O'REILLY®

/theory/in/practice

The Art of Application Performance Testing

From Strategy to Tools

Ian Molyneaux

2nd Edition

O'REILLY®

/ theory / in / practice

The Art of Application Performance Testing

Because performance is paramount today, this thoroughly updated guide shows you how to test mission-critical applications for scalability and performance *before* you deploy them—whether it's to the cloud or a mobile device. You'll learn the complete testing process lifecycle step-by-step, along with best practices to plan, coordinate, and conduct performance tests on your applications.

- Set realistic performance testing goals
- Implement an effective application performance testing strategy
- Interpret performance test results
- Cope with different application technologies and architectures
- Understand the importance of End User Monitoring (EUM)
- Use automated performance testing tools
- Test traditional local applications, web applications, and web services
- Recognize and resolves issues often overlooked in performance tests

Written by a consultant with over 15 years' experience with performance testing, *The Art of Application Performance Testing* thoroughly explains the pitfalls of an inadequate testing strategy and offers a robust, structured approach for ensuring that your applications perform well and scale effectively when the need arises.

Ian Molyneaux, EMEA SME (Subject Matter Expert), is Head of Performance at Intechnica, a software consultancy based in Manchester, UK. He specializes in performance assurance for the enterprise with a strong focus on people, process, and tooling.

US $44.99 **CAN $47.99**

ISBN: 978-1-491-90054-3

9 781491 900543 54499

Twitter: @oreillymedia
facebook.com/oreilly
oreilly.com

Software Testing

The Art of Application Performance Testing

by Ian Molyneaux

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Andy Oram and Brian Anderson
Production Editor: Melanie Yarbrough
Copyeditor: Rachel Monaghan
Proofreader: Sharon Wilkey

Indexer: Wendy Catalano
Interior Designer: David Futato
Cover Designer: Ellie Volkhausen
Illustrator: Rebecca Demarest

December 2014: Second Edition

Revision History for the Second Edition

2014-12-08: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491900543* for release details.

978-1-491-90054-3

[LSI]

TABLE OF CONTENTS

CHAPTER ONE

Why Performance Test?

Faster than a speeding bullet . . .

Superman, Action Comics

WELCOME! BEFORE DIVING INTO THE BASICS OF PERFORMANCE TESTING, I WANT to use this first chapter to talk a little about what we mean by good and bad performance and why performance testing is such a vital part of the software development life cycle (SDLC). Non-performant (i.e., badly performing) applications generally don't deliver their intended benefit to an organization; they create a net cost of time and money, and a loss of kudos from the application users, and therefore can't be considered reliable assets. If a software application is not delivering its intended service in a performant and highly available manner, regardless of causation, this reflects badly on the architects, designers, coders, and testers (hopefully there were some!) involved in its gestation.

Performance testing continues to be the poor, neglected cousin of functional and operational acceptance testing (OAT), which are well understood and have a high maturity level in most business organizations. It is strange that companies continue to overlook the importance of performance testing, frequently deploying applications with little or no understanding of their performance, only to be beset with performance and scalability problems soon after the release. This mindset has changed little over the past 15 years, despite the best efforts of consultants like myself and the widely publicized failure of many high-profile software applications. (Need we mention HealthCare.gov?)

What Is Performance? The End-User Perspective

So when is an application considered to be performing well? My many years of working with customers and performance teams suggest that the answer is ultimately one of perception. A well-performing application is one that lets the end user carry out a given task without undue *perceived* delay or irritation. Performance really is in the eye of the beholder. With a performant application, users are never greeted with a blank

screen during login and can achieve what they set out to accomplish without their attention wandering. Casual visitors browsing a website can find what they are looking for and purchase it without experiencing too much frustration, and the call-center manager is not being harassed by the operators with complaints of poor performance.

It sounds simple enough, and you may have your own thoughts on what constitutes good performance. But no matter how you define it, many applications struggle to deliver even an acceptable level of performance when it most counts (i.e., under conditions of peak loading). Of course, when I talk about application performance, I'm actually referring to the sum of the parts, since an application is made up of many components. At a high level we can define these as the client, the application software, and the hosting infrastructure. The latter includes the servers required to run the software as well as the network infrastructure that allows all the application components to communicate. Increasingly this includes the performance of third-party service providers as an integral part of modern, highly distributed application architectures. The bottom line is that if any of these areas has problems, application performance is likely to suffer. You might think that *all* we need do to ensure good application performance is observe the behavior of each of these areas under load and stress and correct any problems that occur. The reality is very different because this approach is often "too little, too late," so you end up dealing with the symptoms of performance problems rather than the cause.

