O'REILLY®

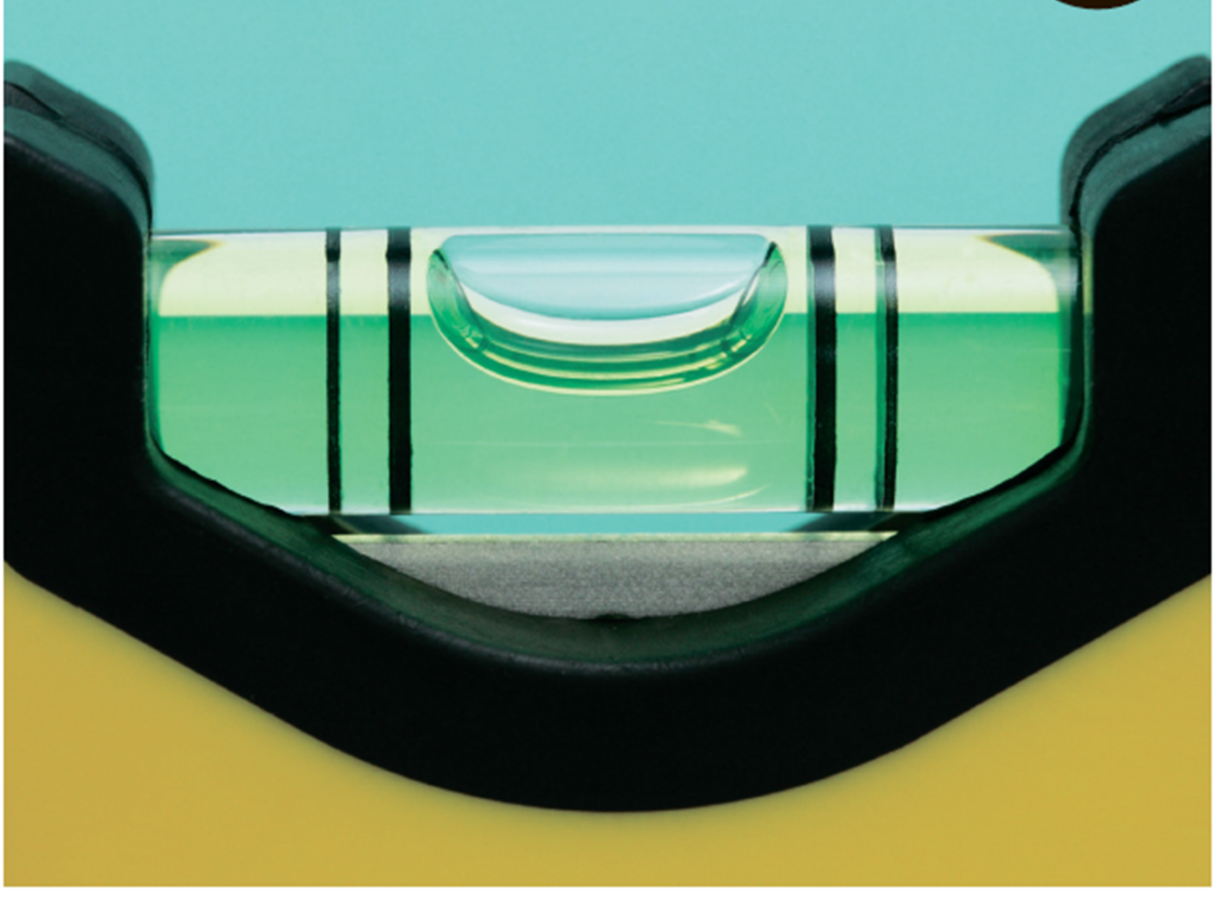
/ theory / in / practice

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

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

Software Testing

The Art of Application Performance Testing

SECOND EDITION

Ian Molyneaux

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

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

Preface

WHEN I PUBLISHED THE ORIGINAL EDITION OF THIS BOOK IN JANUARY 2009, I HAD no idea how popular it would prove to be within the performance testing community. I received many emails from far and wide thanking me for putting pen to paper, which was an unexpected and very pleasant surprise. This book was and still is primarily written for the benefit of those who would like to become performance testing specialists. It is also relevant for IT professionals who are already involved in performance testing, perhaps as part of a web-first company (large or small), especially if they are directly responsible for the performance of the systems they work with.

While I believe that the fundamentals discussed in the original edition still hold true, a lot has happened in the IT landscape since 2009 that has impacted the way software applications are deployed and tested. Take cloud computing, for example: very much a novelty in 2009, with very few established cloud vendors. In 2014 the cloud is pretty much the norm for web deployment, with on-the-fly environment spin-up and spin-down for dev, test, and production requirements. I have added cloud considerations to existing chapter content where appropriate.

Then consider the meteoric rise of the mobile device, which, as of 2014, is expected to be the largest source of consumer traffic on the Internet. I made passing mention of it in the original edition, which I have now expanded into a whole new chapter devoted to performance testing the mobile device. *End-user monitoring*, or EUM, has also come of age in the past five years. It has a clear overlap with performance testing, so I have

added two new chapters discussing how EUM data is an important component to understanding the real-world performance of your software application.

Pretty much all of the original chapters and appendixes have been revised and expanded with new material that I am confident will be of benefit to those involved with performance testing, be they novices or seasoned professionals. Businesses in today's world continue to live and die by the performance of mission-critical software applications. Sadly, applications are often still deployed without adequate testing for scalability and performance. To reiterate, *effective* performance testing identifies performance bottlenecks quickly and early so they can be rapidly triaged, allowing you to deploy with confidence. *The Art of Application Performance Testing, Second Edition*, addresses a continuing need in the marketplace for reference material on this subject. However, this is still *not* a book on how to tune technology X or optimize technology Y. I've intentionally stayed well away from specific tech stacks except where they have a significant impact on how you go about performance testing. My intention remains to provide a commonsense guide that focuses on planning, execution, and interpretation of results and is based on over 15 years of experience managing performance testing projects.

In the same vein, I won't touch on any particular industry performance testing methodology because—truth be told—they (still) don't exist. Application performance testing is a unique discipline and is (still) crying out for its own set of industry standards. I'm hopeful that the second edition of this book will continue to carry the flag for the appearance of formal process.

My career has moved on since 2009, and although I continue to work for a company that's passionate about performance, this book remains tool- and vendor-neutral. The processes and strategies described here can be used with any professional automated testing solution.

Hope you like the revised and updated edition!

—Ian Molyneaux, 2014

Audience

This book is intended as a primer for anyone interested in learning or updating their knowledge about application performance testing, be they seasoned software testers or complete novices.

I would argue that performance testing is very much an *art* in line with other software disciplines and should be not be undertaken without a consistent methodology and appropriate automation tooling. To become a seasoned performance tester takes many years of experience; however, the basic skills can be learned in a comparatively short time with appropriate instruction and guidance.

The book assumes that readers have some familiarity with software testing techniques, though not necessarily performance-related ones. As a further prerequisite, effective performance testing is really possible only with the use of automation. Therefore, to get the most from the book, you should have some experience or at least awareness of automated performance testing tools.

Some additional background reading that you may find useful includes the following:

- *Web Load Testing for Dummies* by Scott Barber with Colin Mason (Wiley)
- *.NET Performance Testing and Optimization* by Paul Glavich and Chris Farrell (Red Gate Books)
- *Web Performance Tuning* by Patrick Killilea (O'Reilly)
- *Web Performance Warrior* by Andy Still (O'Reilly)

About This Book

Based on a number of my jottings (that never made it to the white paper stage) and 10 years plus of hard experience, this book is designed to explain why it is so important to performance test any application before deploying it. The book leads you through the steps required to implement an effective application performance testing strategy.

Here are brief summaries of the book's chapters and appendixes:

Chapter 1, *Why Performance Test?*, discusses the rationale behind application performance testing and looks at performance testing in the IT industry from a historical perspective.

Chapter 2, *Choosing an Appropriate Performance Testing Tool*, discusses the importance of automation and of selecting the right performance testing tool.

Chapter 3, *The Fundamentals of Effective Application Performance Testing*, introduces the building blocks of effective performance testing and explains their importance.

Chapter 4, *The Process of Performance Testing*, suggests a best-practice approach. It builds on Chapter 3, applying its requirements to a model for application performance testing. The chapter also includes a number of case studies to help illustrate best-practice approaches.

Chapter 5, *Interpreting Results: Effective Root-Cause Analysis*, teaches effective root-cause analysis. It discusses the typical output of a performance test and how to interpret results.

Chapter 6, *Performance Testing and the Mobile Client*, discusses performance and the mobile device and the unique challenges of performance testing mobile clients.

Chapter 7, *End-User Experience Monitoring and Performance*, describes the complementary relationship between end-user experience monitoring and performance testing.

Chapter 8, *Integrating External Monitoring and Performance Testing*, explains how to integrate end-user experience monitoring and performance testing.

Chapter 9, *Application Technology and Its Impact on Performance Testing*, discusses the impact of particular software tech stacks on performance testing. Although the approach outlined in this book is generic, certain tech stacks have specific requirements in the way you go about performance testing.

Chapter 10, *Conclusion* is something I omitted from the original edition. I thought it would be good to end with a look at future trends for performance testing and understanding the end-user experience.

Appendix A, *Use-Case Definition Example*, shows how to prepare use cases for inclusion in a performance test.

