. So such notions as "hardening iterations" and "technical debt" get introduced. Although it is probably the same old problem: functionality gets priority over performance (which is somewhat explainable: you first need some functionality 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.



The fundamental issue is, as I see it, 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.

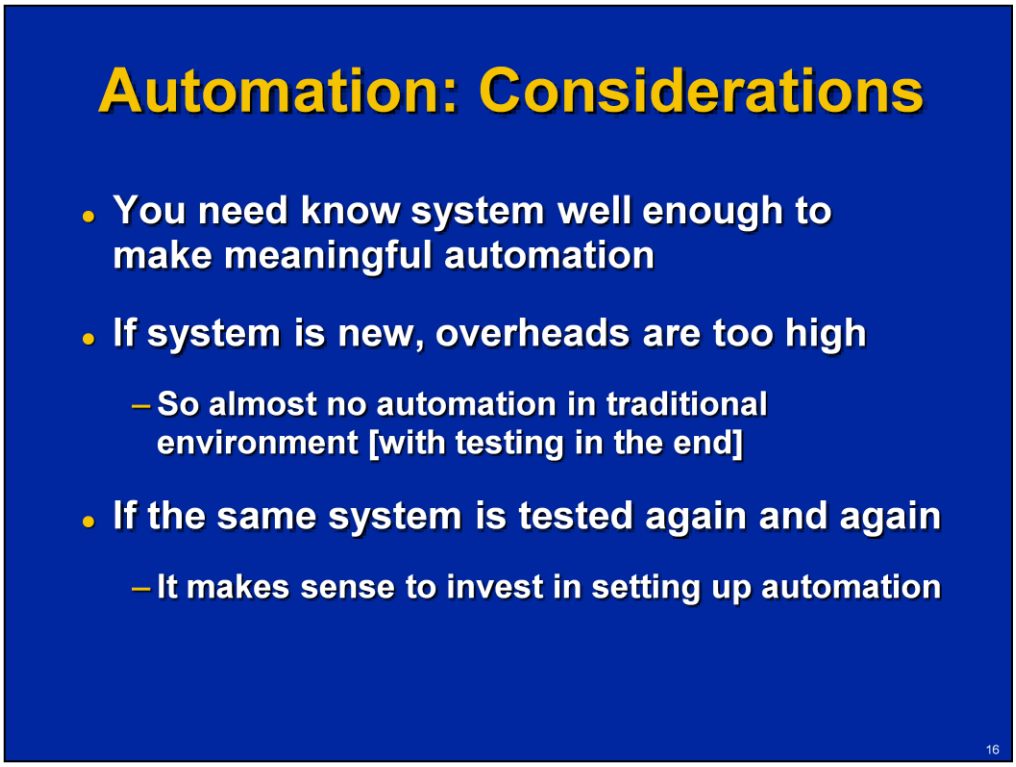
Just to avoid misunderstandings, I am a strong supporter of having performance teams and I believe that it is the best approach to building performance culture. Performance is a special area and performance specialists should have an opportunity to work together to grow professionally. The details of organizational structure may vary, but a center of performance expertise should exist. Only thing I am saying 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).

Actually remedies are well known: automation, making performance everyone jobs (full immersion), etc. [Crispin09, Barber11] However they are not wide-spread yet.



Automation (continuous performance testing) 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 would probably require much more human intervention, but it isn't impossible.

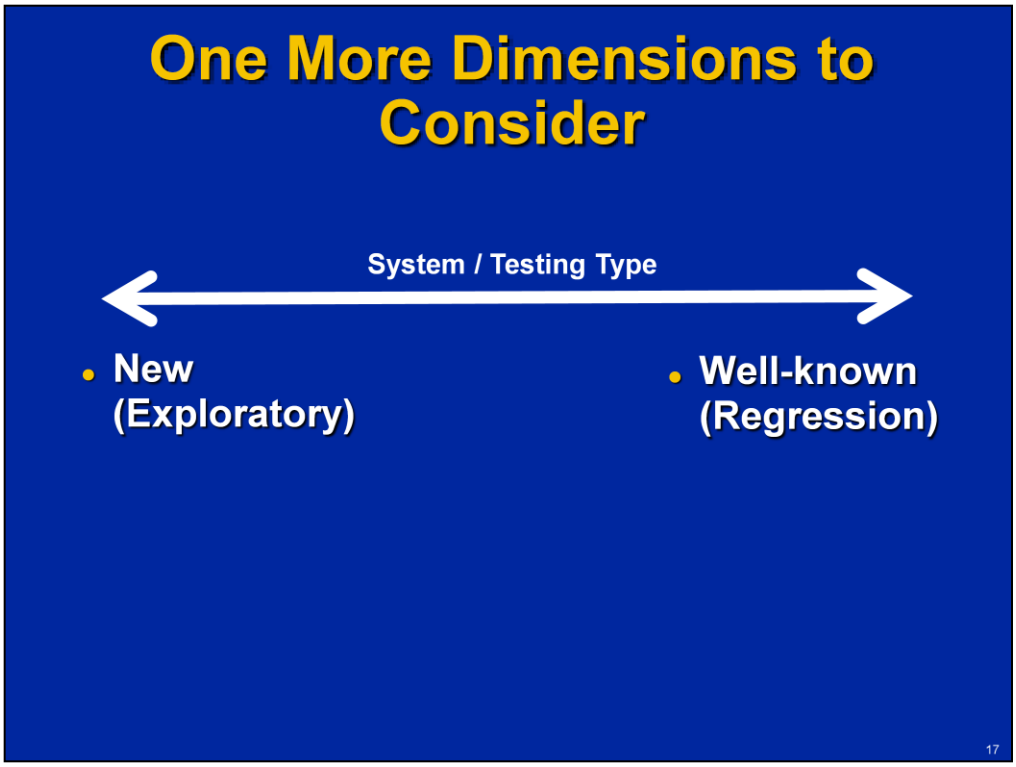


However, the cost of performance testing automation is high. You need to know system well enough to make meaningful automation. Automation for a new system doesn't make much sense - overheads are too high. So there was almost no automation in traditional environment [with testing in the end with a record/playback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

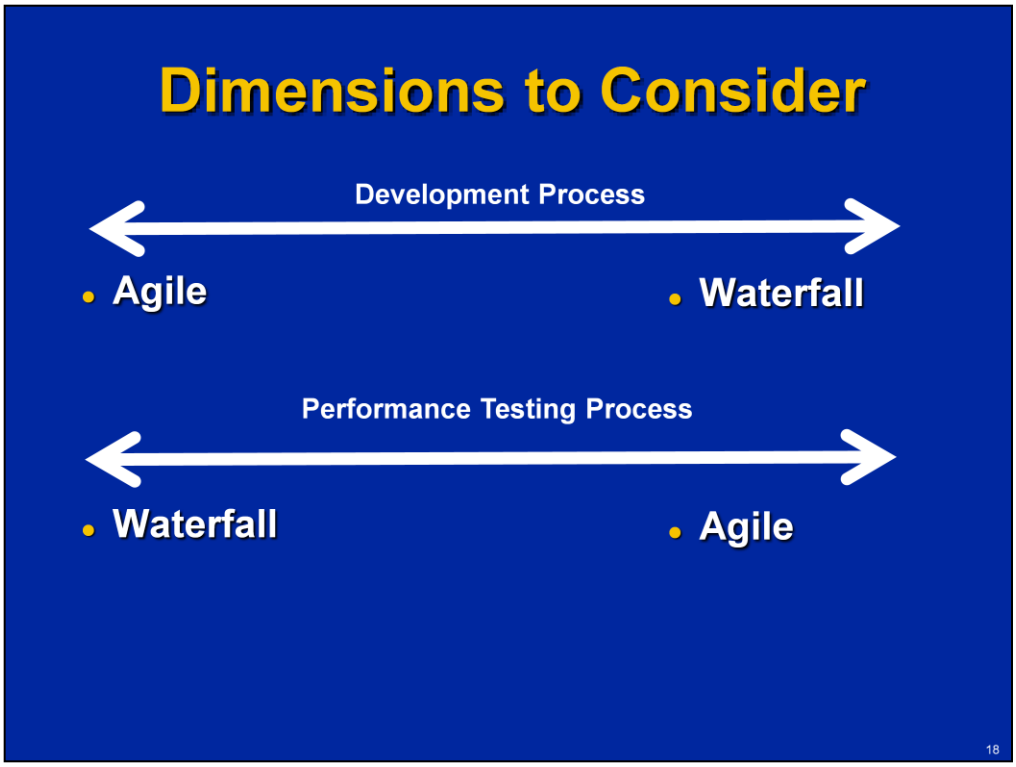
It is opposite when the same system is tested again and again (as it should be in agile projects). It makes sense to invest in setting up automation. It rarely happened in traditional environments – even if you test each release, they are far apart and the difference between the releases prevents re-using 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.

I am not saying that automation would replace performance testing as we know it. 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.

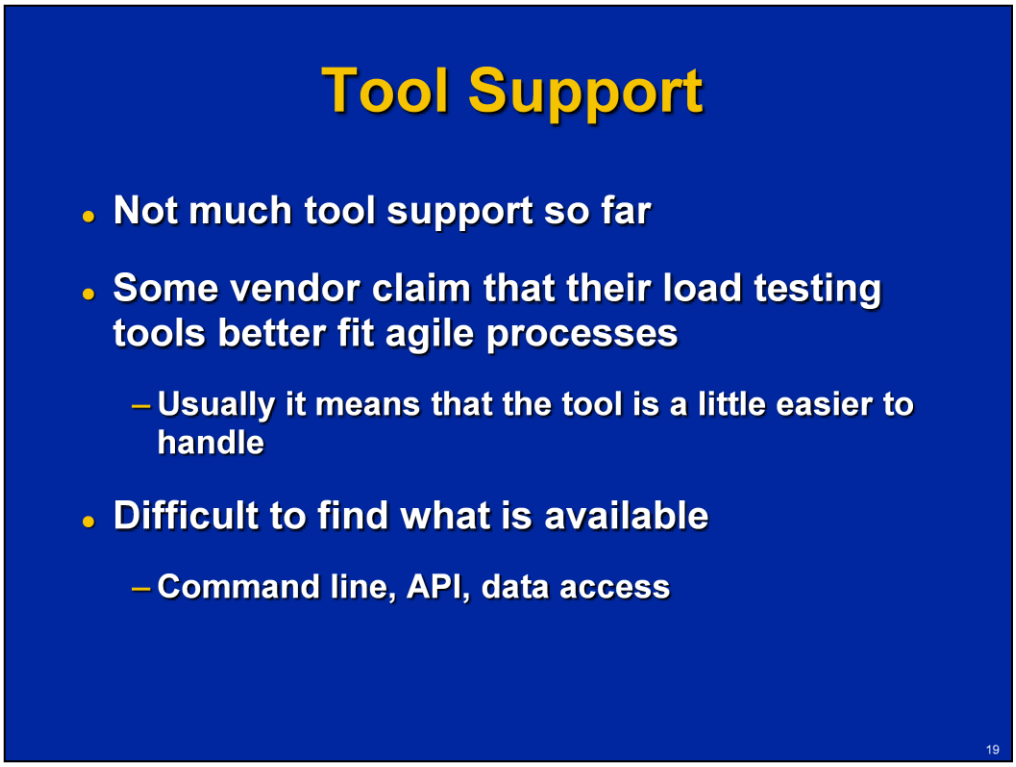
Actually it is similar to functional testing where you need to find a trade-off between automated regression testing and exploratory testing – with the difference that you use tools in performance testing in any case and setting up continuous performance testing is much more new and challenging.



So we have one more dimension to consider: what kind of the system we have and what kind of performance testing we need. For a new system, when we don't know much about it, we need an agile / exploratory kind of testing (one definition of exploratory testing is: *Exploratory testing is simultaneous learning, test design, and test execution* [Bach03] - that is exactly what we need for a new system). For a well-known system, when changes are rather small, we don't need much exploration and automation may be very effective.