Performance Measurement

So how do we go about measuring performance? We've discussed end-user perception, but in order to accurately measure performance, we must take into account certain *key performance indicators* (KPIs). These KPIs are part of the nonfunctional requirements discussed further in Chapter 3, but for now we can divide them into two types: service-oriented and efficiency-oriented.

Service-oriented indicators are availability and response time; they measure how well (or not) an application is providing a service to the end users. *Efficiency-oriented* indicators are throughput and capacity; they measure how well (or not) an application makes use of the hosting infrastructure. We can further define these terms briefly as follows:

Availability
: The amount of time an application is available to the end user. Lack of availability is significant because many applications will have a substantial business cost for even a small outage. In performance terms, this would mean the complete inability of an end user to make effective use of the application either because the application is simply not responding or response time has degraded to an unacceptable degree.

Response time
: The amount of time it takes for the application to respond to a user request. In performance testing terms you normally measure system response time, which is the time between the end user requesting a response from the application and a complete reply arriving at the user's workstation. In the current frame of reference a response can be synchronous (blocking) or increasingly asynchronous, where it does not necessarily require end users to wait for a reply before they can resume interaction with the application. More on this in later chapters.

Throughput
: The rate at which application-oriented events occur. A good example would be the number of hits on a web page within a given period of time.

Utilization
: The percentage of the theoretical capacity of a resource that is being used. Examples include how much network bandwidth is being consumed by application traffic or the amount of memory used on a web server farm when 1,000 visitors are active.

Taken together, these KPIs can provide us with an accurate idea of an application's performance and its impact on the hosting infrastructure.

Performance Standards

By the way, if you were hoping I could point you to a generic industry standard for good and bad performance, you're (still) out of luck because no such guide exists. There continue to be various informal attempts to define a standard, particularly for browser-based applications. For instance, you may have heard the term *minimum page refresh time*. I can remember a figure of 20 seconds being bandied about, which rapidly became 8 seconds and in current terms is now 2 seconds or better. Of course, the application user (and the business) wants "instant response" (in the words of the Eagles, "Everything all the time"), but this sort of consistent performance is likely to remain elusive.

The following list summarizes research conducted in the late 1980s (Martin et al., 1988) that attempted to map end-user productivity to response time. The original research was based largely on green-screen text applications, but its conclusions are still very relevant.

Greater than 15 seconds
: This rules out conversational interaction. For certain types of applications, certain types of end users may be content to sit at a terminal for more than 15 seconds waiting for the answer to a single simple inquiry. However, to the busy call-center operator or futures trader, delays of more than 15 seconds may seem intolerable.

If such delays could occur, the system should be designed so that the end user can turn to other activities and request the response at some later time.

Greater than 4 seconds
: These delays are generally too long for a conversation, requiring the end user to retain information in short-term memory (the end user's memory, not the computer's!). Such delays would inhibit problem-solving activity and frustrate data entry. However, after the completion of a transaction, delays of 4 to 15 seconds can be tolerated.

2 to 4 seconds
: A delay longer than 2 seconds can be inhibiting to operations that demand a high level of concentration. A wait of 2 to 4 seconds can seem surprisingly long when the end user is absorbed and emotionally committed to completing the task at hand. Again, a delay in this range may be acceptable after a minor closure. It may be acceptable to make purchasers wait 2 to 4 seconds after typing in their address and credit card number, but not at an earlier stage when they may be comparing various product features.

Less than 2 seconds
: When the application user has to remember information throughout several responses, the response time must be short. The more detailed the information to be remembered, the greater the need for responses of less than 2 seconds. Thus, for complex activities, such as browsing products that vary along multiple dimensions, 2 seconds represents an important response-time limit.

Subsecond response time
: Certain types of thought-intensive work (such as writing a book), especially with applications rich in graphics, require very short response times to maintain end users' interest and attention for long periods of time. An artist dragging an image to another location must be able to act instantly on his next creative thought.

Decisecond response time
: A response to pressing a key (e.g., seeing the character displayed on the screen) or to clicking a screen object with a mouse must be almost instantaneous: less than 0.1 second after the action. Many computer games require extremely fast interaction.