Appendix B, *Proof of Concept and Performance Test Quick Reference*, reiterates the practical steps presented in the book.

Appendix C, *Performance and Testing Tool Vendors*, lists sources for the automation technologies required by performance testing and performance analysis. Although I have attempted to include the significant tool choices available at the time of writing, this list is not intended as an endorsement for any particular vendor or to be definitive.

Appendix D, *Sample Monitoring Templates: Infrastructure Key Performance Indicator Metrics*, provides some examples of the sort of key performance indicators you would use to monitor server and network performance as part of a typical performance test configuration.

Appendix E, *Sample Project Plan*, provides an example of a typical performance test plan based on Microsoft Project.

Conventions Used in This Book

The following typographical conventions will be used:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings and also within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

> **TIP**
>
> Signifies a tip or suggestion.

> **NOTE**
>
> Indicates a general note.

Glossary

The following terms are used in this book:

APM
: Application performance monitoring. Tooling that provides deep-dive analysis of application performance.

APMaaS
: APM as a Service (in the cloud).

Application landscape
: A generic term describing the server and network infrastructure required to deploy a software application.

AWS
: Amazon Web Services.

CDN
: Content delivery network. Typically a service that provides remote hosting of static and increasingly nonstatic website content to improve the end-user experience by storing such content local to a user's geolocation.

CI
: Continuous integration. The practice, in software engineering, of merging all developer working copies with a shared mainline several times a day. It was first named and proposed as part of extreme programming (XP). Its main aim is to prevent integration problems, referred to as *integration hell* in early descriptions of XP. (Definition courtesy of Wikipedia.)

DevOps
: A software development method that stresses communication, collaboration, and integration between software developers and information technology professionals. DevOps is a response to the interdependence of software development and IT operations. It aims to help an organization rapidly produce software products and services. (Definition courtesy of Wikipedia.)

EUM
: End-user monitoring. A generic term for discrete monitoring of end-user response time and behavior.

IaaS
: Infrastructure as a Service (in the cloud).

ICA
: Independent Computing Architecture. A Citrix proprietary protocol.

ITIL
: Information Technology Infrastructure Library.

ITPM
: Information technology portfolio management.

ITSM
: Information technology service management.

JMS
: Java Message Service (formerly Java Message Queue).

Load injector
: A PC or server used as part of an automated performance testing solution to simulate real end-user activity.

IBM/WebSphere MQ
: IBM's message-oriented middleware.

Pacing
: A delay added to control the execution rate, and by extension the throughput, of scripted use-case deployments within a performance test.

PaaS
: Platform as a Service (in the cloud).

POC
: Proof of concept. Describes a pilot project often included as part of the sales cycle. It enables customers to compare the proposed software solution to their current application and thereby employ a familiar frame of reference. Often used interchangeably with *proof of value*.

RUM
: Real-user monitoring. The passive form of end-user experience monitoring.

SaaS
: Software as a Service (in the cloud).

SOA
: Service-oriented architecture.

SUT
: System under test. The configured performance test environment.

Think time
: Similar to pacing, think time refers to pauses within scripted use cases representing human interaction with a software application. Used more to provide a realistic queueing model for requests than for throughput control.

Timing
: A component of a transaction. Typically a discrete user action you are interested in timing, such as *log in* or *add to bag*.

Transaction
: A typical piece of application functionality that has clearly defined start and end points, for example, the action of logging into an application or carrying out a search. Often used interchangeably with the term *use case*.

UEM
: User experience monitoring. A generic term for monitoring and trending end-user experience, usually of live application deployments.

Use case, user journey
: A set of end-user transactions that represent typical application activity. A typical transaction might be *log in*, *navigate to a search dialog*, *enter a search string*, *click the search button*, and *log out*. Transactions form the basis of automated performance testing.

WOSI
: Windows Operating System Instance. Basically the (hopefully licensed) copy of Windows running on your workstation or server.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require per-

mission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: *The Art of Application Performance Testing, Second Edition*, by Ian Molyneaux. Copyright 2015 Ian Molyneaux, 978-1-491-90054-3.

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/art-app-perf-testing*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Many thanks to everyone at O'Reilly who helped to make this book possible and put up with the fumbling efforts of a novice author. These include my editor, Andy Oram; assistant editor, Isabel Kunkle; managing editor, Marlowe Shaeffer; Robert Romano for the figures and artwork; Jacquelynn McIlvaine and Karen Tripp for setting up my blog and providing me with the materials to start writing; and Karen Tripp and Keith Fahlgren for setting up the DocBook repository and answering all my questions.

For the updated edition, I would also like to thank my assistant editor, Allyson MacDonald, and development editor, Brian Anderson, for their invaluable feedback and guidance.

In addition, I would like to thank my former employer and now partner, Compuware Corporation, for their kind permission to use screenshots from a number of their performance solutions to help illustrate points in this book. I would also like to thank the following specialists for their comments and assistance on a previous draft: Peter Cole, formerly president and CTO of Greenhat, for his help with understanding and expanding on the SOA performance testing model; Adam Brown of Quotium; Scott Barber, principal and founder of Association of Software Testers; David Collier-Brown, formerly of Sun Microsystems; Matt St. Onge; Paul Gerrard, principal of Gerrard Consulting; Francois MacDonald, formerly of Compuware's Professional Services division; and Alexandre Mechain, formerly of Compuware France and now with AppDynamics.

I would also like to thank my esteemed colleague Larry Haig for his invaluable insight and assistance with the new chapters on end-user monitoring and its alignment with performance testing.

Finally, I would like to thank the many software testers and consultants whom I have worked with over the last decade and a half. Without your help and feedback, this book would not have been written!

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.

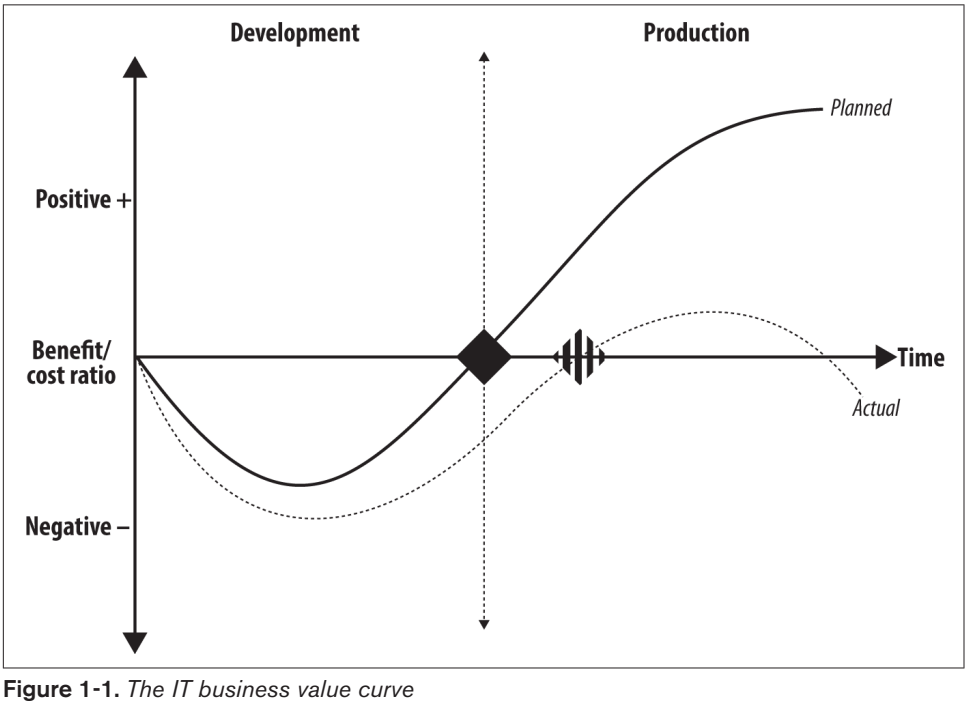


Figure 1-1. *The IT business value curve*

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!

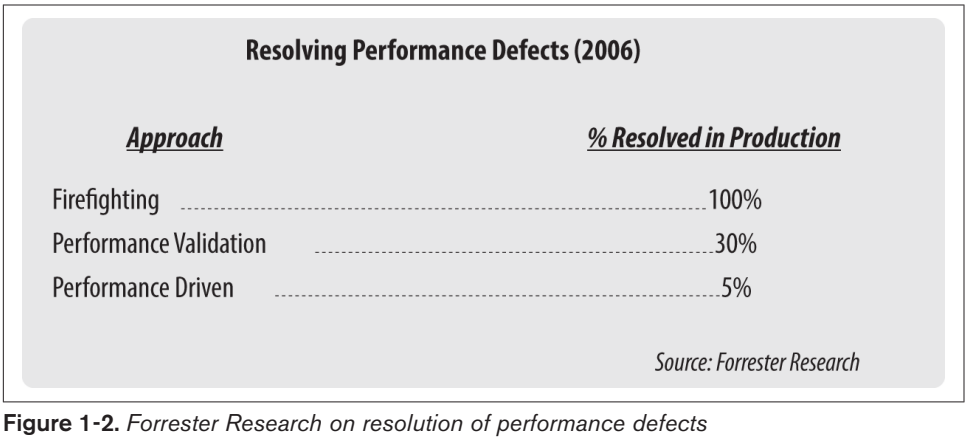


Figure 1-2. *Forrester Research on resolution of performance defects*

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.