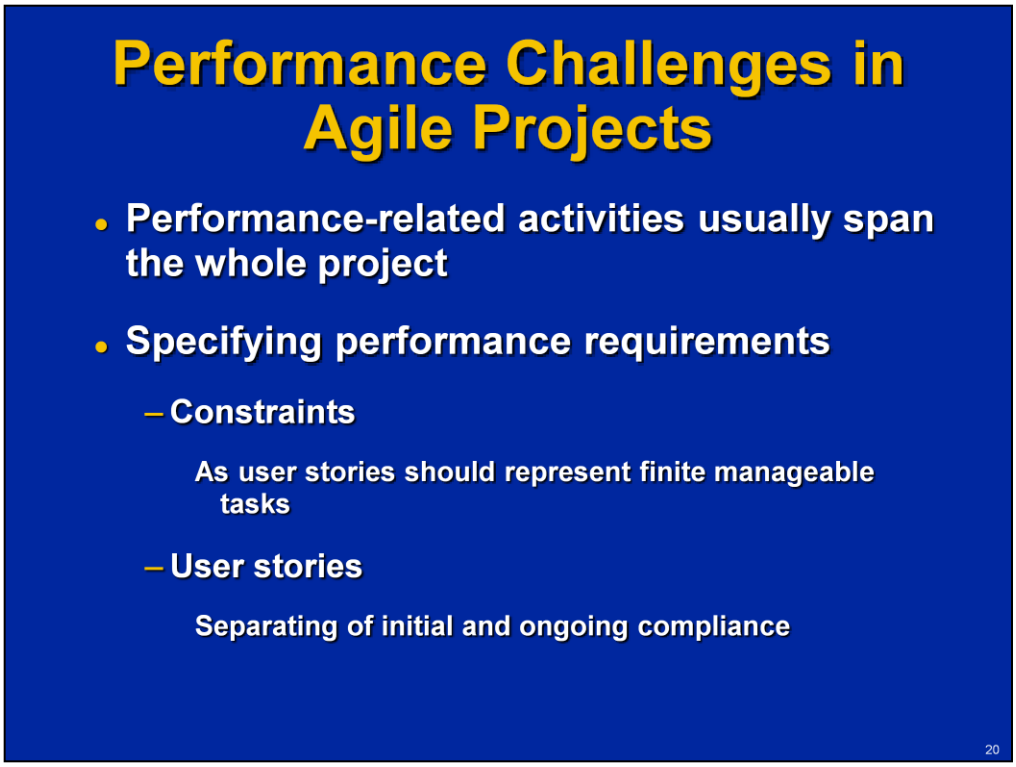
Returning to the diagram of dimensions to consider, actually we need to re-orient it: the more agile / short iteration is the development process, the less agile would be performance testing (we just need to explore a small set of functionality delivered in an iteration). The more waterfall / large iteration is the development process, the more agile / explorative should be performance testing to deliver meaningful and prompt results.



There are some vendors claiming that their load testing tools better fit agile processes, but it looks like in the best case it means that the tool is a little easier to handle (and, unfortunately, often just because there is not much functionality in it). While ease of use is an important factor, it is not specific to agile and should be accompanied by all necessary functionality.

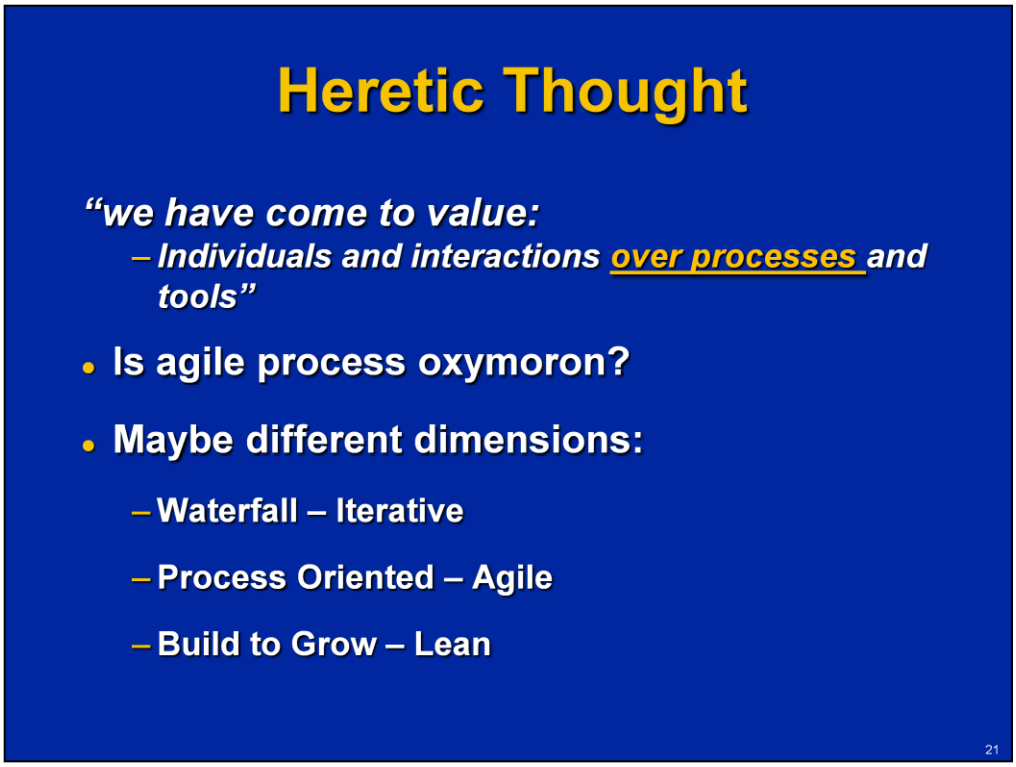
What makes agile projects really different is need of running large number of tests repeatedly – resulting in need for tools to support performance testing automation. Unfortunately, even if a tool has something that may be used for automation, like starting by a command line with parameters, it is difficult to find out.

So far I read about few full-scale implementation of continuous performance testing (for example, [Harding11] or [OpTier12]). Probably we don't see it more often because we don't have much infrastructure for that kind of automation and performance testers may be not the best people to create complex integrated solutions from dozens of not-related pieces (as those who implemented it did). When we get more automation support in tools, we will probably see it more often.



Another issue here is that 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 [Hazrati11].

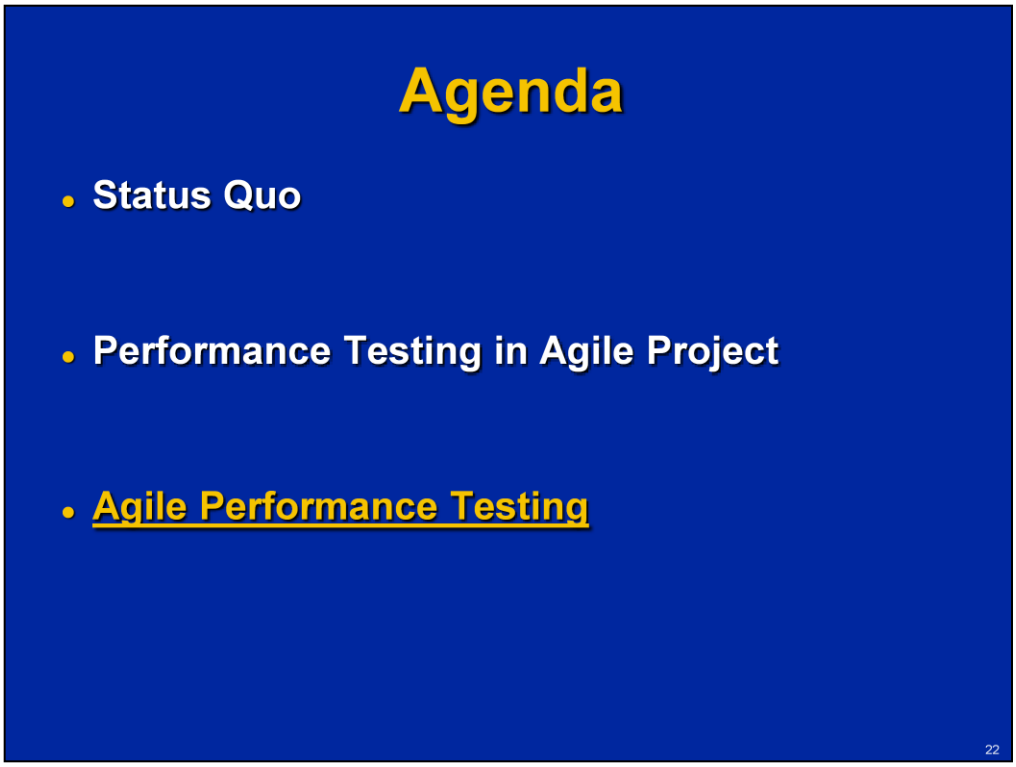


The Agile manifesto [Manifesto01] states "we have come to value … Individuals and interactions over processes and tools". And now we have processes explained in excruciating details (up to how to stand and how to chart), which are referred to as Agile. Do we have a contradiction here? I am not discussing their value here at all – I just saying that if read The Agile manifesto literally, it looks like agile process is an oxymoron. Sure, it still says there is value in the items on the right, but nevertheless.

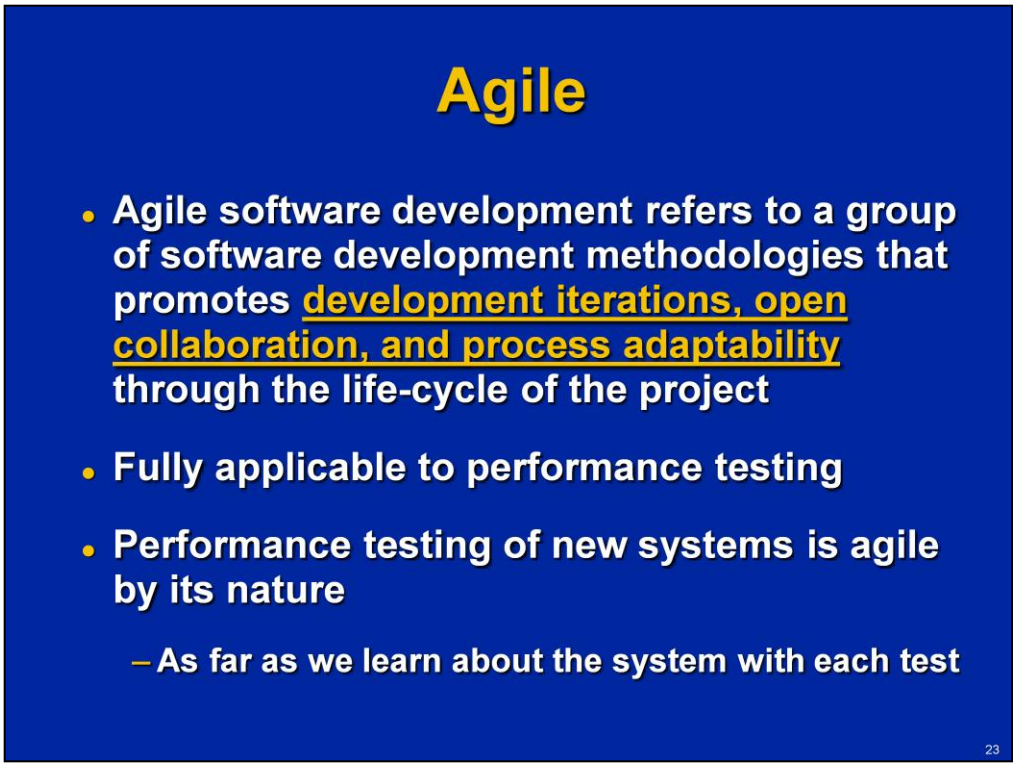
Perhaps we have multiple dimensions here. One dimension would be Waterfall – Iterative (or maybe Large Batch – Small Batch following [Ries11] would be even more exact – waterfall perhaps may be considered iterative too, but with very large iterations). Another dimension may be Process Oriented – Agile.

It is interesting what would be opposite to lean? Probably not fat (although it is what using the word "lean" hints) – you probably don't want to have fat in whatever approach you use. Maybe "Build to Grow"? Lean means that you do absolute minimum (accumulating technical debt) – so the opposite would be if you build infrastructure / framework to grow from the beginning. So Build to Grow – Lean would be probably another dimension. Minimal Viable Product may be not the best option for something well defined with high performance and scalability requirements.

Then every software development methodology may be described by its position on these (and/or other) dimensions and a set of specific practices (which may be useful in some context).

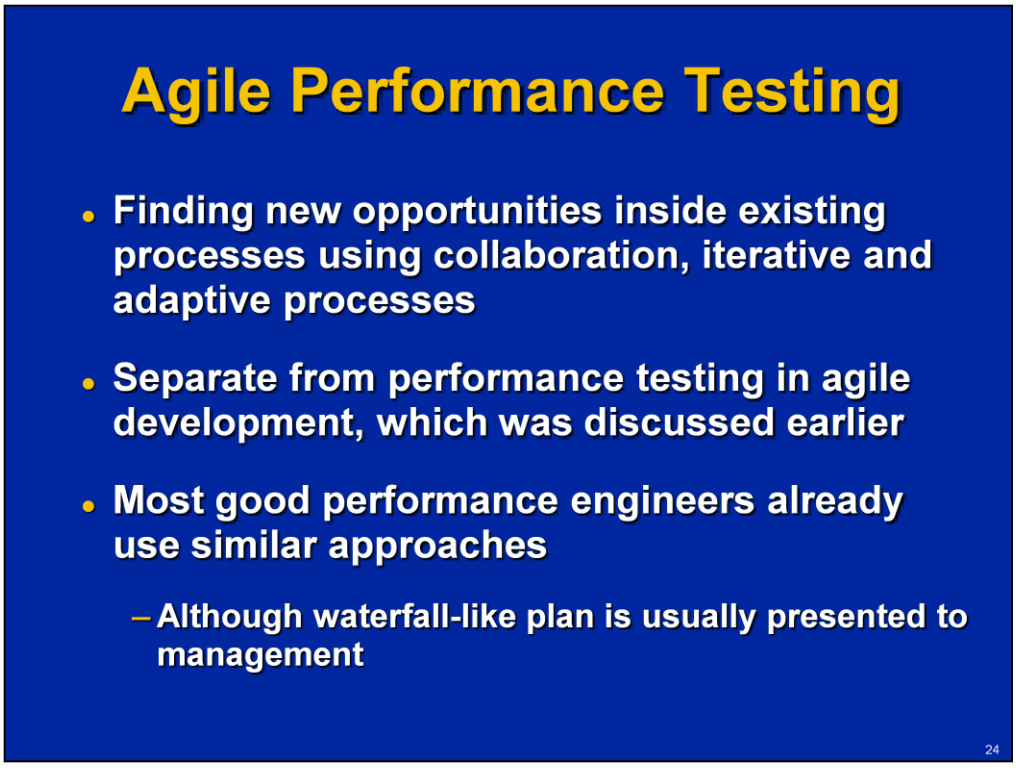


Now we will discuss another agile aspect of performance testing : approaching performance testing projects in agile way (independently from what approach is used in the overall software development project).