As you can see, the critical response-time barrier seems to be 2 seconds, which, interestingly, is where expected application web page response time now sits. Response times greater than 2 seconds have a definite impact on productivity for the average end user, so our nominal page refresh time of 8 seconds for web applications is certainly less than ideal.

The World Wide Web and Ecommerce

The explosive growth of the World Wide Web has contributed in no small way to the need for applications to perform at warp speed. Many (or is that all?) ecommerce businesses now rely on cyberspace for the lion's share of their revenue in what is the most competitive environment imaginable. If an end user perceives bad performance from your website, her next click will likely be on *your-competition.com*.

Ecommerce applications are also highly vulnerable to sudden spikes in demand, as more than a few high-profile retail companies have discovered at peak shopping times of the year.

Bad Performance: Why It's So Common

OK, I've tried to provide a basic definition of good and bad performance. It seems obvious, so why do many applications fail to achieve this noble aspiration? Let's look at some common reasons.

The IT Business Value Curve

Performance problems have a nasty habit of turning up late in the application life cycle, and the later you discover them, the greater the cost and effort to resolve. Figure 1-1 illustrates this point.

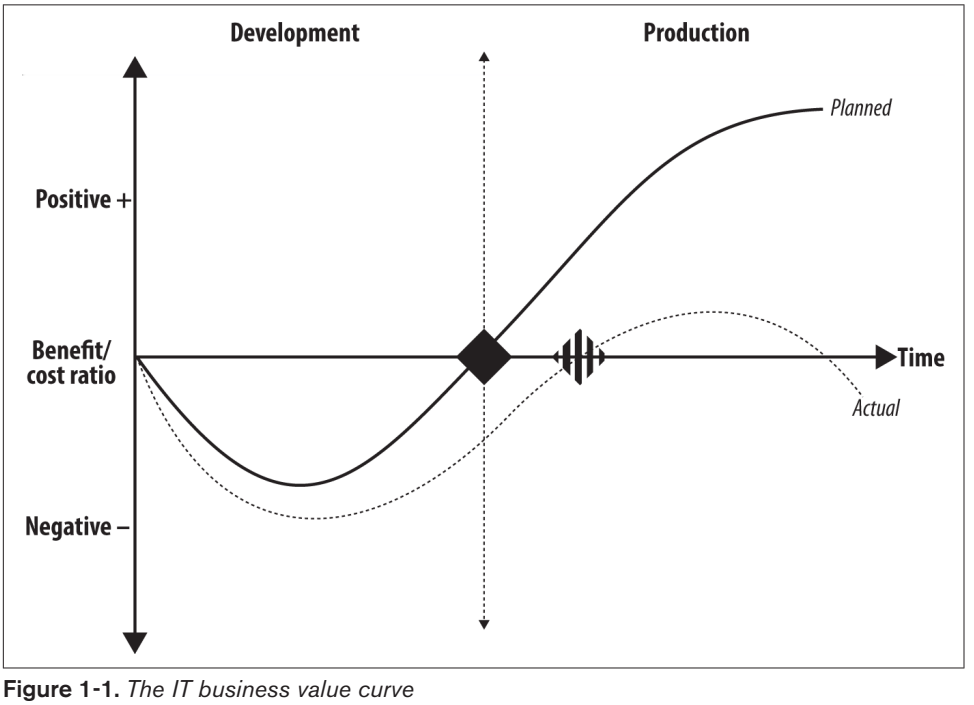

Figure 1-1. *The IT business value curve*

The solid line (planned) indicates the expected outcome when the carefully factored process of developing an application comes to fruition at the planned moment (black diamond). The application is deployed successfully on schedule and immediately starts to provide benefit to the business with little or no performance problems after deployment.

The broken line (actual) demonstrates the all-too-frequent reality when development and deployment targets slip (striped diamond) and significant time and cost is involved in trying to fix performance issues in production. This is bad news for the business because the application fails to deliver the expected benefit.

This sort of failure is becoming increasingly visible at the board level as companies seek to implement information technology service management (ITSM) and information technology portfolio management (ITPM) strategies on the way to the holy grail of Information Technology Infrastructure Library (ITIL) compliance. The current frame of reference considers IT as just another (important) business unit that must operate and deliver within budgetary constraints. No longer is IT a law unto itself that can consume as much money and resources as it likes without challenge.

