

Performance Testing on an Agile Project

Jamie Dobson
Dobson Coaching and Consulting
jamie@jamiedobson.co.uk

Abstract

This experience report is about a software process, designed with performance testing in mind, was used to create a mission critical integration layer. This report focuses on how the team carried out performance tests iteratively, how a large array of processes and techniques were employed to support a specialist performance test team, and how changes in management style were needed to support performance testing on an agile project.

1. Introduction

Iterative performance testing is difficult in so called traditional environments. An example of one such environment is a software department that is segregated into specialist teams that supply each other with work in a batch-and-queue manner. Another example would be an organisation that employed a development methodology modelled, incorrectly, on Royce's waterfall [2]. A worse example is a department that has segregated teams *and* uses the waterfall model. This latter example is the context in which I worked as a team lead on a large scale Java project that I will further refer to as project or product *Daedalus*.

Daedalus is an integration layer used to decouple my client's client-applications from an array of bespoke and third party booking and scheduling applications.

Loads on *Daedalus* were extremely high and response times were important. Therefore, a scalable, and performant system was necessary. The team was having difficulties using XP to deliver their system. I was asked to help recover the project.

I started by focusing on what I feel are the most important practices for developing software: collaborative planning, whole team involvement,

customer tests, and time-boxed iterations. In the beginning, whilst talking to the team members & stakeholders and conducting my own Theory-W research, the issue of performance was brought to my attention.

In the past, poorly performing systems produced by developers had been a source of conflict within the organisation. Senior managers were very keen to satisfy their customers by delivering performant systems. This led me to design a coherent strategy where performance testing became an extremely visible and an integral part of the development process.

1.1. Agile Performance Testing Defined

Agile performance testing, in the author's opinion, would include self-contained development teams, including specialists in performance testing; push-button deployment and fully, or near fully, automated performance tests; an array of developer techniques to address performance testing and optimisation; and, importantly, tests and non-functional requirements that were developed iteratively in a collaborative manner, based on user expectations. In addition to this, the development methodology would need to be iterative, adopt a pull-based task system, and focus strongly on quality assurance activities.

1.2. Team Structure and Methodology

The structure of the organisation we worked within is typical of many batch-and-queue software organisations. Teams of developers produce software that is queued up for a separate, often non-located, test team to verify. The requirements are fed to the team by another separate, often non-located, analyst team. Projects are planned sequentially but development usually takes place in an an-hoc concurrent manner with the requirements team finishing use cases while the development team implement a previous batch.

However, the testing of software is always carried out towards the end of the project. The functional testers are usually the analysts who created the requirements. The performance testing is carried out in a separate department by a team of performance test specialists. A further team is responsible for deploying applications to the company's web-farm. The way to get an application performance tested is to book a test slot and a deployment slot. The deployment team then deploys the application for the performance testers to test against.

1.3. Staff

The teams were made up of a number of consultants from my company; a number of company employees; and a number of contractors. However, the development team was formed of consultants from the company I worked for.

Apart from the obvious deficiencies in this organisation model and the ad-hoc waterfall methodology, there was a lot of bad blood between the deployment & performance testing teams and the development teams. The deployment teams were sick of trying to deploy applications that simply wouldn't deploy. Once an application was deployed it was almost impossible to performance test due to bugs in the system. When the deployment team or the performance testers became recalcitrant they were put under tremendous pressure by management to "get the application deployed" or "get it through its performance test". This naturally led to bad feeling and, in some cases, the gaming of test results.

The developers also had a lot to complain about. They were put under pressure to produce code against use cases that had never been reviewed with little access to the analysts who wrote them. To keep the appearance of progress they would do anything to get the software working so they could "throw it over the wall" to the deployment team. Once over the wall an expensive code-and-fix cycle would begin leading to long hours and ultimately the delivery of poor software.

The functional testers, of course, were subject to the same pressure as the performance testers and would accept a system just so it could be rushed through the performance testing stage. It seemed that pushing the systems through gates on a spreadsheet was more important than actually creating working software.

2. Three Levels of Performance Testing: An Integrated Approach

Three types of performance testing were carried out on the project: developer; deployed; and customer or beta testing.