Agile software development refers to a group of software development methodologies that promotes development iterations, open collaboration, and process adaptability throughout the life-cycle of the project [Agile]. The same approaches are fully applicable to performance testing projects. Performance testing is somewhat agile by its nature; it often resembles scientific research rather than the routine execution of one plan item after another. Probably the only case where you really can do it in a formal way is when you test a well-known system (some kind of performance regression testing). You can't plan every detail from the beginning to the end of the project – you never know at what load level you face problems and what you would be able to do with them. You should have a plan, but it needs to be very adaptable. It becomes an iterative process involving tuning and troubleshooting in close cooperation with developers, system administrators, database administrators, and other experts [Podelko06].

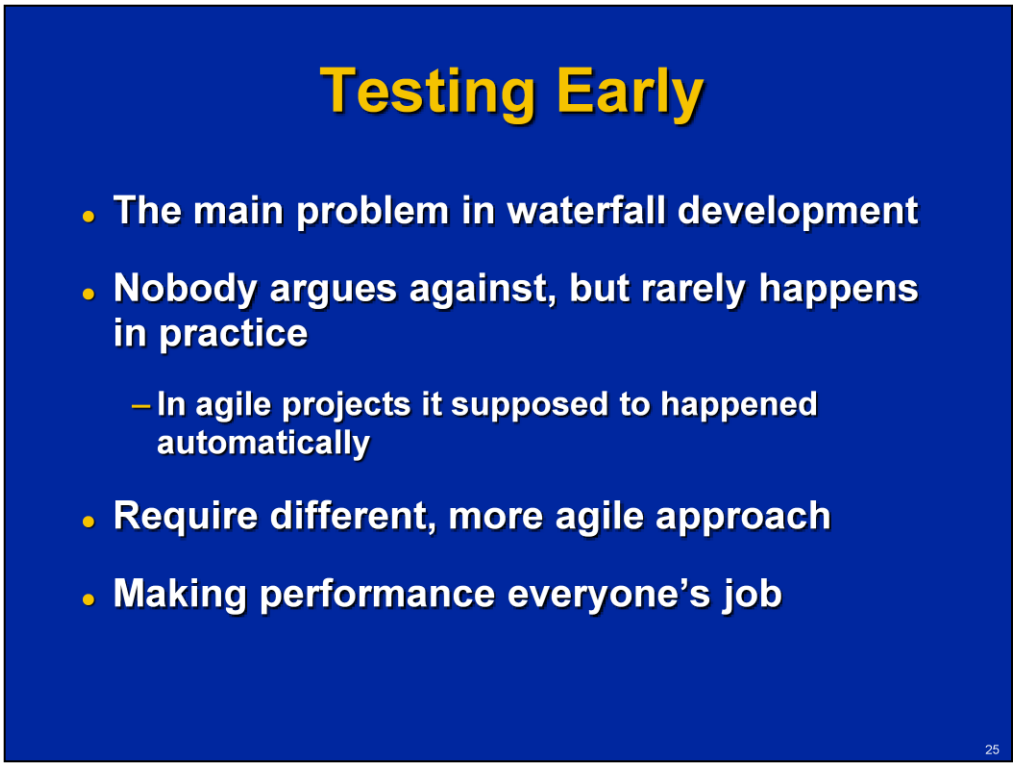
Performance testing is iterative: you run a test and get a lot of information about the system. To be efficient you need to analyze the feedback you get from the system, make modifications to the system and adjust you plans if necessary. For example, you plan to run 20 different tests and after executing the first test you find that there is a bottleneck (for example, the number of web server threads). Therefore, there is no point in running the other 19 tests if they all use the web server, it would be just a waste of time until you find and eliminate the bottleneck. In order to identify the bottleneck the test scenario may need to be changed.



Even if the project scope is limited to pre-production performance testing, approaching testing with an agile, iterative approach you meet your goals faster and more efficiently and, of course, learn more about the system along the way. After we prepare a script for testing (or however the workload is generated), we can run one, a few, and many users (how many depends on the system), analyze results (including resources utilization), and try to sort out any errors. The source of errors can be quite different – script error, functional error, or a direct consequence of a performance bottleneck. It doesn't make much sense to add load until you figure out what is going on. Even with a single script you can find many problems and, at least partially, tune the system. Running scripts separately also allows you to see how much resources are used by each type of load and make some kind of system's "model".

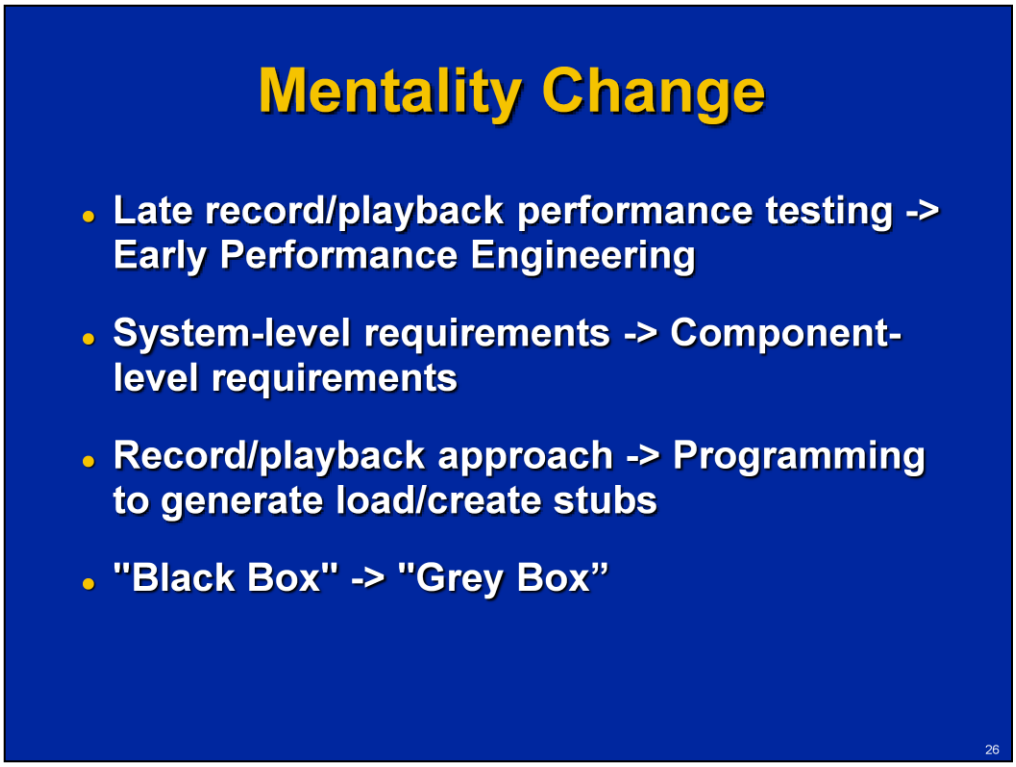
Using the "waterfall" approach doesn't change the nature of performance testing; it just means that you probably do a lot of extra work and still come back to the same point, performance tuning and troubleshooting, much later in the cycle. Not to mention that large tests using multiple use cases are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

Using an agile, iterative approach doesn't mean that you need to re-define the software development process, but rather find new opportunities inside existing processes. I believe that most good performance engineers are already doing performance testing in an agile way but just presenting it as "waterfall" to management (some kind of guerilla tactic). In most cases you need to present a waterfall-like plan to management, and then you are free to do whatever is necessary to properly test the system inside the scheduled timeframe and scope. If opportunities exist, performance engineering may be extended further, for example, to early performance checkpoints or even full Software Performance Engineering [Smith02]. But don't wait until everything is properly in place, make the best possible effort and then look for opportunities to extend it further [Gunther07].

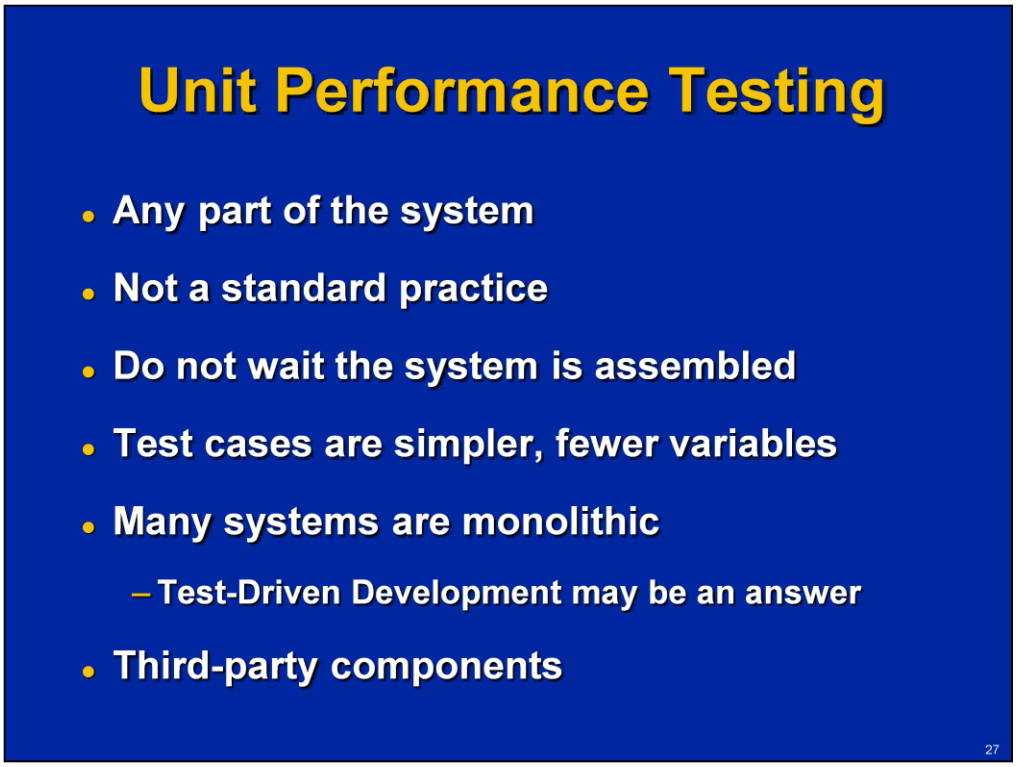


I have never read or heard anybody argue against testing early. Nevertheless, it still rarely happens in practice. Usually some project-specific reasons are stated like tight schedule or budget preventing such activities (if somebody thought about them at all).

The Software Performance Engineering (SPE) approach to the development of software systems to meet performance requirements has long been advocated by Dr. Connie Smith and Dr. Lloyd Williams (see, for example, [Smith02].). While their methodology doesn't focus on testing initiatives, it cannot be successfully implemented without some preliminary testing and data collection to determine both model inputs and parameters as well as to validate model results. Whether you are considering a full-blown SPE or guerrilla–style "back-of-the-envelope" approach, you still need to obtain baseline measurements on which to build your calculations. Early performance testing at any level of detail can be very valuable at this point.