Performance Testing Maturity: What the Analysts Think

But don't just take my word for it. Figure 1-2 is based on data collected by Forrester Research in 2006 looking at the number of performance defects that have to be fixed in production for a typical application deployment. For the revised edition I was considering replacing this example with something more recent, but on reflection (and rather disappointingly) not much has really changed since 2009!

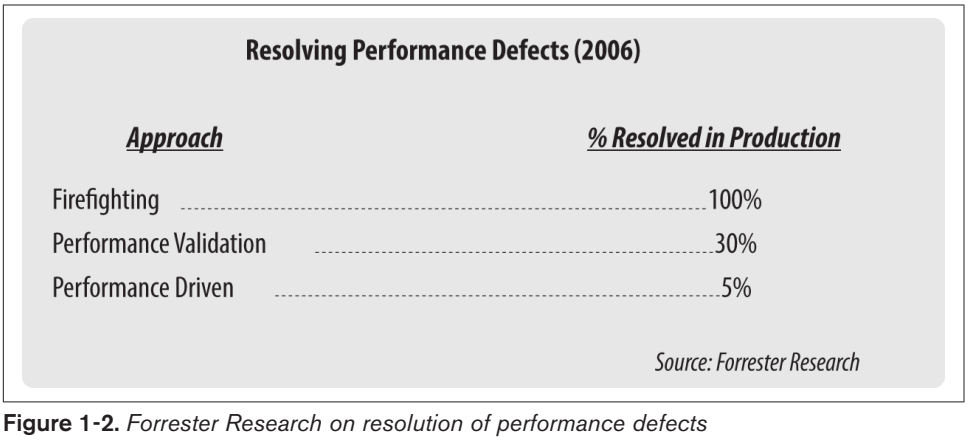

Figure 1-2. *Forrester Research on resolution of performance defects*

As you can see, three levels of performance testing maturity were identified. The first one, *firefighting*, occurs when little or no performance testing was carried out prior to application deployment, so effectively all performance defects must be resolved in the

live environment. This is the least desirable approach but, surprisingly, is still relatively common. Companies in this mode are exposing themselves to serious risk.

The second level, *performance validation* (or verification) covers companies that set aside time for performance testing but not until late in the application life cycle; hence, a significant number of performance defects are still found in production (30%). This is where most organizations currently operate.

The final level, *performance driven*, is where performance considerations have been taken into account at every stage of the application life cycle. As a result, only a small number of performance defects are discovered after deployment (5%). This is what companies should aim to adopt as their performance testing model.

Lack of Performance Considerations in Application Design

Returning to our discussion of common reasons for failure: if you don't take performance considerations into account during application design, you are asking for trouble. A "performance by design" mindset lends itself to good performance, or at least the agility to change or reconfigure an application to cope with unexpected performance challenges. Design-related performance problems that remain undetected until late in the life cycle can be difficult to overcome completely, and doing so is sometimes impossible without significant (and costly) application reworking.

Most applications are built from software components that can be tested individually and may perform well in isolation, but it is equally important to consider the application as a whole. These components must interact in an efficient and scalable manner in order to achieve good performance.

Performance Testing Is Left to the Last Minute

As mentioned, many companies operate in performance validation/verification mode. Here performance testing is done just before deployment, with little consideration given to the amount of time required or to the ramifications of failure. Although better than firefighting, this mode still carries a significant risk that you won't identify serious performance defects—only for them to appear in production—or you won't allow enough time to correct problems identified before deployment.

One typical result of this mode is a delay in the application rollout while the problems are resolved. An application that is deployed with significant performance issues will require costly, time-consuming remedial work after deployment. Even worse, the application might have to be withdrawn from circulation entirely until it's battered into shape.

All of these outcomes have an extremely negative effect on the business and on the confidence of those expected to use the application. You need to test for performance issues as early as you can, rather than leave it to the last minute.

Scalability

Often, not enough thought is given to capacity requirements or an application's ability to scale. The design of the application and the anticipated deployment model may overlook the size and geography of the end-user community. Many applications are developed and subsequently tested without more than a passing thought for the following considerations:

- How many end users will actually use the application?
- Where are these end users located?
- How many of these end users will use it concurrently?
- How will the end users connect to the application?
- How many additional end users will require access to the application over time?
- What will the final application landscape look like in terms of the number and location of the servers?
- What effect will the application have on network capacity?