2.1. Beta or Customer Testing

Because the team were developing an integration component for other applications to use, it was useful to release the software as quickly as possible. Each application that used our component had its own performance testing strategy. In order to take advantage of this, we released as soon as possible to receive feedback from other teams.

The general lesson here is that performance testing against real user data and real user expectations can add tremendous value to a project. However, only agile development will guarantee a working system on any day, i.e. only software developed using agile techniques can be used to exploit early customer feedback. This is a good example of performance testing on an agile project and later we will see that always having a working product impacts performance testing in at least one advantageous, non-foreseeable, way.

2.2. Deployed

Deployed testing is where software is deployed onto life-like systems for testing. Deployed tests are the most common, and probably most inefficient way, to detect performance bugs. They are also a very obvious blocking factor to carrying out performance testing from within an agile project. However, they are also the most important type of test since they test a system in a real world setting.

The problem most development teams face is access to the expensive, and often overworked, deployment and performance testing resources. This was exactly the scenario facing the *Daedalus* team. To circumvent this, and take at least one step towards agile performance testing, the teams agreed to deliver working, deployable, code every six weeks.

The lesson here is to shorten the cycle as much as the context of the project and organisation allows. The most important step for my team, and other teams in general, is to convince the management, the development team, and the performance testers to work in an iterative manner.

The following section discusses how to achieve iterative deployed tests from both a project management and technical standpoint.

2.3. Human Aspects

The single most important thing to circumvent batch-and-queue organisations and thinking is to break down the virtual walls that separate teams. The first thing I did was explain to the performance testers the problems with post-development testing. At the same time I listened to the problems performance testers face. The strategy I came up with tried to address all stakeholder concerns.

The two biggest problems the performance testers expressed with an iterative approach were:

- 1) They did not like the idea of performance testing against a moving target. They were worried that testing an application only to have to re-test it again in an iteration's time was going to be a waste. For example, they argued if we were using XP and allowing our customers to change their minds then the baselines the team created would be invalid against the new requirements.
- 2) The performance testing was such a painful process i.e. it was so difficult to get the application deployed and running the deployment and performance-testing teams would rather do a handful of deployments instead of potentially ten or twenty per project.

It is up to the agile project manager to convince the performance testers that their efforts are not wasted, and that a more iterative approach will stop a big-bang performance test much later in the project when it is can be very painful. In the case of *Daedalus*, I took a bottom up and top down approach to setting up an iterative testing cycle.

Firstly, the requirements of the project were well known, although not necessarily well specified. This enabled us to design early and often and thus settle on the architecture of the system early. The decent requirements coupled with the extra design work helped ease the minds of the performance testers that there wouldn't be too many performance surprises later on in the project. The main problem, in general, is that most agile processes are adaptive, designed to accommodate, and in some cases encourage, change. If a system does change beyond its original incarnation

then agile performance testing may be possible, but it would also probably be a waste of time.

Secondly, using the build tool *Ant* and configuration templates we were able to create an application that would deploy easily. To support this we also created many other technical solutions to demonstrate that the application was deployed and functioning properly

Thirdly, I used my influence to convince senior management to enforce this new mandate. The use of a number of executive sponsors to apply top-down support also helped with buy-in. Once underway it was obvious the new way of working was better so the whole process became self-sustaining. Top-down, heavy-handed, barrel of the gun approaches to enforce change can be dangerous, and should be used sparingly. However, sometimes people are so stubborn that a bit of a nudge can help them see that a more agile way of working can be beneficial.

2.4. Project Management

When moving from a waterfall model of working to an iterative way of working new planning techniques are necessary. Typically at my client's workplace, the deployment and performance testing slots were booked in large blocks towards the end of the project. However, since we wanted to work in an iterative manner, traditional planning needed to be abandoned.

The management had to give up their component based planning approach and give into a feature-driven approach. This is the only way to achieve plan for iterative testing and development. This meant the project manager could safely book the slots once every six weeks and he knew the slots would be utilized fully, even though he wasn't sure how many features would be tested. Basically, we stopped treating time as a variable in the plan and let scope become the variable.

Using evolutionary planning, where the plan was updated every two weeks, the manager's vision of the features complete became clearer as time progressed. The important lesson here is that even agile teams, when faced with dependencies on an external performance testing team, have to plan.