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

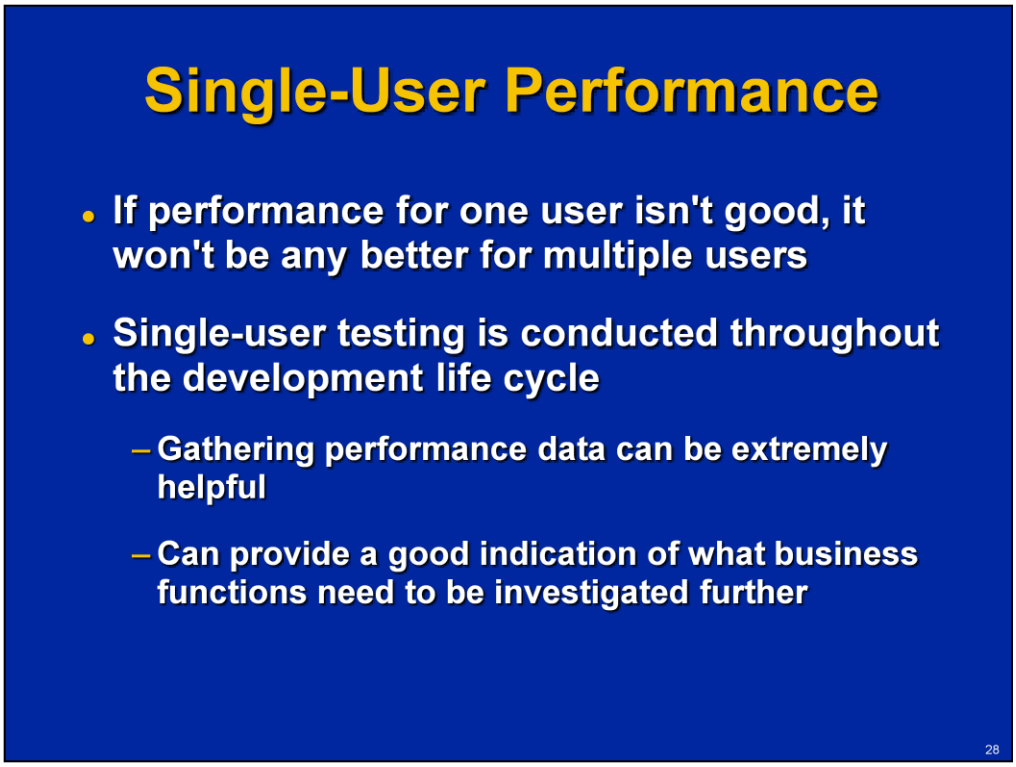


One rarely discussed aspect of early performance testing is unit performance testing. The unit here maybe any part of the system like a component, service, or device. It is not a standard practice, but it should be. As we get later in the development cycle, it is more costly and difficult to make changes. Why should we wait until the whole system is assembled to start performance testing? We don't wait in functional testing, why should we in performance? The pre-deployment performance test is an analogue of system or integration tests, but usually it is conducted without any "unit testing" of performance.

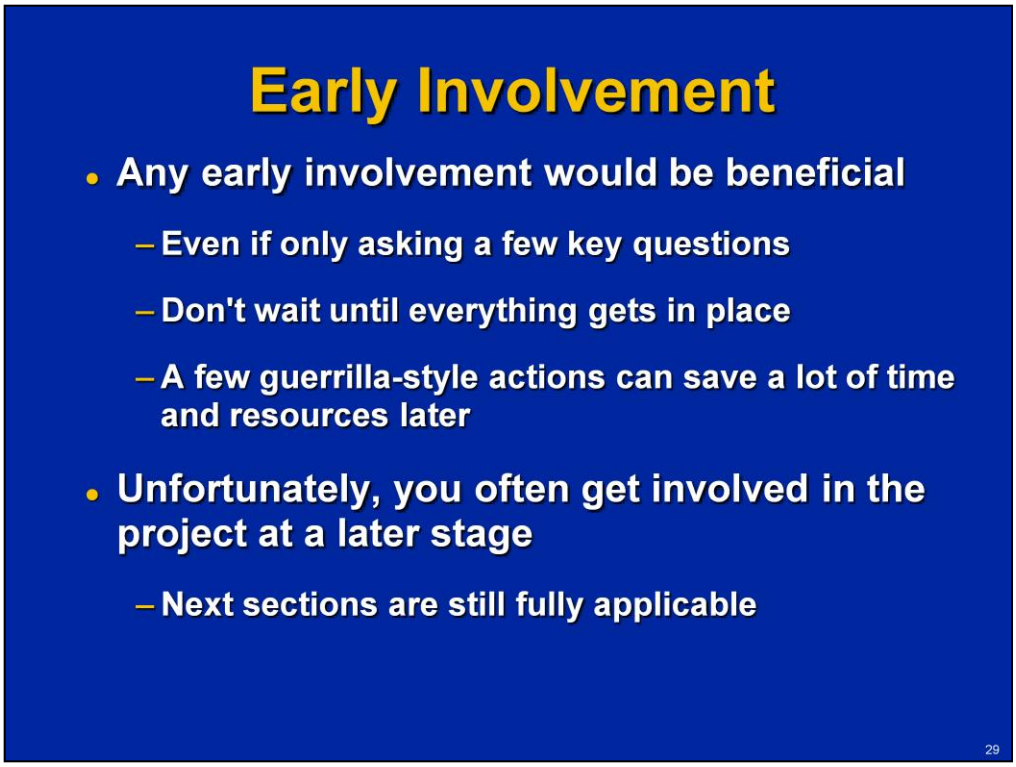
The main obstacle here is that many systems are pretty monolithic; if there are parts - they don't make much sense separately. But there may be significant advantages to test-driven development. If you can decompose the system into components in such way that you may test them separately for performance, then you will only need to fix integration issues when you put the system together. Another problem is that large corporations use a lot of third-party products where the system appears as a "black box" and is not easily understood making it more difficult to test effectively.

During unit testing different variables such as load, the amount of data, security, etc. can be reviewed to determine their impact on performance. In most cases, test cases are simpler and tests are shorter in unit performance testing. There are typically fewer tests with limited scope; e.g., fewer number of variable combinations than we have in a full stress and performance test.

Another kind of early performance testing is infrastructure benchmarking; the hardware and software infrastructure is also a component of the system.

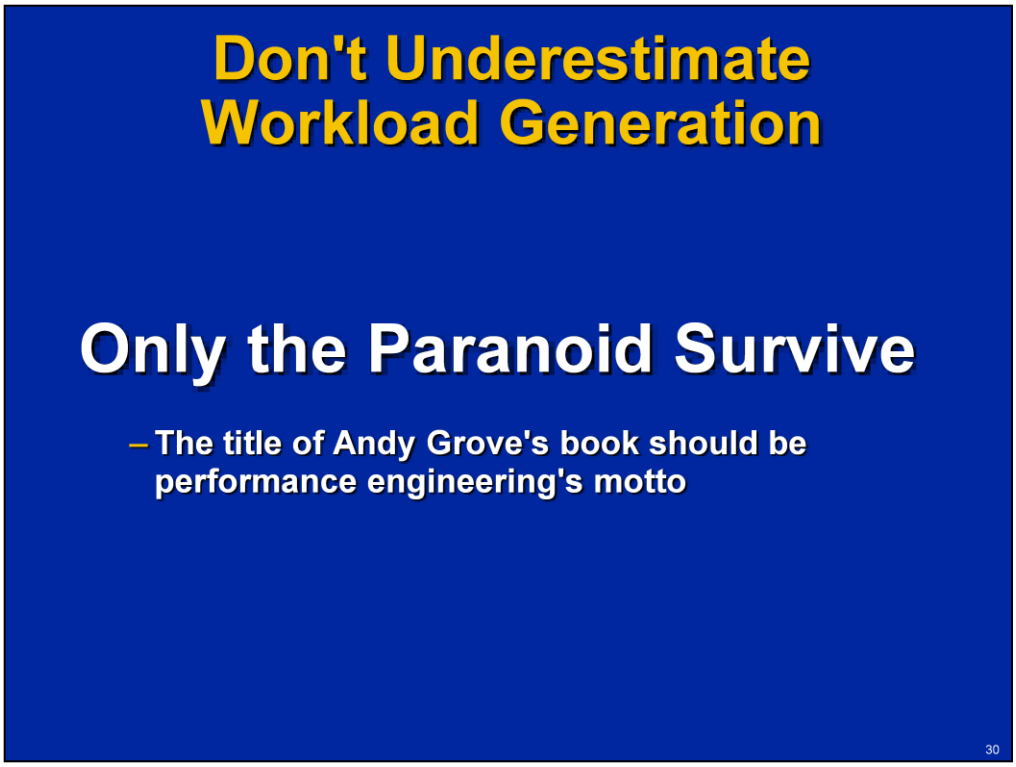


We shouldn't underestimate the power of the single-user performance test. If the performance of the system for a single user isn't good, it won't be any better for multiple users. Single-user testing is conducted throughout application development life-cycle, during functional testing and user acceptance testing; gathering performance data during these stages can be extremely helpful. The single-user performance alone may facilitate the detection of some performance issues earlier. Single-user performance can provide a good indication of what business functions and application code needs to be investigated further. Additionally, between single-user tests and load tests there are also functional multi-user tests as described in Karen Johnson's article [Johnson08]. A well-constructed test with a few users can also help to identify a lot of problems which may be very difficult to diagnose during load testing.

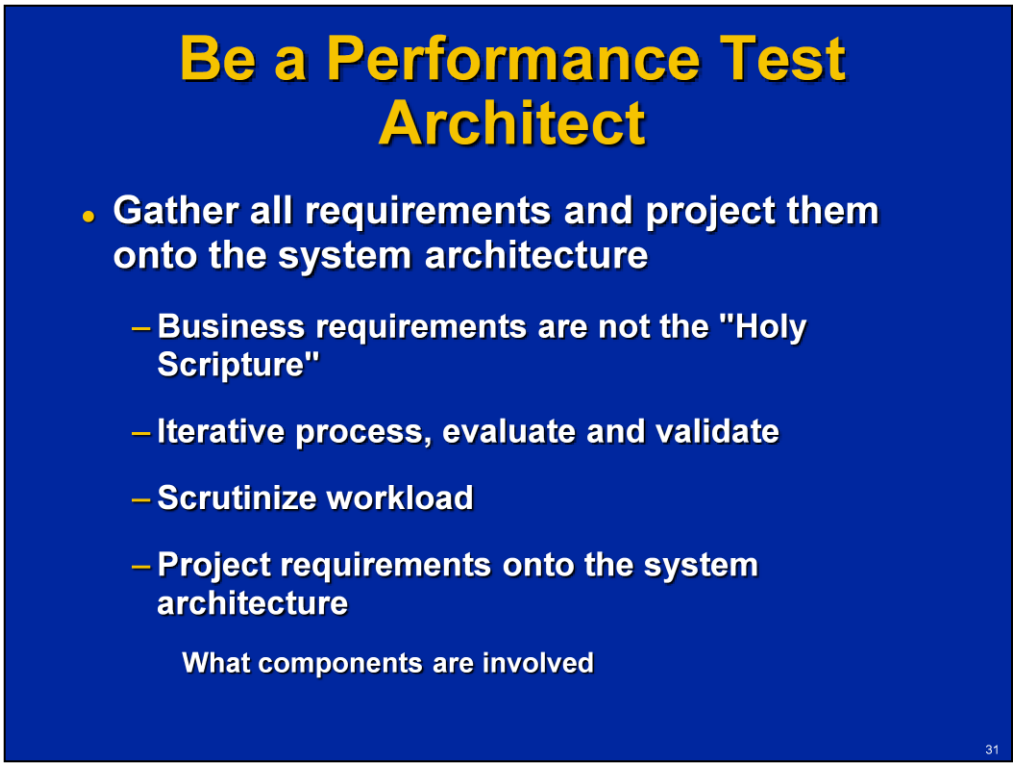


If you are around from the beginning the project and know that you will be involved, a few guerilla-style actions can save you (and the project) a lot of time and resources later. Even if it is only asking a few key questions. Don’t get until everything gets in place – do what is possible as early as possible.

Noting the importance of early performance work, quite often it is just not an option. Still the case when you get on the project just for pre-deployment performance testing is, unfortunately, typical enough. You need test the product for performance before going live as good as you can in the given timeframe – so the following sections discuss what you still can do in such situation.



I believe that the title of the Andy Grove book "Only the Paranoid Survive" [Grove96] relates even better to performance engineers than it does to executives. I can imagine an executive who isn't paranoid, but I can't imagine a good performance engineer without this trait. And it is applicable to the entire testing effort from the scenarios you consider, to the scripts you create, and the results you report.



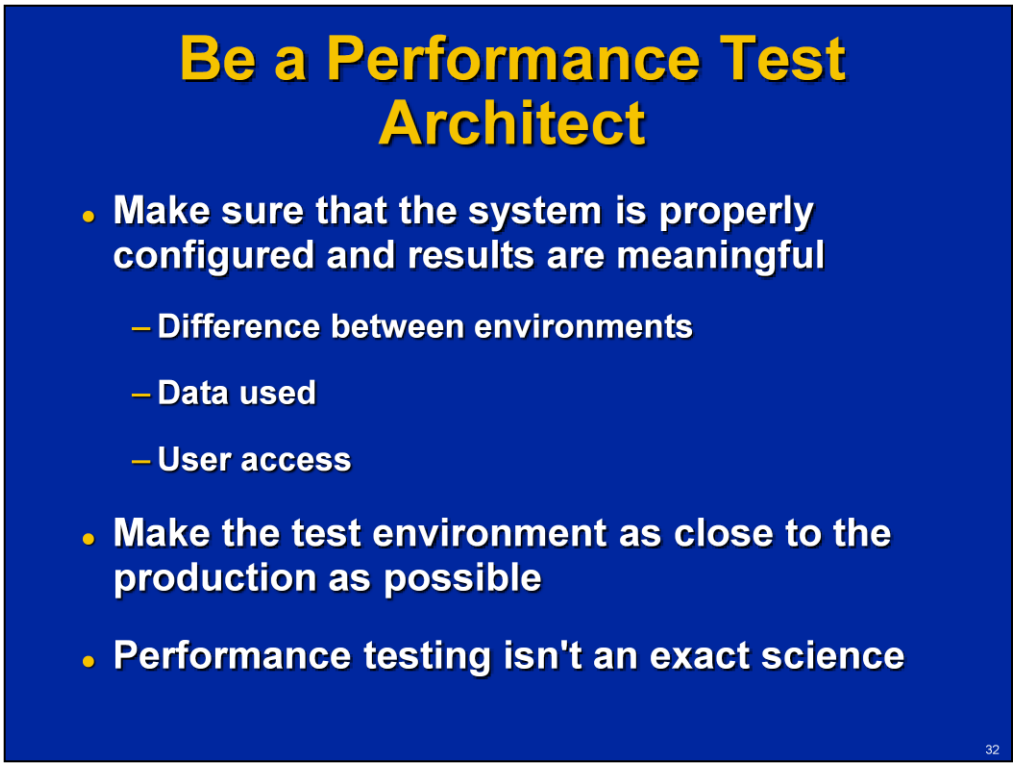
There are two large set of questions requiring architect-type expertise:

Gathering and validation of all requirements (first of all, workload definition) and projecting them onto the system architecture.