CHAPTER TWO

Choosing an Appropriate Performance Testing Tool

> One only needs two tools in life: WD-40 to make things go, and duct tape to make them stop.
>
> —G. Weilacher

AUTOMATED TOOLS FOR PERFORMANCE TESTING HAVE BEEN AROUND IN SOME form for the best part of 20 years. During that period, application technology has gone through many changes, moving from a norm of fat client to web and, increasingly, mobile enablement. Accordingly, the sort of capabilities that automated tools must now provide is very much biased toward web and mobile development, and there is much less requirement to support legacy technologies that rely on a two-tier application model. This change in focus is good news for the performance tester, because there are now many more automated tool vendors in the marketplace to choose from with offerings to suit even modest budgets. There are also a number of popular open source tools available. (See Appendix C.)

All of this is well and good, but here's a note of caution: when your performance testing does occasionally need to move outside of the Web, the choice of tool vendors diminishes rapidly, and technology challenges that have plagued automated tools for many years are still very much in evidence. You are also much more unlikely to find an open source solution, so there will be a cost implication. These problems center not so much on execution and analysis, but rather on your being able to successfully record application activity and then modify the resulting scripts for use in a performance test. Terms such as *encryption* and *compression* are not good news for performance test tools; unless these technologies can be disabled, it is unlikely that you will be able to record scripts—meaning you will have to resort to manual coding or creating some form of test harness. Even web-based applications can present problems. For example, if you need to deal with streaming media or (from a security perspective) client certificates, then not all tooling vendors may be able to offer you a solution. You

should carefully match your needs to performance tool capabilities before making your final choice, and I strongly recommend you insist on a proof of concept (POC) before making any commitment to buy. Despite these challenges, you cannot carry out effective performance testing without tooling. In Chapter 1 I mentioned this as a reason why many applications are not properly performance tested prior to deployment. There is simply no practical way to provide reliable, repeatable performance tests without using some form of automation.

Accordingly, the aim of any automated performance testing tool is to simplify the testing process. It normally achieves this by enabling you to record end-user activity and render this data as scripts. The scripts are then used to create performance testing sessions or scenarios that represent a mix of typical end-user activity. These are the actual performance tests and, once created, they can easily be rerun on demand, which is a big advantage over any form of manual testing. Automation tooling also provides significant benefits to the analysis process, with the results of each test run automatically stored and readily available for comparison with any previous set of test results.

Performance Testing Tool Architecture

Automated performance test tools typically have the following components:

Scripting module
: Enables the recording of end-user activity and may support many different *middleware* protocols. Allows modification of the recorded scripts to associate internal/external data and configure granularity of response-time measurement. The term *middleware* refers to the primary protocol used by the application to communicate between the client and first server tier (for web applications this is principally HTTP or HTTPS).

Test management module
: Allows the creation and execution of performance test sessions or scenarios that represent different mixes of end-user activity. These sessions make use of nominated scripts and one or more load injectors depending on the volume of load required.

Load injector(s)
: Generates the load—normally from multiple workstations or servers, depending on the amount of load required. Each load injector generates multiple "virtual" users, simulating a large amount of end-user activity from a relatively small number of physical or virtual machines. The application client memory and CPU footprint can have a significant impact on the number of virtual users that can run within a given injector platform, affecting the number of injectors required.

Analysis module

Provides the ability to analyze the data collected from each test execution. This data is typically a mixture of autogenerated reports and configurable graphical or tabular presentation. There may also be an expert capability that provides automated analysis of results and highlights areas of concern.

Optional modules

Complements the aforementioned components to monitor server and network performance while a load test is running or allow integration with another vendor's software. Figure 2-1 demonstrates a typical automated performance test tool deployment. The application users have been replaced by a group of servers or workstations that will be used to inject application load by creating virtual users.

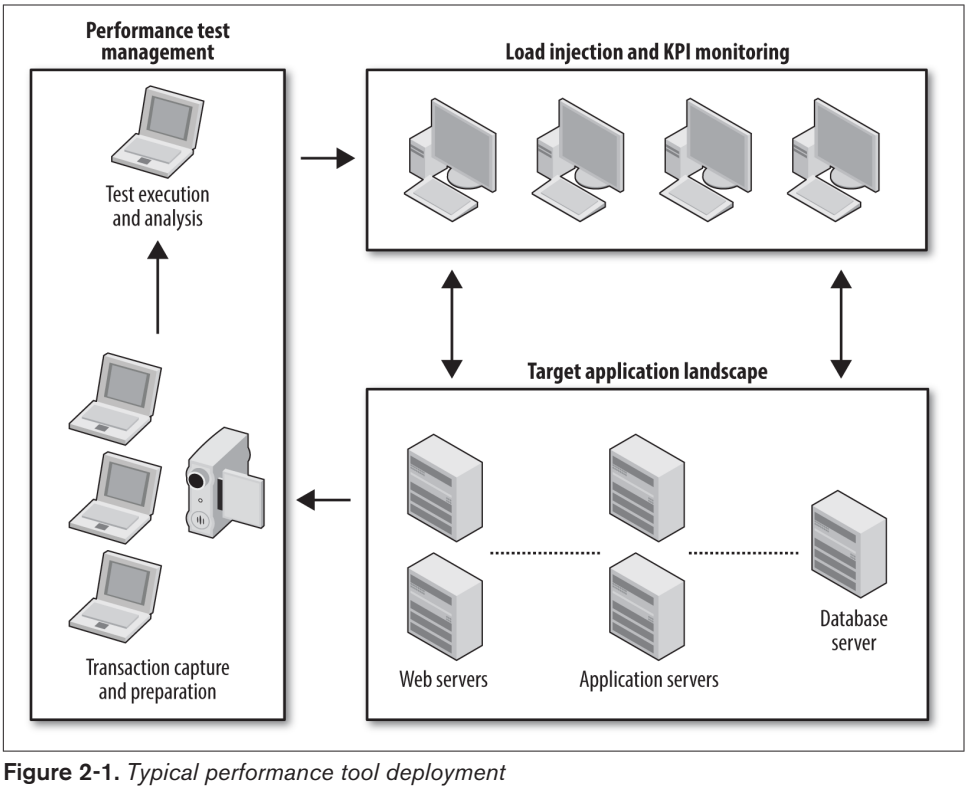


Figure 2-1. *Typical performance tool deployment*

Choosing a Performance Testing Tool

Many performance testing projects run into problems during the scripting stage due to insufficient technical evaluation of the tool being used. Most testing service providers keep a toolbox of solutions from a range of vendors, which allows them to choose the

most appropriate performance testing tool for a particular performance testing requirement.

With the current predominance of web technology, every serious tool vendor provides HTTP/S support. However, there are a lot of subtleties to web design, particularly at the client end, so if you make heavy use of JavaScript, JSON, or Microsoft Silverlight, for example, then be very certain you understand the capabilities and limitations of support in the tools that you shortlist. Here are some considerations to help you choose your tools wisely:

Protocol support
: The most important single requirement when you are choosing a performance testing tool is to ensure that it supports your application tech stack, specifically how the application client talks to the next application tier. For example, in most cases a typical browser client will be using HTTP or HTTPS as its primary communication protocol.

Licensing model
: Having satisfied yourself on the tech-stack support, look next at the licensing model. Most performance testing tool vendors offer a licensing model based on the following components:

  - The largest load test you can execute in terms of virtual users

    This value may be organized in breaks or bands, perhaps offering an entry level of up to 100 virtual users, with additional cost for higher numbers. You may also find tool vendors who offer an unlimited number of virtual users for a (higher) fixed cost. Some vendors can offer a term-license model whereby you can temporarily extend your license for one-off or occasional requirements to test much greater numbers of virtual users.

  - Additional protocols that the tool can support

    You may purchase the tool with support for the HTTP protocol, but a later requirement to support, for example, Citrix, would incur an additional cost. The additional protocol support may or may not be locked to the virtual users.

  - Additional plug-ins for integration and specific tech-stack monitoring

    As discussed, many tool vendors offer a range of optional modules you can add to your base license. These may include monitors for specific tech stacks like Oracle, MS SQL Server, and IIS, together with plug-ins to allow integration with application performance monitoring (APM) or continuous integration (CI) solutions. These may be extra cost (most likely) or free of charge; the important thing to bear in mind is that integration will likely require network access via specific ports or installation of agent software on target servers, so

you need to ensure that this can be accommodated by the system under test (SUT).

Make sure that you are clear in your own mind about the licensing model before making the decision to shortlist a tool.

Scripting effort

Many performance tool vendors claim that there is little or no need to make manual changes to the scripts that their tools generate. This may be true for simple browser use cases that navigate around a couple of pages, but the reality is that you *will* have to delve into code at some point during scripting (if only to insert JavaScript or code modules to deal with some complexity of application client design).

You may ultimately find that you'll need to make many manual changes to your recorded use case before it will replay successfully. If each set of changes requires several hours per script to implement with one tool but seems to be automatically handled by another, then you need to consider the potential cost savings and the impact of the extra work on your performance testing project and the testing team members.

Consider also the skill level of your testing team. If they come from a largely development background, then getting their hands dirty coding should not present too much of a problem. If, however, they are relatively unfamiliar with coding, you may be better off considering a tool that provides a lot of wizard-driven functionality to assist with script creation and preparation even if the upfront cost is greater.