2.5. Blitz and Agile Planning

Blitz planning is a technique where teams brainstorm what needs to be done in a coming time period. Dependencies are identified and can therefore be developed in the correct order. Blitz planning is an ideal way to roughly plan a six-week Sprint and therefore it was a great way for the development team to plan to release the software to the performance team. The outcome of a blitz planning session is a rough plan that can be incorporated into the larger more traditional, release plan. Every two weeks the release plan was refined based on how much the team did in the iteration.

Blitz Planning helped to focus the team around a release, identify the activities they needed to do. Planning like this is an exercise in collaborative working and helps to add control and visibility to a project as well as negating some of the problems with traditional planning.

2.6. Focus and Structure

The team worked in two-week iterations within a six-week release cycle. The last iteration was where the deployment for the performance test team occurred. The six-week cycle is based on the Scrum Sprint.

The last iteration, weeks five and six of the Sprint, was when the two rounds of performance testing occurred. The first round was in week five, usually early in week five, and the second in week six. The two-slot-deal allowed problems found in week five to be fixed ready for further testing in week six. The additional project management helped to plan deployment slots and actively force the team to work within six-week release cycles. This approach, this two-shot-deal, actually helped the development team to utilise their slots wisely.

Usually, one developer would fully support the deployment and testing. This helped to get the performance testing and deployment working smoothly. It also helped enable communication between the development and performance-testing teams.

2.7. Working Software & ‘Hustle’

Previously I mentioned some unforeseen benefits of always having working software. During the Sprint if another development team, for whatever reason, were not prepared for their deployment and performance slots the *Daedalus* team could utilise

them. This was achievable via the agile project management technique known as hustle.

Hustle is a term taken from the game of baseball. A player in the outfield, on bases, starts to move before the pitcher throws. The players who are hustling keep on their toes and are ready to either sprint to the next base, or jump back, depending on what the batter does. This short, intense, activity is only used when the pitcher is winding up, and not throughout the game. Similarly, in response to emergencies, agile teams hustle too. Hustle was used by my team to quickly deploy the application and take advantage of any empty deployment and performance testing slots.

The team always had working software due to the XP practices. They also had a suite of automated performance tests. XP, coupled with hustle, agile planning and automated performance regression tests is a big step toward agile performance testing proper.

2.8. Publish Everything

Publish everything is more a philosophy than a technique. With Publish Everything the team tries to publicize test results, configuration information, performance testing targets, and anything else that is important to the project. In relation to performance testing, the smoke test is a very important aspect of Publish Everything.

2.9. Smoke Test

The daily build and smoke test are both well-known quality assurance techniques. Many software teams assume that by using continuous integration they are already doing a regular build and smoke test. However, this is not the case. With continuous integration you test against one configuration, the build server configuration. In enterprise development there are many dependencies between systems. These, in the case of *Daedalus*, includes multiple databases, messaging systems, customer relationship management tools, be-spoke systems, payment systems, and object brokers or Enterprise Java Bean servers, to name the ones I can remember.

A smoke test that is deployed with an application is an asset to the deployment and performance test teams. The smoke test allows these teams to quickly verify and diagnose any faults in the deployed application. These faults could be configuration errors or they could be because a third party system is down.

The *Daedalus* team deployed integration tests with their application. The tests were used to verify that all configuration files were correct; that communication to dependent systems worked; and that all dependent systems were running. The performance testers used the integration tests to verify everything was working before they started their tests. Additionally, should a performance test begin to fail they could re-run the smoke test to ensure everything was still working properly.

The smoke test was available via the project's dashboard. This meant that it also provided the development team a way to check the application was deployed properly.

2.10. Manual Automatic Testing

This technique is where the development team creates a framework to test an application. Our web site included ways to automatically load scripts, alter parameters, and basically interact with the system. This tool, in addition to the smoke test, was really helpful for the performance testers. They could use the tool to verify their test data, and carry out other rudimentary tests.

2.11. Push-Button Deployment

When something goes wrong with the deployment or with a performance test the development team may need to quickly fix the problem and build a new version. To this end *Daedalus* had a fully functional build system. In general, without this, you are not agile and you cannot carry out agile testing of any kind.