Too many testers consider all detailed information that they obtain from the business people (i.e., workload descriptions, scenarios, use cases, etc.) as the "holy script". But business people know the business, and they rarely know anything about performance engineering. So obtaining requirements is an iterative process and every requirement submitted should be evaluated and, if possible, validated [Podelko11]. Sometimes performance requirements are based on reliable data, sometimes they are just a pure guess, and it is important to understand how reliable they are.

The load the system should handle should be carefully scrutinized; the workload is an input to testing, while response times are output. You may decide if response times are acceptable even after the test – but you should define workload before.

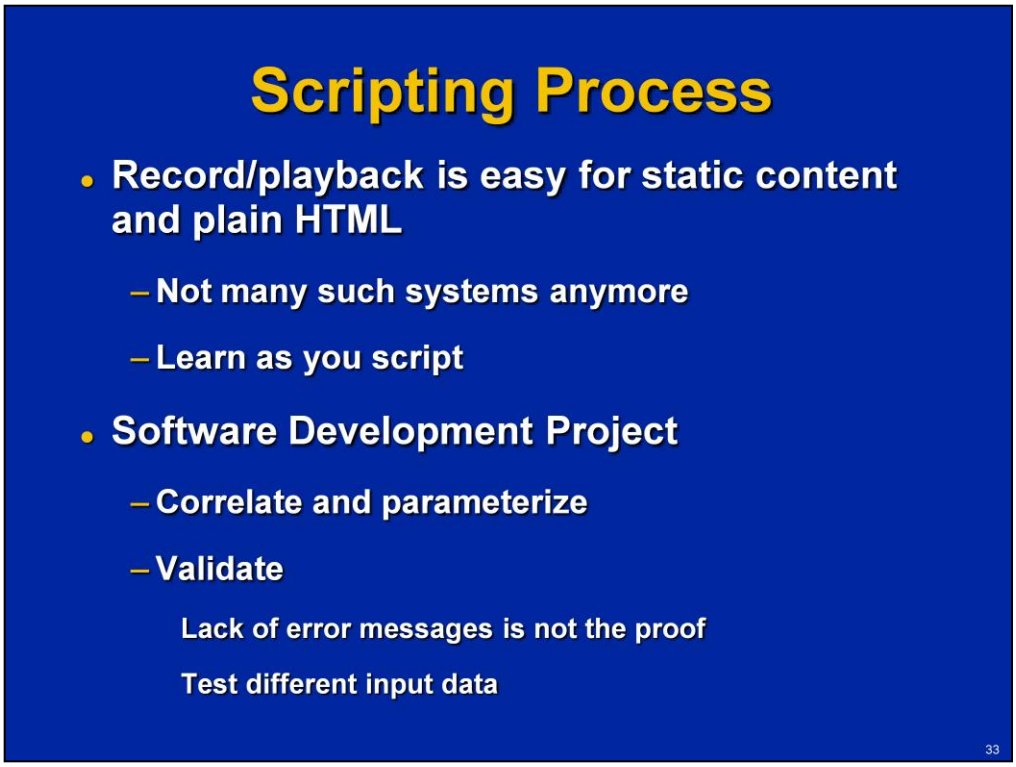
The gathered requirements should be projected onto the system architecture – it is important to understand if included test cases add value testing a different set of functionality or different components of the system. From another side, it is important to make sure that we have test cases for every component (or, if we don't, we know why).



Making sure that the system under test is properly configured and the results obtained may be used (or at least projected) for the production system. Environment and setup-related considerations can have a dramatic effect. Here are a few:
•What data are used? Is it real production data, artificially generated data, or just a few random records? Does the volume of data match the volume forecasted for production? If not, what is the difference?
•How are users defined? Do you have an account set with the proper security rights for each virtual user or do you plan to re-use a single administrator id?
•What are the differences between the production and the test environment? If your test system is just a subset of your production - can you simulate the entire load or just a portion of that load?

It is important to get the test environment as close as possible to the production environment, but some differences may still remain. Even if we executed the test in the production environment with the actual production data, it would only represent one point in time, other conditions and factors would also need to be considered. In real life the workload is always random, changing each moment, including actions that nobody could even guess.

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

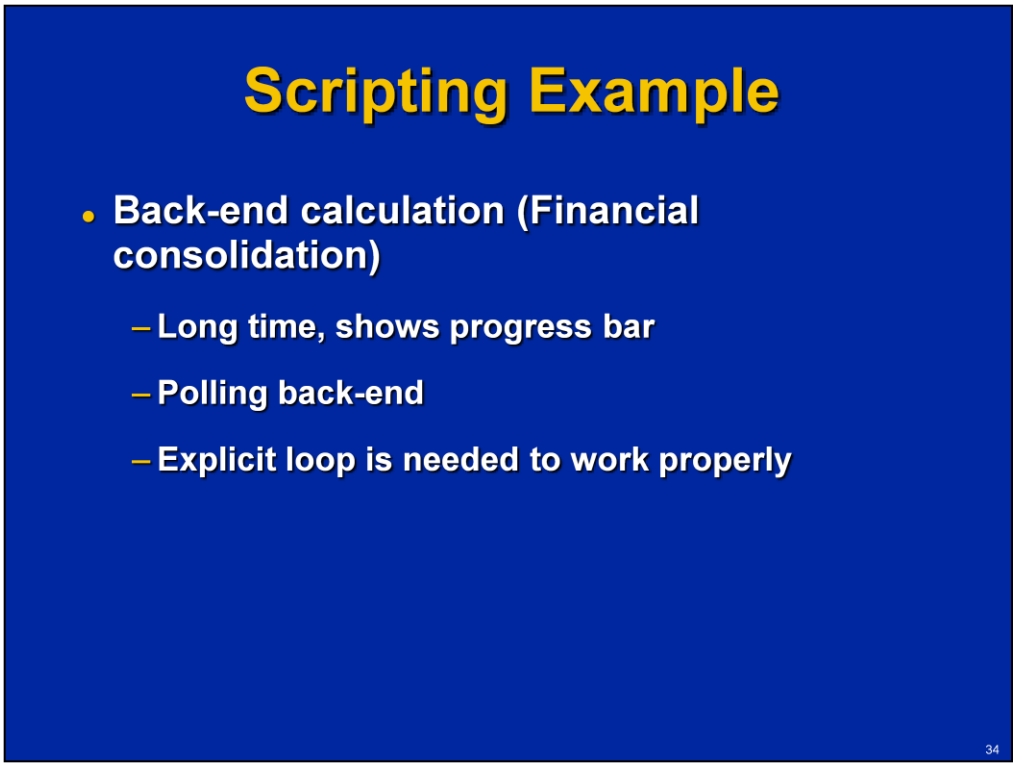


There are very few systems today that are stateless with static content using plain HTML, the kind of systems that lend themselves to a simplistic "record/playback" approach. In most cases there are many stumbling blocks in your way to create a proper workload. Starting from the approach you use to create the workload – the traditional "record/playback" approach just doesn't work in many cases [Podelko12]. If it is the first time you see the system there is absolutely no guarantee that you can quickly record and playback scripts to create the workload, if at all.

Creating performance testing scripts and other objects is, in essence, a software development project. Sometimes automatic script generation from recording is mistakenly interpreted as the whole process of script creation, but it is only the beginning. Automatic generation provides ready scripts in very simple cases; in most non-trivial cases it is just a first step. You need to correlate (get dynamic variables from the server) and parameterize (use different data for different users) scripts. These are operations prone to errors because we make changes directly in the communication stream. Every mistake is very dangerous because such mistakes usually can not happen in the real world where the users works with the system through a user interface or API calls.

After the script is created it should be evaluated for a single user, multiple users, and with different data. You should not assume that the system works correctly when the script was executed without errors. A very important part of load testing is workload validation. We should be sure that the applied workload is doing what it is supposed to do and that all errors are caught and logged. It can be done directly by analyzing server responses or, in cases when this is impossible, indirectly. It can be done, for example, by analyzing the application log or database for the existence of particular entries.

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



My group specializes in performance testing of financial analysis products. A few of scripting challenges exist for almost every product. Nothing exceptional – you should resolve them if you have some experience and would be attentive enough, but time after time we are called to save performance testing projects ("nothing works" or "the system doesn't perform") to find out that there are serious problems with scripts and scenarios which make test results meaningless. Even very experienced testers stumble, but problems could be avoided if more time were spent analyzing what is going on. Let's consider an example – probably typical for challenges you can face with modern Web-based applications.

Some operations, like financial consolidation, can take a long time. The client starts the operation on the server and then waits until it will finish, a progress bar is shown in meanwhile. When recorded, the script looks like (in LoadRunner pseudo-code):

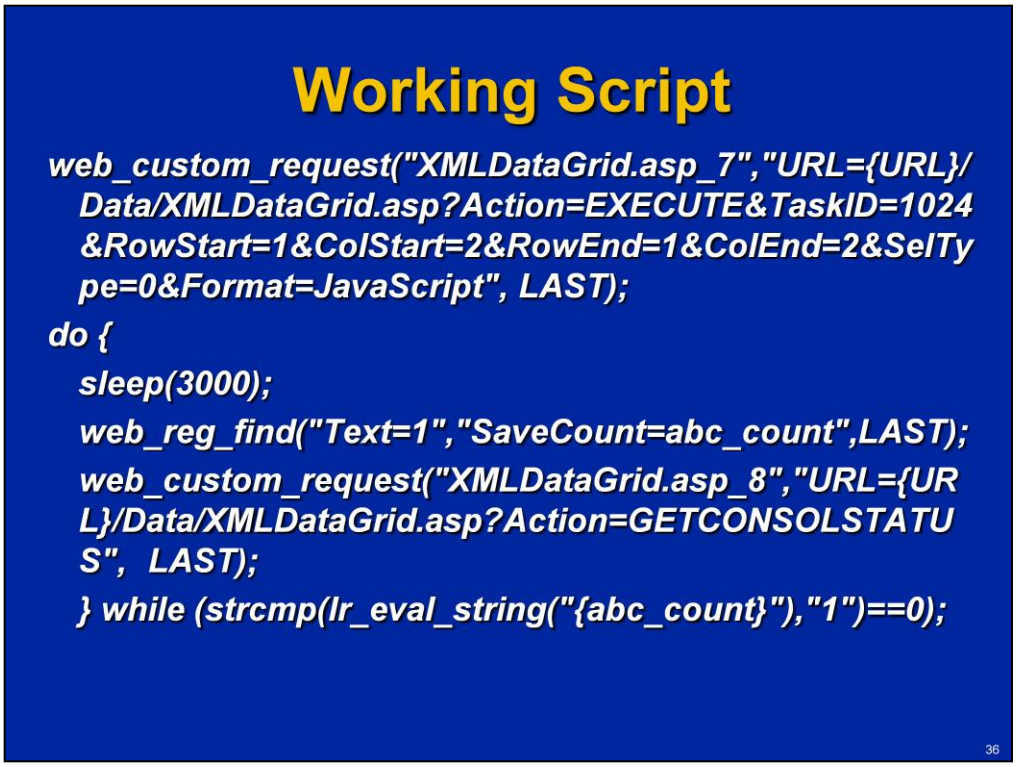
web_custom_request("XMLDataGrid.asp_7",
"URL={URL}/Data/XMLDataGrid.asp?Action=EXECUTE&TaskID=1024&RowStart=1&ColStart=2&RowEnd=1&ColEnd=2&SelType=0&Format=JavaScript", LAST);

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

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

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

What each request is doing is defined by the *?Action=* part. The number of *GETCONSOLSTATUS* requests recorded depends on the processing time. In the example above it was recorded three times; it means that the consolidation was done by the moment the third *GETCONSOLSTATUS* request was sent to the server. If playback this script, it will work in the following way: the script submits the consolidation in the *EXECUTE* request and then calls *GETCONSOLSTATUS* three times. If we have a timer around these requests, the response time will be almost instantaneous. While in reality the consolidation may take many minutes or even an hour (yes, this is a good example when sometimes people may be happy having one hour response time in a Web application). If we have several iterations in the script, we will submit several consolidations, which continue to work in background competing for the same data, while we report sub-second response times.