Solution versus load testing tool

Some vendors offer only a load testing tool, whereas others offer a performance testing solution. Solution offerings will inevitably cost more but generally allow for a much greater degree of granularity in the analysis they provide. In addition to performance testing, they may include any or all of the following: automated requirements management, automated data creation and management, pre-performance test application tuning and optimization, response-time prediction and capacity modeling, APM providing analysis to class and method level, integration with end-user experience (EUE) monitoring after deployment, and dashboarding to provide visibility of test results and test assets.

In-house versus outsourced

If you have limited resources internally or are working within strict time constraints, consider outsourcing the performance testing project to an external vendor. Some tool vendors offer a complete range of services, from tool purchase with implementation assistance to a complete performance testing service, and there

are many companies that specialize in testing and can offer the same kind of service using whatever performance toolset is considered appropriate. This has the advantage of moving the commercials to time and materials and removes the burden of selecting the right testing tool for your application. The main disadvantage of this approach is that if you need to carry out frequent performance tests, the cost per test may rapidly exceed the cost of buying an automated performance testing solution outright.

The alternatives

If your application is web-based and customer-facing, you might want to consider one of a growing number of Software as a Service (SaaS) vendors who provide a completely external performance testing service. Essentially, they manage the whole process for you, scripting key use cases and delivering the load via multiple remote points of presence. This approach has the advantages of removing the burden of providing enough injection hardware and of being a realistic way of simulating user load, since these vendors typically connect directly to the Internet backbone via large-capacity pipes. This is an attractive alternative for one-off or occasional performance tests, particularly if high volumes of virtual users are required. These vendors often provide an EUE monitoring service that may also be of interest to you.

The downside is that you will have to pay for each test execution, and any server or network KPI monitoring will be your responsibility to implement and synchronize with the periods of performance test execution. This could become an expensive exercise if you find problems with your application that require many test reruns to isolate and correct.

Still, this alternative is well worth investigating. Appendix C includes a list of vendors who provide this service.

Performance Testing Toolset: Proof of Concept

You need to try the performance testing tools you shortlist against your application, so insist on a proof of concept so you can be certain that the toolset can deliver to your requirements. Identify a minimum of two use cases to record: one that can be considered read-only (e.g., a simple search activity that returns a result set of one or more entries), and one that inserts or makes updates to the application database. This will allow you to check that the recorded use case actually replays correctly. If your application is read-only, make sure to check the replay logs for each use case to ensure that there are no errors indicating replay failure.

The POC accomplishes the following:

Provides an opportunity for a technical evaluation of the performance testing tool against the target application

Obviously, technical compliance is a key goal; otherwise, you would struggle (or fail!) during the use-case scripting phase. You must try out the tool against the real application so that you can expose and deal with any problems before committing (or not) to the project.

Identifies scripting data requirements

The typical POC is built around a small number of the use cases that will form the basis of the performance testing project. It is effectively a dress rehearsal for the scripting phase that allows you to identify the input and the runtime data requirements for successful test execution. Because the POC tests only two or three use cases, you'll have additional data requirements for the remaining ones; these need to be identified later in the scripting phase of the project. However, the POC will frequently identify such common test input data requirements as login credentials and stateful information that keeps a user session valid.

Allows assessment of scripting effort

The POC allows you to estimate the amount of time it will take to script a typical use case and take into account the time to make alterations to scripts based on results of the POC.

Demonstrates capabilities of the performance testing solution versus the target application

It is always much more desirable to test a toolset against your application than against some sort of demo sandbox.

Proof of Concept Checklist

The following checklist provides a guide to carrying out an effective POC. Every POC will have its unique aspects in terms of schedule, but you should anticipate no more than a couple of days for completion—assuming that the environment and application are available from day one.

Prerequisites

Allocate these early in the project to make sure they're in place by the time you need to set up the POC environment:

- A set of success or exit criteria that you and the customer (or internal department) have agreed on as determining the success or failure of the POC. Have this signed off and in writing before starting the POC.
- Access to a standard build workstation or client platform that meets the minimum hardware and software specification for your performance testing tool or solution. This machine must have the application client *and* any supporting software installed.

- Permission to install any required monitoring software (such as server and network monitors) into the application landscape.
- Ideally, sole access to the application for the duration of the POC.
- Access to a person who is familiar with the application (i.e., a power user) and can answer your usability questions as they arise.
- Access to an expert who is familiar with the application (i.e., a developer) for times when you come up against technical difficulties or require an explanation of how the application architecture works at a middleware level.
- A user account that will allow correct installation of the performance testing software onto the standard build workstation and access to the application client.
- At least two sets of login credentials (if relevant) for the target application. The reason for having at least two is that you need to ensure that there are no problems with concurrent execution of the scripted use cases.
- Two sample use cases to use as a basis for the POC. One transaction should be a simple read-only operation, and the other should be a complex use case that updates the target data repository. These let you check that your script replay works correctly.

Process

This list helps you make sure the POC provides a firm basis for the later test:

- Record two instances of each sample use case and compare the differences between them using whatever method is most expedient. Windiff is one possibility, although there are better alternatives, or your performance testing tool may already provide this capability. Identifying what has changed between recordings of the same activity should highlight any runtime or session data requirements that need to be addressed.
- After identifying input and runtime data requirements as well as any necessary modifications to scripts, ensure that each scripted use case will replay correctly in single-user *and* multiuser mode. Make sure that any database updates occur as expected and that there are no errors in the relevant replay logs. In the POC (and the subsequent project), make certain that any modifications you have made to the scripts are free from memory leaks and other undesirable behavior.

NOTE

Windiff.exe is a Windows utility program that allows you to identify the differences between two files; it's free and has been part of Windows for some time. If you need greater functionality, try these tools:

- ExamDiff Pro from prestoSoft
- WinMerge

Deliverables

These are the results of the POC and the basis for approaching any subsequent project with confidence:

- The output of a POC should be a go/no-go assessment of the technical suitability of the performance testing tool for scripting and replaying application use cases.
- You should have identified the input and session data requirements for the sample use cases and gained insight into the likely data requirements for the performance testing project.
- You should identify any script modifications required to ensure accurate replay and assess the typical time required to script an application use case.
- If this is part of a sales cycle, you should have impressed the customer, met all the agreed-upon success criteria, and be well on the way to closing the sale!

Summary

This chapter has illustrated the importance of automation to performance testing and discussed the options available. The key takeaways are as follows:

- You simply cannot performance test effectively without automation.
- It is essential to select the most appropriate automation options for your requirements.

In the next chapter we move on to a discussion of the building blocks that are required to carry out effective application performance testing. These are commonly referred to as (non)functional requirements.

CHAPTER THREE

The Fundamentals of Effective Application Performance Testing

For the want of a nail . . .

—Anonymous

AFTER OUR BRIEF DISCUSSION OF THE DEFINITION OF GOOD AND BAD PERFOR-mance and the reasons for performance testing in Chapters 1 and 2, this chapter focuses on what is required to performance test effectively—that is, the prerequisites. The idea of a formal approach to performance testing is still considered novel by many, although the reason is something of a mystery, because (as with any kind of project) failing to plan properly will inevitably lead to misunderstandings and problems. Performance testing is no exception. If you don't plan your software development projects with performance testing in mind, then you expose yourself to a significant risk that your application will never perform to expectation. As a starting point with any new software development project, you should ask the following questions:

- How many end users will the application need to support at release? After 6 months, 12 months, 2 years?
- Where will these users be located, and how will they connect to the application?
- How many of these users will be concurrent at release? After 6 months, 12 months, 2 years?

These answers then lead to other questions, such as the following:

- How many and what specification of servers will I need for each application tier?
- Where should these servers be hosted?

- What sort of network infrastructure do I need to provide?

You may not be able to answer all of these questions definitively or immediately, but the point is that you've started the ball rolling by thinking early on about two vital topics, capacity and scalability, which form an integral part of design and its impact on application performance and availability. You have probably heard the terms *functional* and *nonfunctional* requirements. Broadly, functional requirements define what a system is supposed to do, and nonfunctional requirements (NFRs) define how a system is supposed to be (at least according to Wikipedia). In software testing terms, performance testing is a measure of the performance and capacity *quality* of a system against a set of benchmark criteria (i.e., what the system is *supposed to be*), and as such sits in the nonfunctional camp. Therefore, in my experience, to performance test effectively, the most important considerations include the following:

- Project planning
  - Making sure your application is stable enough for performance testing
  - Allocating enough time to performance test effectively
  - Obtaining a code freeze
- Essential NFRs
  - Designing an appropriate performance test environment
  - Setting realistic and appropriate performance targets
  - Identifying and scripting the business-critical use cases
  - Providing test data
  - Creating a load model
  - Ensuring accurate performance test design
  - Identifying the KPIs

NOTE

There are a number of possible mechanisms for gathering requirements, both functional and nonfunctional. For many companies, this step requires nothing more sophisticated than Microsoft Word. But serious requirements management, like serious performance testing, benefits enormously from automation. A number of vendors provide tools that allow you to manage requirements in an automated fashion; these scale from simple capture and organization to solutions with full-blown Unified Modeling Language (UML) compliance. You will find a list of tools from leading vendors in Appendix C.

Many of these (nonfunctional) requirements are obvious, but some are not. It's the requirements you overlook that will have the greatest impact on the success or failure of a performance testing project. Let's examine each of them in detail.