2.12. Summary of Deployed Testing

Integrating dedicated hardware and human resources with an iterative development model is hard. In order to do this at least three things should be done:

- 1) In terms of **staff** a clear vision and a sense of team has to be created. Upper management must be supportive of the iterative approach and teams have to understand each other's viewpoints. Techniques such as *Soap Boxing*, *Theory-W*, and *Drinks in a Bar* can help to foster mutual respect.
- 2) On the **project management** level, iterative development must be adopted along with a feature-driven approach to planning and developing. The project manager must foster learning and an attitude of total quality to keep

their staff motivated and capable enough to 'hustle'.

- 3) The **development team** must provide technical support to the deployment and performance testers. Support can come in two forms:
 - a. **Manual.** The developers support deployments and testing.
 - b. **Automatic.** The developers create tools to help diagnose a deployed application. With frameworks like *Cactus* and *JUnit* this is easy to accomplish. If in container tests are developed as part of the development then they are very easy to leverage as a smoke test.

We have so far looked at how team *Daedalus* carried out and supported iterative testing in the context of a six-week Sprint. The next section looks at the activities the team performed on a daily basis.

2.12. Developer Testing

The third and final type of performance testing on our project was developer testing. Developer testing cannot replace deployed testing. But, by paying attention to performance issues and expanding some XP practices, the development team greatly reduced the burden on the performance testing team. The practices we used included *agile optimisation* and *local regression and load testing*.

2.13. Local Load Testing/Explorative Load Testing

Usually a feature is done when it is written, tested and accepted by a customer. However, often a unit test and an acceptance test will not reveal performance problems or bugs associated with load. For example, any threading problems or memory leaks will not be found with a simple unit test.

Threading and memory leaks are, in my opinion, usually quite trivial to find and fix. To go through the process of deploying and performance testing an application just to reveal, for example, that a developer forgot to close a data base connection, is an extravagant waste of time. Therefore, our use cases or stories were considered done only after a load test, that was executed locally, passed.

This was achieved by simply generating a heavy load for a sustained period of time. We did this with the open source tool *JMeter*. This allowed us to remove any trivial errors in advance of the

performance test. Once created the local load tests can be added to a suite and executed overnight or over the weekend.

2.14. Local Load Tests and Deployed Tests

When the performance test team found a problem the development team always endeavoured to reproduce it locally. As with all bugs on an agile project, the first thing to do is reproduce it locally via a failing test. The team could then get the test passing and add it to the regression suite. This failing test could be captured as a unit test, an integration test, or if heavy sustained load was needed, a JMeter test.

Obviously, the development server is different to the production server, thus we treated the local performance tests as supplementary. Therefore, once a performance bug was fixed, we still needed to retest in the deployed environment. Local tests are useful, but not conclusive. This type of testing was a great way for the performance test team and the development team to supplement each other's activities.

2.15. Agile Performance Optimisation

Agile optimisation is a simple procedure for optimising methods. Usually when the performance of a system needs to be improved there are one or two methods where all the time is spent. Agile optimisation works by decorating existing unit tests with a timing element.

The process is very simple:

- 1) Profile the application and identify the bottleneck.
- 2) Find out the average runtime of the method and create a baseline.
- 3) Decorate the test with the desired runtime.
- 4) Optimise the method.
- 5) Repeat until finished.

This process is very simple to grasp, however, we learnt some lessons:

- 1) The baselines cannot be trusted. The baseline will depend on the amount of processing power of the developer's machine and how much memory it has. Additionally, how many processes the operating system is running will also influence the baseline. Therefore, not only are the baselines not valid across machines, they are not valid on the same machine at different times.

- 2) Automation is really difficult due to the different performance characteristics of each machine. Additionally, using an evolutionary design strategy will almost certainly mean that the runtime of a method will change.
- 3) Unit tests often don't use user data. Unit tests are usually happy path tests. That means the data they use may not be representative of real user data. Therefore, optimisations against poor data could be false optimisations. For agile performance optimisation to work it is crucial to have access to user data.

Agile performance optimisation is useful to optimise a single method or sub-system. However, it must be carried out with care and the results must be verified by the performance test team.

2.16. Design, Requirements and Performance

There is a relationship between requirements and the amount of design that should be carried out. Additionally, the quicker a team converges on a design the quicker the performance baselines will stabilise. Because a large proportion of the requirements were recognized we were able to come up with a design that persisted for the duration of the project. This added value to the project because the earlier performance tests were still valid late in the project.