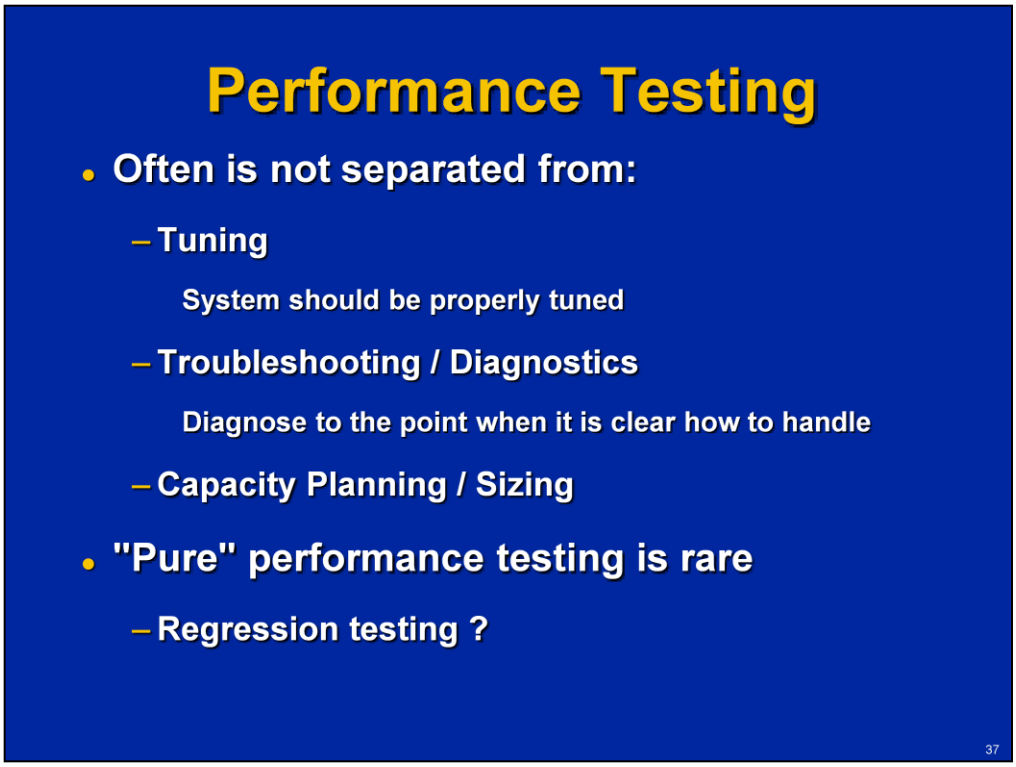


Consolidation scripts require creating an explicit loop around GETCONSOLSTATUS to catch the end of consolidation:

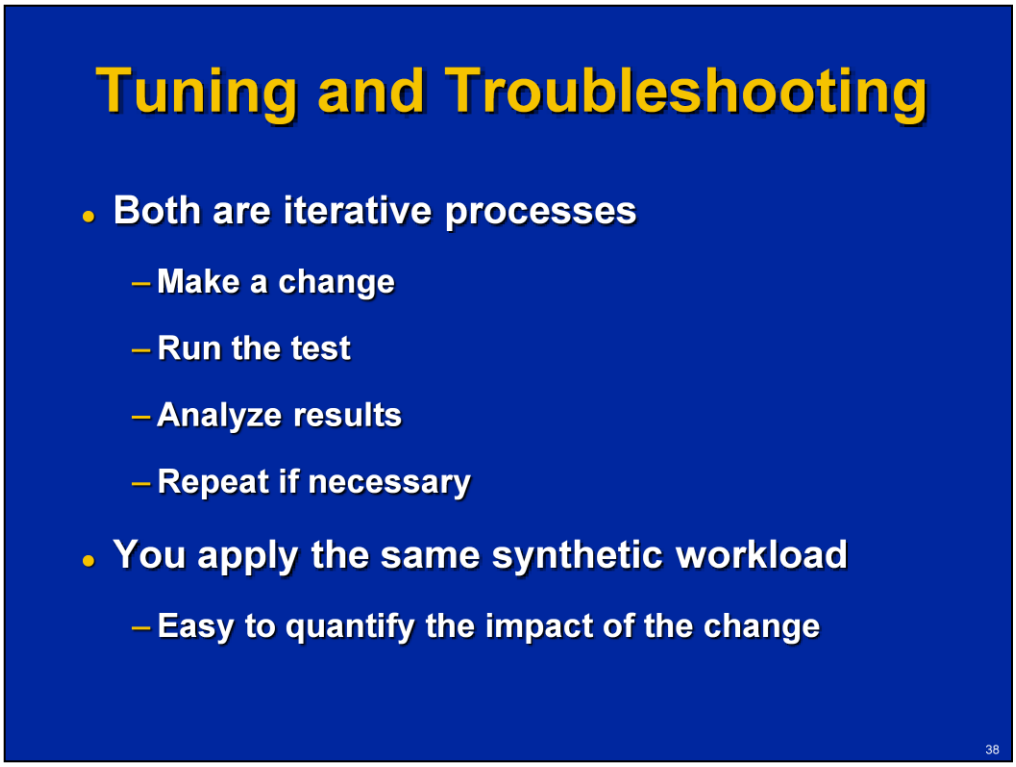
```
web_custom_request("XMLDataGrid.asp_7",
"URL={URL}/Data/XMLDataGrid.asp?Action=EXECUTE&TaskID=1024&RowStart=1&ColStart=2&RowEnd=1&ColEnd=2&SelType=0&Format=JavaScript", LAST);

do {
                sleep(3000);
                web_reg_find("Text=1", "SaveCount=abc_count", LAST);
                web_custom_request("XMLDataGrid.asp_8",
                "URL={URL}/Data/XMLDataGrid.asp?Action=GETCONSOLSTATUS",  LAST);
    } while (strcmp(lr_eval_string("{abc_count}"),"1")==0);
```

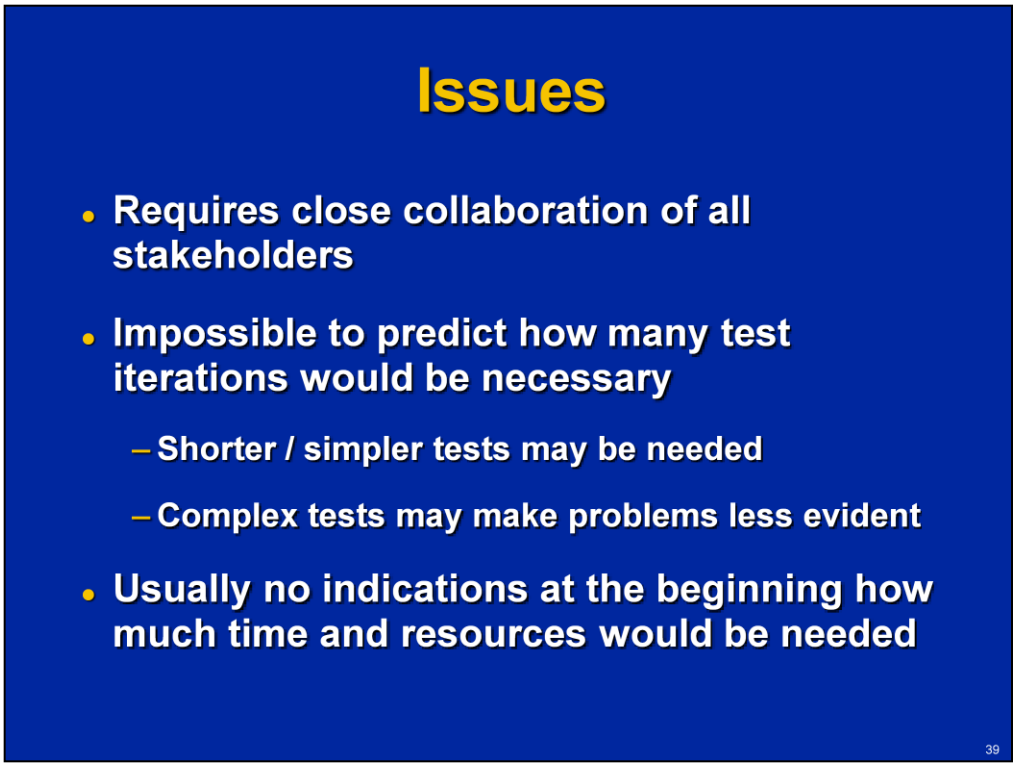
Here the loop simulates the internal logic of the system sending *GETCONSOLSTATUS* requests each 3 sec until the consolidation is done. Without such loop the script just checks the status and finishes the iteration while the consolidation continues for a long time after that.



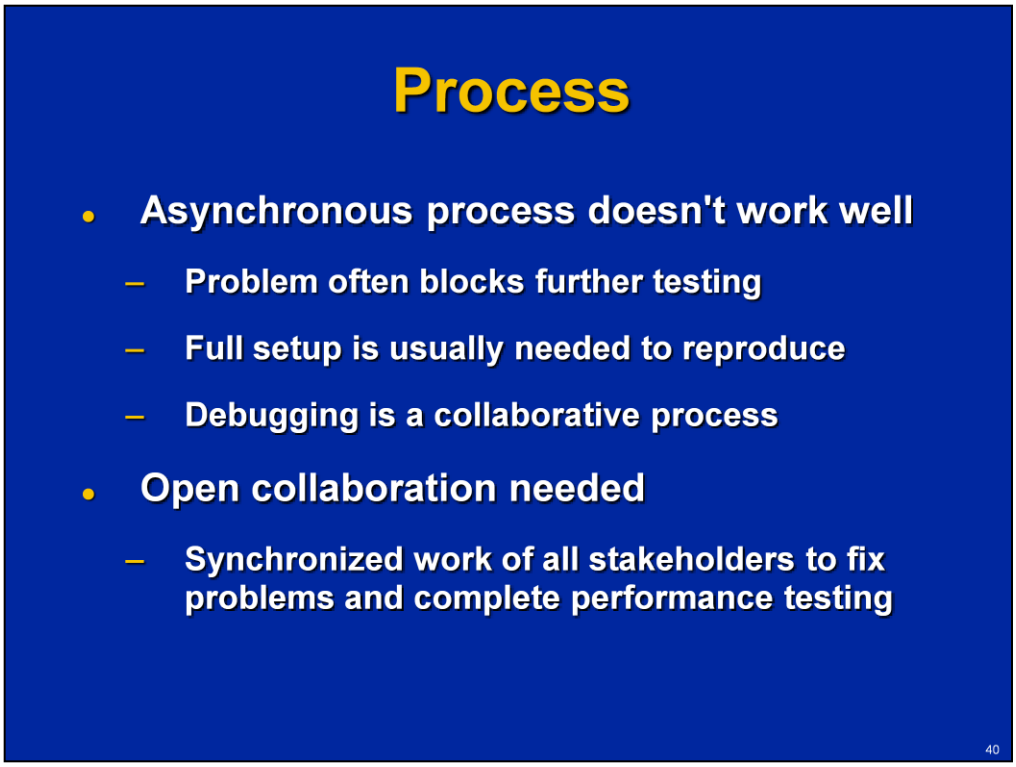
Usually, when people are talking about performance testing, they do not separate it from tuning, diagnostics, or capacity planning. “Pure” performance testing is possible only in rare cases when the system and all optimal settings are well known. Some tuning activities are usually necessary at the beginning of the testing to be sure that the system is properly tuned and the results are meaningful. In most cases, if a performance problem is found, it should be diagnosed further up to the point when it is clear how to handle it. Generally speaking, 'performance testing', 'tuning', 'diagnostics', and 'capacity planning' are quite different processes and excluding any of them from the test plan (if they are assumed) will make it unrealistic from the beginning.