Making Sure Your Application Is Ready

Before considering any sort of performance testing, you need to ensure that your application is functionally stable. This may seem like stating the obvious, but all too often performance testing morphs into a frustrating bug-fixing exercise, with the time allocated to the project dwindling rapidly. Stability is confidence that an application does what it says on the box. If you want to create a purchase order, this promise should be successful every time, not 8 times out of 10. If there are significant problems with application functionality, then there is little point in proceeding with performance testing because these problems will likely mask any that are the result of load and stress. It goes almost without saying that code quality is paramount to good performance. You need to have an effective unit and functional test strategy in place.

I can recall being part of a project to test the performance of an insurance application for a customer in Dublin, Ireland. The customer was adamant that the application had passed unit/regression testing with flying colors and was ready to performance test. A quick check of the database revealed a stored procedure with an execution time approaching 60 minutes for a single iteration! This is an extreme example, but it serves to illustrate my point. There are tools available (see Appendix C) that help you to assess the suitability of your application to proceed with performance testing. The following are some common areas that may hide problems:

High data presentation
: Your application may be functionally stable but have a high network data presentation due to coding or design inefficiencies. If your application's intended users have limited bandwidth, then such behavior will have a negative impact on performance, particularly over the last mile. Excessive data may be due to large image files within a web page or large numbers of redundant conversations between client and server.

Poorly performing SQL
: If your application makes use of an SQL database, then there may be SQL calls or database stored procedures that are badly coded or configured. These need to be identified and corrected before you proceed with performance testing; otherwise, their negative effect on performance will only be magnified by increasing load (see Figure 3-1).

Large numbers of application network round-trips
: Another manifestation of poor application design is large numbers of conversations leading to excessive network chattiness between application tiers. High num-

bers of conversations make an application vulnerable to the effects of latency, bandwidth restriction, and network congestion. The result is performance problems in this sort of network condition.

Undetected application errors

Although the application may be working successfully from a functional perspective, there may be errors occurring that are not apparent to the users (or developers). These errors may be creating inefficiencies that affect scalability and performance. An example is an HTTP 404 error in response to a nonexistent or missing web page element. Several of these in a single transaction may not be a problem, but when multiplied by several thousand transactions per minute, the impact on performance could be significant.

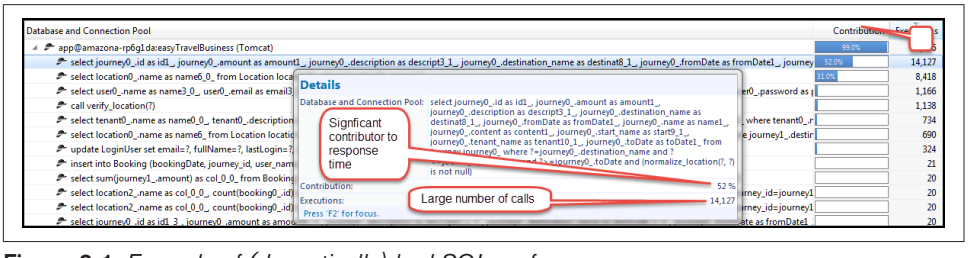


Figure 3-1. *Example of (dramatically) bad SQL performance*

Allocating Enough Time to Performance Test

It is extremely important to factor into your project plan enough time to performance test effectively. This cannot be a "finger in the air" decision and must take into account the following considerations:

Lead time to prepare test environment

If you already have a dedicated *performance* test environment, this requirement may be minimal. Alternatively, you may have to build the environment from scratch, with the associated time and costs. These include sourcing and configuring the relevant hardware as well as installing and configuring the application into this environment.

Lead time to provision sufficient load injectors

In addition to the test environment itself, consider also the time you'll need to prepare the resource required to inject the load. This typically involves a workstation or server to manage the performance testing and multiple workstations/servers to provide the load injection capability.

Time to identify and script use cases

It is vitally important to identify and script the use cases that will form the basis of your performance testing. Identifying the use cases may take from days to weeks,

but the actual scripting effort is typically 0.5 days per use case, assuming an experienced technician. The performance test scenarios are normally constructed and validated as part of this process. By way of explanation, a use case is a discrete piece of application functionality that is considered high volume or high impact. It can be as simple as navigating to a web application home page or as complex as completing an online mortgage application.

Time to identify and create enough test data
: Because test data is key to a successful performance testing project, you must allow enough time to prepare it. This is often a nontrivial task and may take many days or even weeks. You should also consider how long it may take to reset the target data repository or re-create test data between test executions, if necessary.

Time to instrument the test environment
: This covers the time to install and configure any monitoring of the application landscape to observe the behavior of the application, servers, database, and network under load. This may require a review of your current tooling investment and discussions with Ops to ensure that you have the performance visibility required.

Time to deal with any problems identified
: This can be a significant challenge, since you are dealing with unknowns. However, if sufficient attention has been given to performance during development of the application, the risk of significant performance problems during the testing phase is substantially reduced. That said, you still need to allocate time for resolving any issues that may crop up. This may involve the application developers and code changes, including liaison with third-party suppliers.

Obtaining a Code Freeze

There's little point in performance testing a moving target. It is absolutely vital to carry out performance testing against a consistent release of code. If you find problems during performance testing that require a code change, that's fine, but make sure the developers aren't moving the goalposts between—or, even worse, during—test cycles without good reason. If they do, make sure somebody tells the testing team! As mentioned previously, automated performance testing relies on scripted use cases that are a recording of real user activity. These scripts are normally version dependent; that is, they represent a series of requests and expected responses based on the state of the application *at the time* they were created. An unanticipated new release of code may partially or completely invalidate these scripts, requiring in the worst case that they be completely re-created. More subtle effects may be an apparently successful execution but an invalid set of performance test results, because code changes have rendered the scripts no longer an accurate representation of end-user activity. This is a common

problem that often highlights weaknesses in the control of building and deploying software releases and the communication between the Dev and QA teams.

Designing a Performance Test Environment

Next we need to consider the performance test environment. In an ideal world, it would be an exact copy of the production environment, but for a variety of reasons this is rarely the case.

The number and specification of servers (probably the most common reason)
: It is often impractical, for reasons of cost and complexity, to provide an exact replica of the server content and architecture in the test environment. Nonetheless, even if you cannot replicate the numbers of servers at each application tier, try to at least match the specification of the production servers. This will allow you to determine the capacity of an individual server and provide a baseline for modeling and extrapolation purposes.

Bandwidth and connectivity of network infrastructure
: In a similar vein, it is uncommon for the test servers to be deployed in the same location as their production counterparts, although it is often possible for them to share the same network infrastructure.

Tier deployment
: It is highly desirable to retain the production tier deployment model in your performance test environment unless there is absolutely no alternative. For example, if your production deployment includes a web application server and database tier, then make sure this arrangement is maintained in your performance test environment even if the number of servers at each tier cannot be replicated. Avoid the temptation to deploy multiple application tiers to a single physical platform.

Sizing of application databases
: The size and content of the test environment database should closely approximate the production one; otherwise, the difference will have a considerable impact on the validity of performance test results. Executing tests against a 1 GB database when the production deployment will be 1 TB is completely unrealistic.

Therefore, the typical performance test environment is a subset of the production environment. Satisfactory performance here suggests that things can only get better as we move to full-blown deployment. (Unfortunately, that's not always true!)

I have come across projects where all performance testing was carried out on the production environment. However, this is fairly unusual and adds its own set of additional considerations, such as the effect of other application traffic and the impact on real application users when a volume performance test is executed. Such testing is

usually scheduled out of normal working hours in order to minimize external effects on the test results and the impact of the testing on the production environment.

In short, you should strive to make the performance test environment as close a replica of production as possible within existing constraints. This requirement differs from functional testing, where the emphasis is on ensuring that the application works correctly. The misconception persists that a minimalist deployment will be suitable for both functional and performance testing. (Fail!) Performance testing needs a dedicated environment. *Just to reiterate the point: unless there is absolutely no way to avoid it, you should never try to use the same environment for functional and performance testing*.

For example, one important UK bank has a test lab set up to replicate its largest single branch. This environment comprises over 150 workstations, each configured to represent a single teller, with all the software that would be part of a standard desktop build. On top of this is deployed test automation software providing an accurate simulation environment for functional and performance testing projects.

As you can see, setting up a performance test environment is rarely a trivial task, and it may take many weeks or even months to achieve. Therefore, you need to allow for a realistic amount of time to complete this activity.

To summarize, there are three levels of preference when it comes to designing a performance test environment:

An exact or very close copy of the production environment
: This ideal is often difficult to achieve for practical and commercial reasons.

A subset of the production environment with fewer servers but specification and tier-deployment matches to that of the production environment
: This is frequently achievable—the important consideration being that from a bare-metal perspective, the specification of the servers at each tier should match that of the production environment. This allows you to accurately assess the capacity limits of individual servers, providing you with a model to extrapolate horizontal and vertical scaling.

A subset of the production environment with fewer servers of lower specification
: This is probably the most common situation; the performance test environment is sufficient to deploy the application, but the number, tier deployment, and specification of servers may differ significantly from the production environment.

Virtualization

A common factor influencing test environment design is the use of virtualization technology that allows multiple virtual server instances to exist on a single physical machine. VMWare remains the market leader despite challenges from other vendors (such as Microsoft) and open source offerings from Xen.

On a positive note, virtualization makes a closer simulation possible with regard to the number and specification of servers present in the production environment. It also simplifies the process of adding more RAM and CPU power to a given server deployment. If the production environment also makes use of virtualization, then so much the better, since a very close approximation will then be possible between test and production.