Therefore, extra requirements activities, such as creating FIT tables, can assist in the early creation of a stable design. This in turn will allow early iterative performance testing.

2.17. Summary of Developer Techniques

XP coding techniques can be expanded to carry out continuous performance testing and fix performance bugs in a test-first manner. Developer techniques supplement deployed techniques and are necessary for a coherent performance testing strategy.

2.18. Summary of Integrated Approach

I have explained how the *Daedalus* team carried out performance testing at three levels. Together, this integrated approach enabled us to deliver a system that met all of its performance requirements. The synergy created between the performance testing team and the development team helped to address performance issues early and often.

From a developer perspective the two best techniques were probably the deployed smoke tests and local load tests. These two techniques provide quick, albeit crude, feedback about the system. Agile optimisation is useful but is somewhat dangerous, this was only used a couple of times on the project.

From a team leadership perspective the most productive technique used was probably allowing the team to take responsibility of the project. The reversal of the task system, from push to pull, helps to create a culture of responsibility.

From the performance testers point of view the iterative testing and smoke tests probably helped the most. Instead of unofficially debugging the system or finding trivial problems with system performance they were freed to carry out their duties. When things did go wrong the support of the development team was greatly appreciated. The testing tools, of course, made diagnosing problems very easy.

3. Conclusions

Performance testing on an agile project is, theoretically, really easy. Change the model, get everyone motivated, and use a few XP practices. However, the hard part of anything agile is not knowing the path, but walking it. This takes dedication, energy, and the design of good processes that suit the context. This is what we had on *Daedalus*.

Daedalus was a success due to the people on the project. The team proved that iterative performance testing is possible and a very good way of working. I cannot really say if true agile performance testing is possible. My instincts tell me that it would be too expensive because of the resources needed and the complete waste of time it is to baseline a changing system. Agile performance testing may be possible as more and more developers become skilled in performance testing. Scott Ambler describes a generalising specialist as:

“A generalizing specialist is someone with a good grasp of how everything fits together. As a result they will typically have a greater understanding and appreciation of what their teammates are working on. They are willing to listen to and work with their teammates because they know that they’ll likely learn something new. Specialists, on the other hand, often don’t have the background to appreciate what other specialists are doing, often look

down on that other work, and often aren’t as willing to cooperate. Specialists, by their very nature, can become a barrier to communication within your team.”[1]

The creation of multi-skilled teams, like the *Daedalus* team, and the integration of local performance testing techniques into the development methodology may be one way to remove the performance testing specialist from the process and thus remove them as a process bottleneck. More automation and access to life-like hardware could enable the rapid deployment agile teams need. I feel that the *Daedalus* team took an important first step towards agile performance testing and in time this way of performance testing could become common with other agile teams.

4. Bibliography/Further Reading

[1] Ambler, Scott. *Generalizing Specialists: Improving Your IT Career Skill*. Ambler, 2007. URL: <http://www.agilemodeling.com/essays/generalizingSpecialists.htm>

[2] Royce, Winston W. Royce, “Managing the Development of Large Software Systems”, *Proceedings of IEEE WESCON 26*, August 1970.

McConnell, Steve. *Software Estimation: The Black Art Demystified*. Microsoft Press, 2006.

McConnell, Steve. *Rapid Development*. Microsoft Press, 1996.

Beck, Kent. *Extreme Programming Explained*. Addison-Wesley, 2000.

Cohn, Mike. *Agile Estimation and Planning*. Prentice Hall, 2006.

Highsmith, Jim. *Agile Project Management*. Addison-Wesley, 2004.

Schwaber, Ken. *Agile Project Management with Scrum*. Microsoft Press, 2004.

Brookes, Fred. *The Mythical Man Month*. Addison-Wesley, 1995.

Gamma et al. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, 1995.

Mugridge, Rick & Cunningham, Ward. *Fit for Developing Software*. Prentice Hall, 2004.

4.1. Hyperlinks

Framework For Integrated Tests, <http://fit.c2.com/>

FitNesse, <http://fitnesse.org/>

JMeter, <http://jakarta.apache.org/jmeter/>

Cactus, <http://jakarta.apache.org/cactus/>