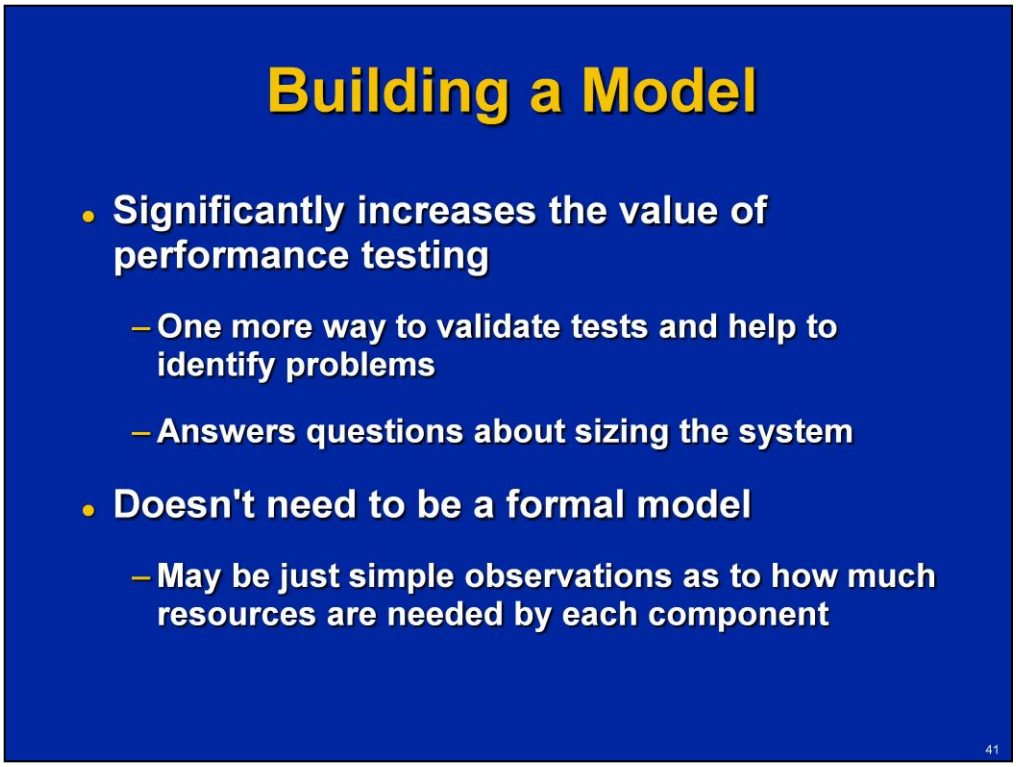
Both performance tuning and troubleshooting are iterative processes where you make the change, run the test, analyze the results, and repeat the process based on the findings. The advantage of performance testing is that you apply the same synthetic load, so you can accurately quantify the impact of the change that was made. That makes it much simpler to find problems during performance testing than wait until they happen in production where workload is changing all the time.



Still, even in the test environment, tuning and performance troubleshooting are quite sophisticated diagnostic processes usually requiring close collaboration between a performance engineer running tests and developers and/or system administrators making changes. In most cases it is impossible to predict how many test iterations would be necessary. Sometimes it makes sense to create a shorter and simpler test still exposing the problem under investigation. Running a complex, "real-life" test on each tuning or troubleshooting iteration can make the whole process very long as well as make the problem less evident due to different effects the problem may have on different workloads.

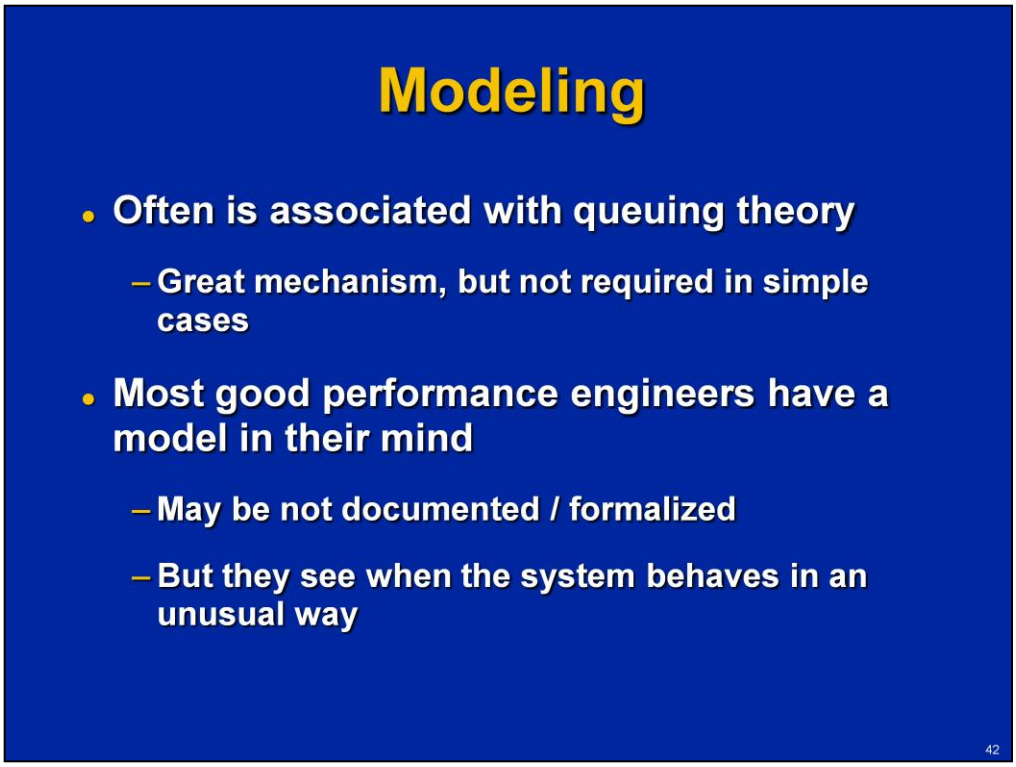


An asynchronous process to fixing defects, often used in functional testing - testers look for bugs and log them into a defect tracking system, and then the defects are prioritized and independently fixed by development – doesn't work well for performance testing. First, a reliability or performance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



Creating a model of the system under test is very important and significantly increases the value of performance testing. First, it is one more way to validate test correctness and help to identify problems with the system – if you see deviations from the expected behavior it may mean issues with the system or issues with the way you create workload (see good examples in [Gunther06]). Second, it allows answering questions about sizing and capacity planning of the system.

It doesn't need to be a formal model created by a sophisticated modeling tool (at least for the purpose of performance testing – if any formal model is required, it is a separate activity). It may be just simple observations how much resources on each component of the system are needed for the specific workload. For example, the workload A creates significant CPU utilization on the server X while the server Y is hardly touched. This means that if increase the workload A the lack of CPU resources on the server X will create a bottleneck. As you run more and more complex tests, you verify results you get against your "model", your understanding how the system behaves – and if they don't match, you need to figure out what is wrong.

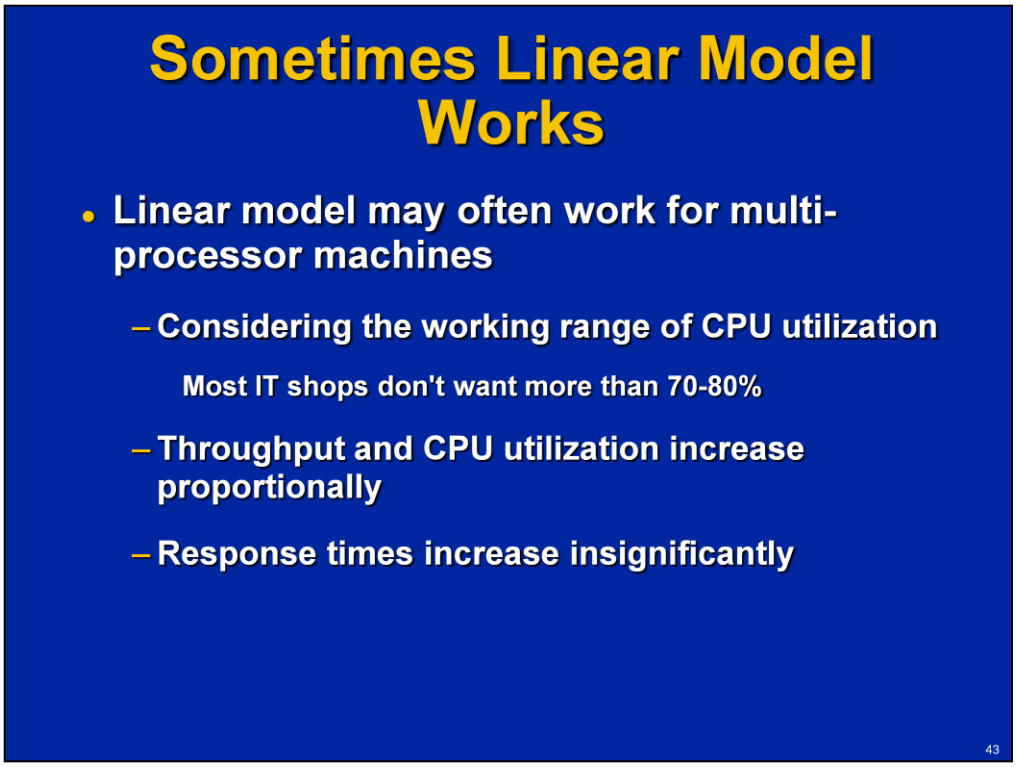


Modeling often is associated with queuing theory and other sophisticated mathematical constructs. While queuing theory is a great mechanism to build a sophisticated computer system models, it is not required in simple cases. Most good performance engineers and analyst build a model subconsciously, even without using such words or any formal efforts. While they don't describe or document this model in any way, they see when system behaves in an unusual way (doesn't match the model) and can do some simple predictions (for example, it doesn't look like we have enough resources to handle X users).

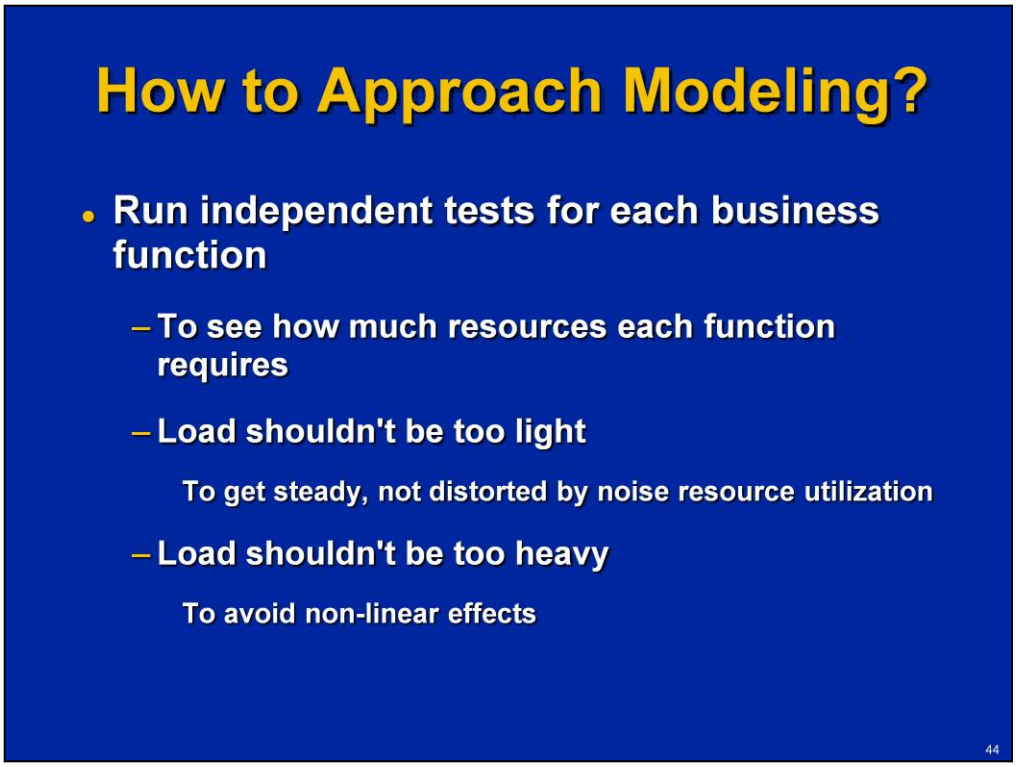
The best way to understand the system is to run independent tests for each business function to generate a workload resource profile. The load should be not too light (so resource utilization will be steady and won't be distorted by noise), nor too heavy (so resource utilization won't be distorted by non-linear effects).

This kind of tests is described as T2 in the "T1 – T2 – T3" approach to modeling [Gimarc04]. Gimarc's paper presents a methodology for constructing tests and collecting data that can be used both for load testing and modeling efforts. Three types of tests are considered in the paper. The first is T1, a single business function trace, used to capture the tier-to-tier workflow of each of the application's business function. The second is T2, a single business function load test, designed to determine the system resources required to process each business function on each server. The third is T3, which is a typical load test with multiple business functions, used to validate the model.