Possible negatives include the fact that you generally need more virtual than physical resources to represent a given bare-metal specification, particularly with regard to CPU performance where virtual servers deal more in virtual processing units rather than CPU cores. The following are some other things to bear in mind when comparing logical (virtual) to physical (real) servers:

Hypervisor layer
: Any virtualization technology requires a management layer. This is typically provided by what is termed the *hypervisor*, which acts as the interface between the physical and virtual world. The hypervisor allows you to manage and deploy your virtual machine instances but invariably introduces an overhead into the environment. You need to understand how this will impact your application behavior when performance testing.

Bus versus LAN-WAN
: Communication between virtual servers sharing the same physical bus will exhibit different characteristics than servers communicating over LAN or WAN. Although such virtual communication may use virtual *network interface cards* (NICs), it will not suffer (unless artificially introduced) typical network problems like bandwidth restriction and latency effects. In a data center environment there should be few network problems, so this should not be an issue. However, if your physical production servers connect across distance via LAN or WAN, then substituting common-bus virtual servers in test will not give you a true representation of server-to-server communication. In a similar vein, it is best (unless you are matching the production architecture) to avoid deploying virtual servers from different tiers on the same physical machine. In other words, don't mix and match: keep all web, application, and database virtual servers together but on different physical servers.

Physical versus virtual NICs
: Virtual servers tend to use virtual NICs. This means that, unless there is one physical NIC for each virtual server, multiple servers will have to share the same physical NIC. Current NIC technology is pretty robust, so it takes a lot to overload a card. However, channeling several servers' worth of network traffic down a single NIC increases the possibility of overload. I would advise minimizing the ratio of physical NICs to virtual servers.

Cloud Computing

The emergence of cloud computing has provided another avenue for designing performance test environments. You could argue that the cloud, at the end of the day, is nothing more than commoditized virtualized hosting; however, the low cost and relative ease of environment spin-up and teardown makes the cloud an attractive option for hosting many kinds of test environments.

Having gone from novelty to norm in a few short years, cloud computing has become one of the most disruptive influences on IT since the appearance of the Internet and the World Wide Web. Amazon has made the public cloud available to just about anyone who wants to set up her own website, and now other cloud vendors, too numerous to mention, offer public and private cloud services in all kinds of configurations and cost models. The initial concerns about data security have largely been dealt with, and the relatively low cost (provided you do your homework) and almost infinite horizontal and vertical flex (server spin-up/spin-down on demand) make the cloud a very compelling choice for application hosting, at least for your web tier. If you don't want the worry of managing individual servers, you can even go down the Platform as a Service (PaaS) route, principally offered by Microsoft Azure.

So is cloud computing a mini-revolution in IT? Most definitely, but how has the adoption of the cloud affected performance testing?

Load injection heaven

One of the biggest historical bug-bears for performance testing has been the need to supply large amounts of hardware to inject load. For smaller tests—say, up to a couple of thousand virtual users—this may not have been too much of a problem, but when you started looking at 10,000, 20,000, or 100,000 virtual users, the amount of hardware typically required often became impractical, particularly where the application tech stack imposed additional limits on the number of virtual users you could generate from a given physical platform. Certainly, hardware costs have come down dramatically over the last 10 years and you now get considerably more bang for your buck in terms of server performance; however, having to source, say, 30 servers of a given spec for an urgent performance testing requirement is still unlikely to be a trivial investment in time and money. Of course, you should be managing performance testing requirements from a strategic perspective, so you should never be in this situation in the first place. That said, the cloud may give you a short-term way out while you work on that tactical, last-minute performance mind-set.

On the surface, the cloud appears to solve the problem of load injector provisioning in a number of ways, but before embracing load injection heaven too closely, you need to consider a few important things.

On the plus side:

It's cheap.
: Certainly, the cost per cloud server—even of comparatively high spec—borders on the trivial, especially given that you need the servers to be active only while the performance test is executing. A hundred dollars can buy you a lot of computing power.

It's rapid (or relatively so).
: With a little planning you can spin up (and spin down) a very large number of injector instances in a comparatively short time. I say *comparatively* in that 1,000 injectors might take 30+ minutes to spin up but typically a much shorter period of time to spin down.

It's highly scalable.
: If you need more injection capacity, you can keep spinning up more instances pretty much on demand, although some cloud providers may prefer you give them a little advance notice. In the early days of cloud computing, my company innocently spun up 500 server instances for a large-scale performance test and managed to consume (at the time) something like 25 percent of the total server capacity available to Western Europe from our chosen cloud vendor. Needless to say, we made an impression.

It's tooling friendly.
: There are a lot of performance testing toolsets that now have built-in support for cloud-based injectors. Many will let you combine cloud-based and locally hosted injectors in the same performance test scenario so you have complete flexibility in injector deployment. A note of caution, though: level of automation to achieve this will vary from point-and-click to having to provide and manage your own load injector model instance and manually spin up/spin down as many copies as you need.

And now on the minus side:

It's not cheap.
: This is particularly true if you forget to spin down your injector farm after that 200,000-virtual-user performance test has finished. While cloud pricing models have become cheaper and more flexible, I would caution you about the cost of leaving 1,000 server instances running for 24 hours as opposed to the intended 3 hours. Think *$10,000* instead of *$100*.

It's not always reliable.
: Something I've noticed about cloud computing that has never entirely gone away is that you occasionally spin up server instances that, for some reason, don't work properly. It's usually pretty fundamental (e.g., you can't log in or connect to the

instance). This is not too much of a problem, as you can simply terminate the faulty instance and spin up another, but it is something you need to bear in mind.

Load Injection Capacity

As part of setting up the performance test environment, you need to make sure there are sufficient server resources to generate the required load. Automated performance test tools use one or more machines as load injectors to simulate real user activity. Depending on the application technology, there will be a limit on the number of virtual users that you can generate from a given machine.

During a performance test execution, you need to monitor the load being placed on the injector machines. It's important to ensure that none of the injectors are overloaded in terms of CPU or memory utilization, since this can introduce inaccuracy into your performance test results. Although tool vendors generally provide guidelines on injection requirements, there are many factors that can influence injection limits per machine. You should always carry out a dress rehearsal to determine how many virtual users can be generated from a single injector platform. This will give you a more accurate idea of how many additional injector machines are required to create a given virtual user load.

Automated performance testing will always be a compromise, simply because you are (normally) using a relatively small number of machines as load injectors to represent many application users. In my opinion it's a better strategy to use as many injector machines as possible to spread the load. For example, if you have 10 machines available but could generate the load with 4 machines, it's preferable to use all 10.

The number of load injectors in use can also affect how the application reacts to the Internet Protocol (IP) address of each incoming virtual user. This has relevance in the following situations:

Load balancing
: Some load balancing strategies use the IP address of the incoming user to determine the server to which the user should be connected for the current session. If you don't take this into consideration, then all users from a single injector will have the same IP address and may be allocated to the same server, which will not be an accurate test of load balancing and likely cause the SUT to fail. In these circumstances you may need to ask the client to modify its load balancing configuration or to implement *IP spoofing*, where multiple IP addresses are allocated to the NIC on each injector machine and the automated performance tool allocates a different IP address to each virtual user. Not all automated performance test tools provide this capability, so bear this in mind when making your selection.

User session limits
: Application design may enforce a single user session from one physical location. In these situations, performance testing will be difficult unless this limitation can be overcome. If you are limited to one virtual user per injector machine, then you will need a very large number of injectors!

Certain other situations will also affect how many injectors you will need to create application load:

The application technology may not be recordable at the middleware level
: Chapter 9 discusses the impact of technology on performance testing. In terms of load injection, if you cannot create middleware-level scripts for your application, you have a serious problem. Your options are limited to (a) making use of functional testing tools to provide load from the presentation layer; (b) making use of some form of thin-client deployment that *can* be captured with your performance testing tool—for example, Citrix ICA or MS Terminal Services RDP protocol; or (c) building some form of custom test harness to generate protocol-level traffic that can be recorded. If you can't take the thin-client route, then your injection capability will probably be limited to one virtual user per machine.

You need to measure performance from a presentation layer perspective
: Performance testing tools tend to work at the middleware layer, and so they have no real concept of activity local to the application client apart from periods of dead time on the wire. If you want to time, for example, how long it takes a user to click on a combo box and choose the third item, you may need to use presentation layer scripts and functional testing tools. Some tool vendors allow you to freely combine load and functional scripts in the same performance test, but this is not a universal capability. If you need this functionality, check to see that the vendors on your shortlist can provide it.

Addressing Different Network Deployment Models

From where will end users access the application? If everybody is on the local area network (LAN), your load injection can be entirely LAN based. However, if you have users across a wide area network (WAN), you need to take into account the prevailing network conditions they will experience. These primarily include the following:

Available bandwidth
: A typical LAN currently offers a minimum of 100 Mb, and many LANs now boast 1,000 or even 10,000 Mb of available bandwidth. WAN users, however, are not as fortunate and may have to make do with as little as 256 Kb. Low bandwidth and high data presentation do not generally make for good performance, so you must factor this into your performance testing network deployment model.

Network latency
: Think of this as delay. Most LANs have little or no latency, but the WAN user is often faced with high latency conditions that can significantly affect application performance.

Application design can have a major impact on how WAN friendly the application turns out to be. I have been involved in many projects where an application flew on the LAN but crawled on the WAN. Differing network deployment models will have a bearing on how you design your performance testing environment.