Neglect of these issues manifests itself in unrealistic expectations for the number of concurrent end users that the application is expected to support. Furthermore, there is often little thought given to end users who may be at the end of low-bandwidth, high-latency WAN links. I will cover connectivity issues in more detail in Chapter 2.

Underestimating Your Popularity

This might sound a little strange, but many companies underestimate the popularity of their new web applications. This is partly because they deploy them without taking into account the novelty factor. When something's shiny and new, people generally find it interesting and so they turn up in droves, particularly where an application launch is accompanied by press and media promotion. Therefore, the 10,000 hits you had carefully estimated for the first day of deployment suddenly become 1,000,000 hits, and your application infrastructure goes into meltdown!

Putting it another way, you need to plan for the peaks rather than the troughs.

Spectacular Failure: A Real-World Example

Some years ago, the UK government decided to make available the results of the 1901 census on the Internet. This involved a great deal of effort converting old documents into a modern digital format and creating an application to provide public access.

I was personally looking forward to the launch, since I was tracing my family history at the time and this promised to be a great source of information. The site was launched and I duly logged in. Although I found things a little slow, I was able to carry out my initial searches without too much issue. However, when I returned to the site 24 hours later, I was greeted with an apologetic message saying that the site was unavailable. It remained unavailable for many weeks until finally being relaunched.

This is a classic example of underestimating your popularity. The amount of interest in the site was far greater than anticipated, so it couldn't deal with the volume of hits. This doesn't mean that no performance testing was carried out prior to launch. But it does suggest that the performance and importantly *capacity* expectations for the site were too conservative.

You have to allow for those peaks in demand.

Performance Testing Is Still an Informal Discipline

As mentioned previously, performance testing is still very much an informal exercise. The reason for this is hard to fathom, because functional testing has been well established as a discipline for many years. There is a great deal of literature and expert opinion available in that field, and many established companies specialize in test consulting.

Back in 2009 the converse was true, at least in terms of reference material. One of the reasons that I was prompted to put (virtual) pen to paper was the abject lack of anything in the way of written material that focused on (static) software performance testing. There were and still are myriad publications that explain how to tune and optimize an application, but little about how to carry out effective performance testing in the first place.

In 2014 I am pleased to say that the situation has somewhat improved and any Google search for performance testing will now bring up a range of companies offering performance testing services and tooling, together with a certain amount of training, although this remains very tooling-centric.

Not Using Automated Testing Tools

You can't carry out effective performance testing without using automated test tools. Getting 100 (disgruntled) staff members in on a weekend (even if you buy them all lunch) and strategically deploying people with stopwatches just won't work. Why?

You'll never be able to repeat the same test twice. Furthermore, making employees work for 24 hours if you find problems is probably a breach of human rights.

Also, how do you possibly correlate response times from 100 separate individuals, not to mention what's happening on the network and the servers? It simply doesn't work unless your application has fewer than 5 users, in which case you probably don't need this book.

A number of vendors make great automated performance testing tools, and the choice continues to grow and expand. Costs will vary greatly depending on the scale of the testing you need to execute, but it's a competitive market and biggest is not always best. So you need to do your homework and prepare a report for those who control your IT budget. Appendix C contains a list of the leading vendors. Chapter 2 talks more on how to choose the right performance tool for your requirements.

Application Technology Impact

Certain technologies that were commonly used in creating applications didn't work well with the first and even second generation of automated test tools. This has become a considerably weaker excuse for not doing any performance testing, since the vast majority of applications are now web-enabled to some degree. Web technology is generally well supported by the current crop of automated test solutions.

The tech-stack choices for web software development have crystallized by now into a (relatively) few core technologies. Accordingly, most automated tool vendors have followed suit with the support that their products provide. I will look at current (and some legacy) application technologies and their impact on performance testing in Chapter 9.

Summary

This chapter has served as a brief discussion about application performance, both good and bad. I've touched on some of the common reasons why failure to do effective performance testing leads to applications that do not perform well. You could summarize the majority of these reasons with a single statement:

> Designing and testing for performance is (still) not given the importance it deserves as part of the software development life cycle.

In the next chapter we move on to a discussion of why automation is so important to effective performance testing and how to choose the most appropriate automation solution for your requirements.