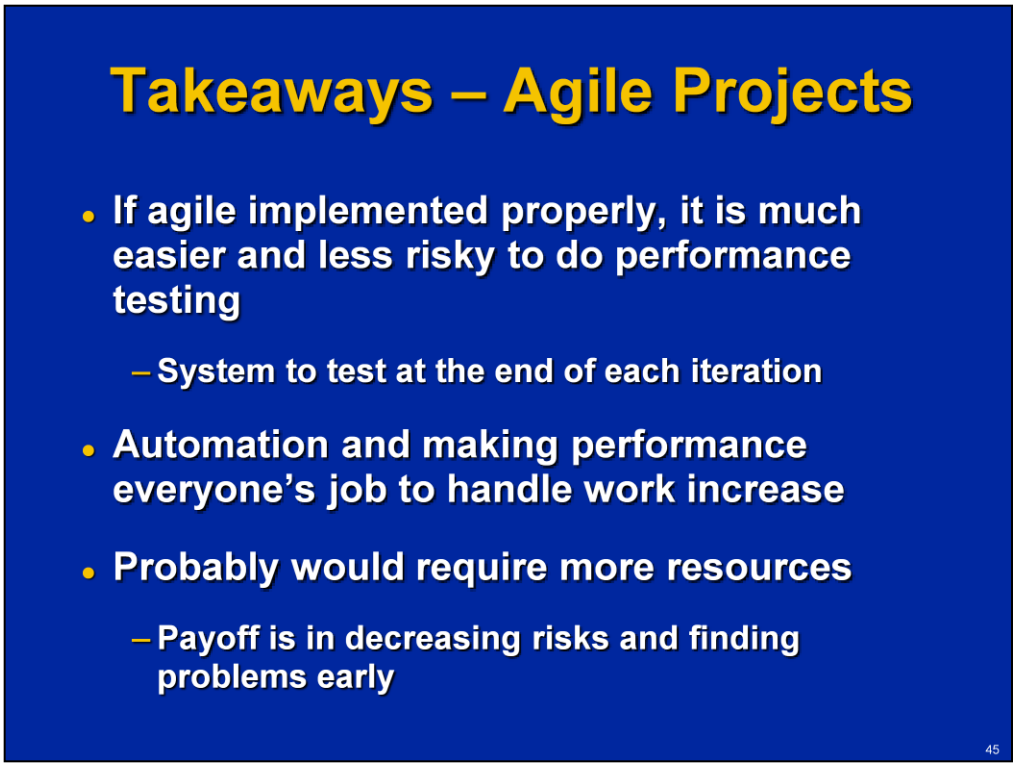
Another good example of such light-weight modeling is 'transaction cost analysis' described in [Capacitas13].



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



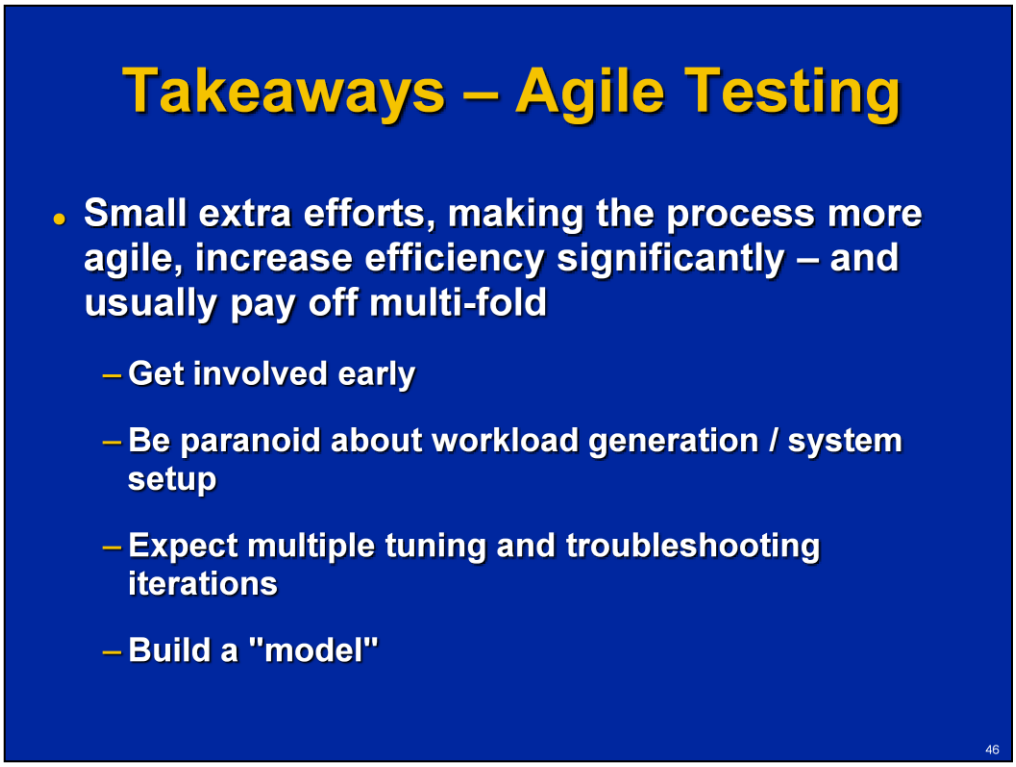
Running all scripts together makes it difficult to build a model. While you still can make some predictions for scaling the overall workload proportionally, it won't be easy to find out where is the problem if something doesn't behave as expected. The value of modeling increases drastically when your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



If agile methodology is implemented properly, it should be much easier and less risky to do performance testing as far as the system should be available for testing at the end of each iteration.

However, if you are involved from the beginning and testing the system each iteration as it gets delivered, you probably would need more resources comparing to pre-production performance testing. Payoff is in decreasing performance-related risks and finding problems early (which, unfortunately, is quite difficult to quantify comparing with the cost of resources you need to invest).

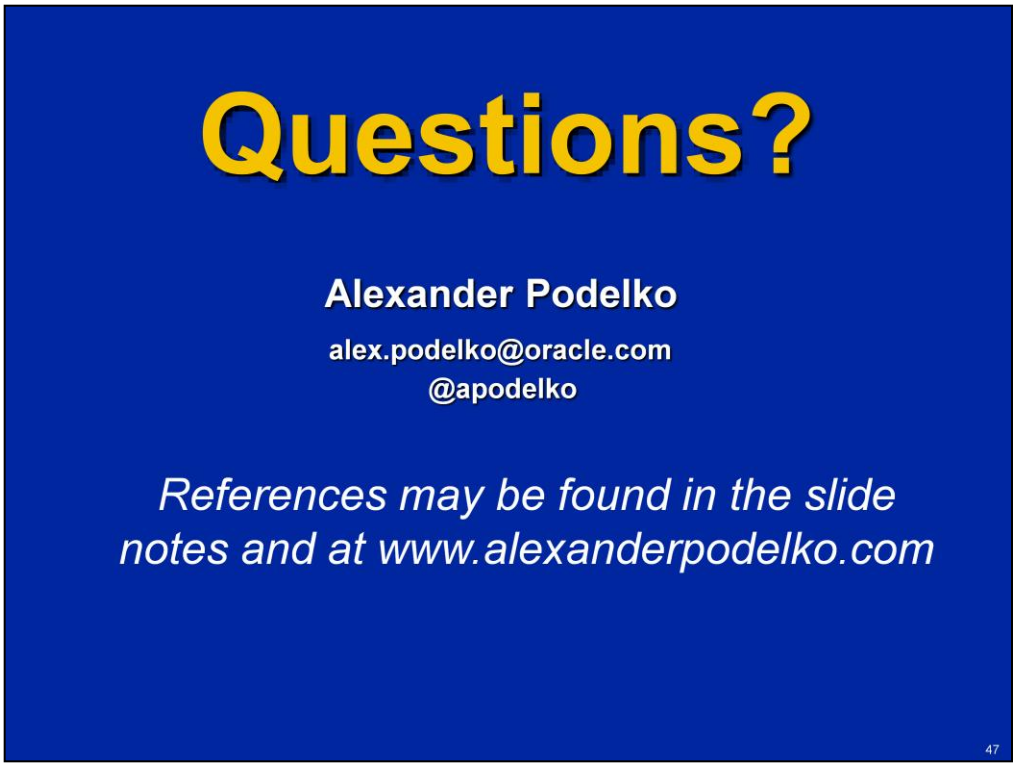
Automation and making performance everyone's job is the way to handle work increase and make it more manageable.



Even if the project scope is limited to pre-production performance testing, approaching testing with an agile, iterative approach you meet your goals faster and more efficiently and, of course, learn more about the system along the way. After we prepare a script for testing (or however the workload is generated), we can run one, a few, and many users (how many depends on the system), analyze results (including resources utilization), and try to sort out any errors. The source of errors can be quite different – script error, functional error, or a direct consequence of a performance bottleneck. It doesn't make much sense to add load until you figure out what is going on. Even with a single script you can find many problems and, at least partially, tune the system. Running scripts separately also allows you to see how much resources are used by each type of load and make some kind of system's "model".

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

Using an agile, iterative approach doesn't mean that you need to re-define the software development process, but rather find new opportunities inside existing processes. I believe that most good performance engineers are already doing performance testing in an agile way but just presenting it as "waterfall" to management (some kind of guerrilla tactic). In most cases you need to present a waterfall-like plan to management, and then you are free to do whatever is necessary to properly test the system inside the scheduled timeframe and scope. If opportunities exist, performance engineering may be extended further, for example, to early performance checkpoints or even full Software Performance Engineering. But don't wait until everything is properly in place, make the best possible effort and then look for opportunities to extend it further.



[Agile] Agile Software Development, Wikipedia. http://en.wikipedia.org/wiki/Agile_software_development

[AgileLoad12] Agile Performance Testing process, AgileLoad whitepaper, 2012. http://www.agileload.com/agileload//blog/2012/10/22/agile-performance-testing-process---whitepaper

[Bach03] Bach, J. Exploratory Testing Explained, 2003. http://people.eecs.ku.edu/~saiedian/teaching/Fa07/814/Resources/exploratory-testing.pdf

[Barber07] Barber, S. An Explanation of Performance Testing on an Agile Team, 2007. http://www.logigear.com/newsletter-2007/320-an-explanation-of-performance-testing-on-an-agile-team-part-1-of-2.html http://www.logigear.com/newsletter-2007/321-an-explanation-of-performance-testing-on-an-agile-team-part-2-of-2.html

[Barber11] Barber, S. Performance Testing in the Agile Enterprise, STP, 2011. http://www.slideshare.net/rsbarber/agile-enterprise

[Buksh11] Buksh, J. Performance By Design – an Agile Approach, 2011. http://www.perftesting.co.uk/performance-by-design-an-agile-approach/2011/11/18/

[Capacitas13] Improving the Value of Your Agile Performance Testing (Part 1), Capacitas's blog, 2013.

http://capacitas.wordpress.com/2013/05/20/improving-the-value-of-your-agile-performance-testing-part-1/

[Crispin09] Crispin L., Gregory, J. Agile Testing: A Practical Guide for Testers and Agile Teams. Pearson Education, 2009.

[Dobson07] Dobson, J. Agile Performance Testing, Agile, 2007 http://agile2007.agilealliance.org/downloads/proceedings/075_Performance%20Testing%20on%20an%20Agile%20Project_617.pdf

[Harding11] Harding A. Continuous Integration - A Performance Engineer's Tale, 2011. http://www.slideshare.net/sthair/continuous-integration-a-performance-engineers-journey

[Hazrati11] Hazrati, V. Nailing Down Non-Functional Requirements, InfoQ, 2011. http://www.infoq.com/news/2011/06/nailing-quality-requirements

[Gimarc04] Gimarc R., Spellmann A., Reynolds J. Moving Beyond Test and Guess. Using Modeling with Load Testing to Improve Web Application Readiness, CMG, 2004.

[Grove96] Grove A. Only the Paranoid Survive, Doubleday, New York, 1996.

[Gunther06] Gunther N. Benchmarking Blunders and Things That Go Bump in the Night, MeasureIT, June 2006 (Part I), August 2006 (Part II). http://www.cmg.org/measureit/issues/mit32/m_32_2.html http://www.cmg.org/measureit/issues/mit34/m_34_1.html

[Gunther07] Gunther N. Guerilla Capacity Planning: A Tactical Approach to Planning for Highly Scalable Applications and Services, Springer-Verlag, 2007.

[Johnson08] Johnson K. Multi-User Testing, Software Test & Performance, February 2008, 20-23.

[Kua09] Kua P. Top Ten Secret Weapons For Agile Performance Testing, 2009. http://www.slideshare.net/melnykenator/top-ten-secret-weapons-for-agile-performance-testing

[Manifesto01] Manifesto for Agile Software Development, 2001 http://agilemanifesto.org/

[OpTier12] How OpTier's Continuous Performance Testing Process Ensures Our Industry Leading Technology, 2012. http://www.optier.com/blog/how-optiers-continuous-performance-testing-process-ensures-our-industry-leading-technology/

[Pepperdine06] Pepperdine K. Performance Anti-patterns, 2006. http://huettermann.net/media/performance_anti-patterns.pdf

[Performance07] Performance Testing Guidance for Web Applications. Chapter 6, Managing an Agile Performance Test Cycle, 2007. http://perftestingguide.codeplex.com/Wiki/View.aspx?title=Chapter%206%20%u2013%20Managing%20an%20Agile%20Performance%20Test%20Cycle

[Podelko06] Podelko A. Load Testing: Points to Ponder, CMG, 2006.

[Podelko08] Podelko, A. Agile Performance Testing, CMG, 2008.

[Podelko11] Podelko A. Performance Requirements: An Attempt of a Systematic View, CMG, 2011.

[Podelko12] Podelko A. Load Testing: See a Bigger Picture, CMG, 2012.

[Ries11] Ries, E. The Lean Startup, Crown Business, 2011.

[Smith02] Smith C., Williams L. Performance Solutions, Addison-Wesley, 2002.

[Waterfall] Waterfall Model, Wikipedia. http://en.wikipedia.org/wiki/Waterfall_model