NOTE

You may be interested to know that inherent network latency is based on a number of factors, including these:

Speed of light
: Physics imposes an unavoidable overhead of 1 millisecond of delay per ~130 kilometers of distance.

Propagation delay
: Basically the impact of the wiring and network devices such as switches, routers, and servers between last mile and first mile.

There are tools available that allow you to model application performance in differing network environments. If significant numbers of your application users will be WAN-based, then I encourage you to consider using those. Typically they allow you to vary the bandwidth, latency, and congestion against a live recording of application traffic. This allows you to accurately re-create the sort of experience an end user would have at the end of a modest ADSL connection.

You may also wish to include WAN-based users as part of your performance test design. There are a number of ways to achieve this:

Load injection from a WAN location
: This is certainly the most realistic approach, although it is not always achievable in a test environment. You need to position load injector machines at the end of real WAN links and simulate the use-case mix and the number of users expected to use this form of connectivity as part of your performance test execution. (See Chapter 7 for further discussion on measuring performance from the perspective of the end user.)

Modify transaction replay
: Some performance testing tools allow you to simulate WAN playback even though the testing is carried out on a LAN environment. They achieve this by altering the replay characteristics of nominated use-case load-injector deployments to repre-

sent a reduction in available bandwidth—in other words, slowing down the rate of execution. In my experience, there is considerable variation in how tool vendors implement this feature, so be sure that what is provided will be accurate enough for your requirements.

Network simulation
: There are products available that allow you to simulate WAN conditions from a network perspective. Essentially, a device is inserted into your test network that can introduce a range of network latency effects, including bandwidth reduction.

Environment Checklist

The following checklist will help you determine how close your test environment will be to the production deployment. From the deployment model for the application, collect the following information where relevant for each server tier. This includes black-box devices such as load balancers and content servers if they are present.

Number of servers
: The number of physical or virtual servers for this tier.

Load balancing strategy
: The type of load balancing mechanism in use (if relevant).

Hardware inventory
: Number and type of CPUs, amount of RAM, number and type of NICs.

Software inventory
: Standard-build software inventory *excluding* components of application to be performance tested.

Application component inventory
: Description of application components to be deployed on this server tier.

Internal and external links
: Any links to internal or external third-party systems. These can be challenging to replicate in a test environment and are often completely ignored or replaced by some sort of stub or mock-up. Failing to take them into account is to ignore a potential source of performance bottlenecks. At the very minimum you should provide functionality that represents expected behavior. For example, if the external link is a web service request to a credit reference service that needs to provide subsecond response, then build this into your test environment. You will then have confidence that your application will perform well *as long* as external services are doing their part. (Make sure that any external link stub functionality you provide is robust enough to cope with the load that you create!)

Network connectivity is usually less challenging to replicate during testing, at least with regard to connections between servers. Remember that any load you apply should be present at the correct location in the infrastructure. For incoming Internet or intranet traffic, this is typically in front of any security or load balancing devices that may be present.

Software Installation Constraints

An important and often-overlooked step is to identify any constraints that may apply to the use of third-party software within the test environment. By *constraints*, I mean internal security policies that restrict the installation of software or remote access to servers and network infrastructure. These may limit the granularity of server and network monitoring that will be possible, and in the worst case may prevent the use of any remote monitoring capability built into the performance test tool you wish to use.

Although not normally a concern when you are performance testing in-house, such constraints are a real possibility when you're providing performance testing services to other organizations. This situation is more common than you might think and is not something you want to discover at the last minute! If this situation does arise unexpectedly, then you will be limited to whatever monitoring software is already installed in the performance test environment.

Setting Realistic Performance Targets

Now, what are your performance targets? These are often referred to as *performance goals* and may be derived in part from the service-level agreement (SLA). Unless you have some clearly defined performance targets in place against which you can compare the results of your performance testing, you could be wasting your time.

Consensus

It's crucial to get consensus on performance targets from all stakeholders. Every group on whom the application will have an impact—including the application users and senior management—must agree on the same performance targets. Otherwise, they may be reluctant to accept the results of the testing. This is equally true if you are performance testing in-house or providing a testing service to a third party.

Application performance testing should be an integral part of an internal strategy for application life cycle management. Performance testing has traditionally been an overlooked or last-minute activity, and this has worked against promoting consensus on what it delivers to the business.

Strategies for gaining consensus on performance testing projects within an organization should center on promoting a culture of consultation and involvement. You should get interested parties involved in the project at an early stage so that everyone

has a clear idea of the process and the deliverables. This includes the following groups or individuals:

The business

C-level management responsible for budget and policy decision–making:

- Chief information officer (CIO)
- Chief technology officer (CTO)
- Chief financial officer (CFO)
- Departmental heads

  Remember that you may have to build a business case to justify purchase of an automated performance testing solution and construction of an (expensive) test environment. So it's a good idea to involve the people who make business-level decisions and manage mission-critical business services (and sign the checks!).

IT

Stakeholders responsible for performance testing the application:

- The developers
- The testers (internal or outsourced)
- The infrastructure team(s)
- The service providers (internal and external)
- The end users

All these groups are intimately involved in the application, so they need to be clear on expectations and responsibility. The developers, whether in-house or external, put the application together; hence, there needs to be a clear path back to them should application-related problems occur during performance testing.

The testers are the people at the sharp end, so among other considerations they need to know the correct use cases to script, the type of test design required, and the performance targets they are aiming for.

Just as important are the people who look after the IT infrastructure. They need to be aware well in advance of what's coming to ensure that there is enough server and network capacity and that the application is correctly configured and deployed.

NOTE

If your application makes use of internal or external service providers, it is vital for them to be involved at an early stage. Make sure that any service SLAs contain specific content on expected performance and capacity.

Last but not least are the end users. End users will (or should have been) involved in the application design process and user acceptance testing (UAT). Assuming they have been using a legacy application that the application under test will replace, then they will have a clear and informed view of what will be considered acceptable performance.

Performance Target Definition

Moving on to specifics, I would argue that the following performance targets apply to any performance testing engagement. These are based on the service-oriented performance indicators that we have already discussed briefly in Chapter 1:

- Availability or uptime
- Concurrency
- Throughput
- Response time

To these can be added the following, which are as much a measure of capacity as of performance:

- Network utilization
- Server utilization

Availability or uptime

This requirement is simple enough: the application must be available to the end user at all times, except perhaps during periods of planned maintenance. It must not fail within the target level of concurrency or throughput. Actually, testing for availability is a matter of degree. A successful ping of the web server's physical machine doesn't necessarily mean that the application is available. Likewise, just because you can connect to the web server from a client browser doesn't mean that the application's home page is available. Finally, the application may be available at modest loads but may start to time out or return errors as the load increases, which (assuming no application problems) would suggest a lack of capacity for the load being generated.

Concurrency

Concurrency is probably the least understood performance target. Customers will often quote some number of concurrent users that the application must support without giving sufficient thought to what this actually entails.

In his excellent white paper "Get performance requirements right: Think like a user," Scott Barber suggests that calculating concurrency based on an hourly figure is more realistic and straightforward. Concurrency from the perspective of a performance testing tool is the number of active users generated by the software, which is not necessarily the same as the number of users concurrently accessing the application. Being able to provide an accurate level of concurrency depends very much on the design of your use-case scripts and performance test scenarios, discussed later in this chapter.

Scott also makes the point that from concurrency and *scalability* we derive capacity goals. In other words, achieving our scalability targets demonstrates sufficient capacity in the application landscape for the application to deliver to the business. Yet equally important is finding out just how much extra capacity is available.

In performance testing terms, concurrency refers to the following two distinct areas:

Concurrent virtual users
: The number of active virtual users *from the point of view of your performance testing tool.* This number is often very different from the number of virtual users actually accessing the system under test (SUT).

Concurrent application users
: The number of active virtual application users. By *active* I mean actually logged into or accessing the SUT. This is a key measure of the actual virtual user load against the SUT at any given moment during a performance test. It depends to a large degree on how you design your performance testing use cases (discussed later in the chapter). For now you need to decide if the login and logout process—whatever that involves—will be part of the application activity that is to be tested.

If you do include login-logout, then there will be a recurring point for every virtual user during test execution where he or she is logged out of the application and therefore not truly concurrent with other users. Achieving a certain number of concurrent virtual users is a balancing act between the time it takes for a use-case iteration to complete and the amount of pacing (or delay) you apply between use-case iterations. This is also called *use-case throughput.*

Performance Tool Versus Application Users Example

An example of this situation occurred in a recent project I was involved with. The customer wanted to test to 1,000 concurrent users. But because each user was logging in, completing an application activity, and then logging out, there were never more than a couple hundred users actively using the application at any point during the performance test. This was despite the testing tool indicating 1,000 active users.

The solution was (1) to increase the execution time or persistence of each use case iteration so that the user involved remained active for longer, and (2) to apply sufficient pacing between use-case executions to slow down the rate of execution. At the same time, it was important to ensure that the rate of throughput achieved was an accurate reflection of application behavior.

If you do not intend to include login-logout as part of use-case iteration, then concurrency becomes a whole lot easier to deal with, because every user who successfully logs in remains active (errors aside) for the duration of the performance test.

The concurrency target quoted either will be an extrapolation of data from an existing application or, in the case of a new application, will probably be based on a percentage of the expected total user community.

As with many things in business, the 80/20 rule often applies here: out of a total user community of 100, an average of 20 users will be using the application at any time during the working day. That said, every application will be different, and it's better to go for the high end of the scale rather than the low end. It is just as important that you look beyond deployment to determine what the concurrency will be 6 months, 12 months, or even 18 months later.

You must also include allowances for usage peaks outside of normal limits. Take, for example, an application that provides services to the students and staff of a large university. Day-to-day usage is relatively flat, but at certain times of the year—such as during student enrollment or when examination results are published online—concurrent usage increases significantly. Failure to allow for such usage peaks is a common oversight that may lead to unexpected performance degradation and even system failure.

Throughput

In certain circumstances *throughput* rather than concurrency is the performance testing target. This often occurs with applications that are considered stateless; that is to say, there is no concept of a logged-in session. Casual browsing of any website's home page would fall into this category. In these situations, the specific number of users who are simultaneously accessing the site is less important than the number of accesses or hits

in a given time frame. This kind of performance is typically measured in hits or page views per minute or per second.

In the world of ecommerce you can liken the usage profile of a website to a funnel in that the vast majority of activity will be casual browsing and "add to basket." When a customer decides to buy (or *convert*), he normally needs to log in to complete his purchase, thereby returning to a more traditional concurrency model. Interestingly, this is typically less than 5% of site activity (hence the *funnel* profile) even during peak times such as sales.

TIP

In the first edition of this book I suggested that a good rule of thumb was to add 10% to your anticipated "go live" concurrency or throughput target so you would at least be testing beyond predicted requirements. Building on this recommendation now, I would strongly advise including stress testing in your performance test scenarios so that you gain a real appreciation of your application's capacity limits and supporting infrastructure.

Response time

Upon reaching the target concurrency (or throughput), the application must allow the end users to carry out their tasks in a timely fashion.

As discussed in Chapter 1, from an end-user perspective good response time is very much a matter of perception; for example, an average response time of 60 seconds for a call-center application may be perfectly acceptable, whereas the same response for a critical stock-trading application would render it completely unusable. It could also be that the time taken to complete an activity such as a stock trade is more important than a visual indication of success appearing on the end user's PC.

This is often the challenge with migrating from a legacy green-screen application to a web-enabled replacement that is feature rich but cannot provide the same kind of instant response. Thoughtful application design—that is, making use of asynchronous functionality where appropriate—can go a long way toward convincing application users that the new version is a winner.

NOTE

In programming, asynchronous events are those occurring independently of the main program flow. Asynchronous actions are those executed in a nonblocking scheme, allowing the main program flow to continue processing. The effect on the application users is that they don't have to wait for something to complete before they can move on to a new activity. In contrast, synchronous events impose a strict sequence of events in that users cannot normally continue until the most recent request has completed or at least returned a success/fail response.

If there is an existing system that the application under test will replace, then there may be some performance statistics on current functionality to provide guidance on what is an acceptable response time for key application tasks. Absent this, your next resort is to use baselining. This is where a single user—running a single use case for a set period of time with no other activity occurring in the performance test environment—can provide a best possible response-time figure, assuming no significant application design problems.

Assuming this baseline value is acceptable, it will be the amount of variance from this figure at target concurrency (or throughput) that determines success or failure. For example, you may baseline the time it takes to raise a purchase order as 75 seconds. If this response time increases to an average of 500 seconds when 1,000 users are active, then this may be considered unacceptable. However, a smaller increase—say, 250 seconds—may be a perfectly acceptable result to maintain end-user productivity.

The bottom line is that there will almost always be some slowdown in response time as load increases, but this should not be in lockstep with the injection of additional load. More on this in Chapter 5.

Network Utilization

Every application presents data to the network. Just how much of an impact this has on performance will depend on the available bandwidth between application tiers and the end user. Within a modern data center, there is little chance of exhausting available bandwidth (thus, on-premise testing is not likely to turn up a network capacity-related performance problem here). However, as you move progressively closer to the user, performance can change dramatically—especially when communication involves the Internet.

High data presentation rates with large numbers of network conversations will typically have the strongest impact on the transmission path with the lowest bandwidth. So the data presented by our application may have to stay below a predetermined threshold, particularly for the last-mile connection to the Internet user.

In summary, the typical network metrics that you should measure when performance testing include the following:

Data volume
: The amount of data presented to the network. As discussed, this is particularly important when the application will have end users connecting over low-

bandwidth WAN links. High data volume, when combined with bandwidth restrictions and network latency effects, does not usually yield good performance.

The Impact of WAN on Application Performance

When looking to optimize WAN performance you are often working with limited bandwidth and higher-than-desirable latency conditions. It is therefore very important to minimize data presentation so you are consuming only the bandwidth you need and to make sure that your application can deal with periods of network slowdown and even occassional loss of connectivity. This is particularly important for mobile devices, where patchy network and cellular communications is pretty much the norm.

Data throughput
: The rate that data is presented to the network. You may have a certain number of page hits per second as a performance target. Monitoring data throughput will let you know if this rate is being achieved or if throughput is being throttled back. Often a sudden reduction in throughput is the first symptom of capacity problems, where the servers cannot keep up with the number of requests being made and virtual users start to suffer from server time-outs.

Data error rate
: Large numbers of network errors that require retransmission of data may slow down throughput and degrade application performance. Ideally, there should be zero network errors present; however, the reality is that there will almost always be some. It is important to determine the nature and severity of any errors that are detected and whether they are related to the application or to problems with the network infrastructure. Packet sniffing technology, as provided by the popular Wireshark toolset, can be extremely insightful when you are investigating network problems.

Server Utilization

In a similar fashion to the network, there may be fixed limits on the amount of server resources that an application is allowed to use. You can determine this information only by monitoring while the servers are under load. Obtaining this information relies on appropriately set server KPIs, which are discussed later in this chapter. There are many server KPIs that can be monitored, but in my experience the most common are as follows:

- CPU utilization
- Memory utilization

- Input/output—disk and network

These categories are frequently broken down by process or thread into the top 10 CPU, memory, and I/O hogs to provide a high-level view of the impact of load on server capacity and performance.

In an ideal world we would own the entire application landscape, but in real life the servers or network are often hosted by external vendors. This forces us to reassess the scope of infrastructure performance targets that can be realistically set.

You can probably test the data center comprehensively, but you often can't do so for the environment outside of the data center, where application traffic may pass through the hands of multiple service providers (Internet and SaaS) before reaching the end user. You have little real control over how the performance of your application will be affected by events in cyberspace, but you can try to ensure that there are no server and network performance bottlenecks within the data center.

> **NOTE**
>
> It is possible to enter into performance agreements with ISPs, but in my experience these are very difficult to enforce and will cover only a single ISP's area of responsibility.

In these situations, often the best you can aim for is to ensure your application is optimized to be as WAN-friendly as possible by minimizing the negative effects of bandwidth restriction, latency, and network congestion.

Identifying and Scripting the Business-Critical Use Cases

Use cases will form the basis of all your performance tests, so you must be sure that you have identified them correctly. A simple example of a use case is logging in to the application, navigating to the search page, executing a search, and then logging out again. You need to determine the high-volume, mission-critical activities that an average user will carry out during a normal working day.

Don't confuse performance testing use cases with their functional equivalents. Remember that your aim is not to test the application functionality (you should have already done that) but to create a realistic load to stress the application and then assess its behavior from a performance perspective. This should aim to reveal any problems that are a result of concurrency, lack of adequate capacity, or less-than-optimal configuration.

Use-Case Checklist

For each use case selected, you should complete the following actions:

Document each step so that there is no ambiguity in the scripting process
: Each step must be clearly defined with appropriate inputs and outputs. This is important not only for generating accurate scripts but also for defining the parts of each use case that will be timed during performance test execution. (See Appendix A for an example of a well-documented use case.)

Identify all input data requirements and expected responses
: This is the information that needs to be entered by the user at each step and is a critical part of defining the use case. Examples include logging in, entering a password, and—significantly—what should happen in response to these activities. Accurate documentation of input data will determine the test data requirements for each use case.

Determine user types
: Examples include a new or returning user for ecommerce applications, and supervisor, customer, call-center operator, and team leader for other application types. Deciding on the type of user will help you determine whether this is a high- or low-volume use case. For example, there are more call-center operators than supervisors, so supervisor users will likely limit themselves to background administration tasks, whereas the operators will be dealing with large numbers of customer inquiries.

Determine LAN or WAN
: What is the network distribution of users for each use case? They could be distributed via the LAN, the WAN, the Internet, or perhaps a combination. As discussed, a use case that performs well in a LAN environment may not do so when deployed in a low-bandwidth, high-latency WAN environment.

Decide whether the use case will be active or passive
: Active use cases represent the high-volume activities that you want to perform well, whereas passive use cases are background activities often included just to provide the test scenario with a realistic load. For example, there may be a requirement to include a proportion of users who are logged in but not actively using the application. The use-case definition for this sort of user may simply be logging in, doing nothing for a period of time, and then logging out—a passive use case.

TIP

Don't worry about this step, even if your application is considered complex and rich in features. In my experience, the number of use cases that are finally selected rarely exceeds 10, pretty much regardless of what the application does or how it is architectured. If you should find yourself with a larger-than-expected number of use cases, it may be necessary to carry out a risk assessment to determine those activities that must be tested in the time available for performance testing.

Use-Case Replay Validation

Once you have identified the use cases, you must convert them to some form of *script* using your performance testing tool. You typically achieve this by putting the testing tool into "record" mode and then carrying out the activity to be scripted. At the end of the recording process, the tool should automatically generate a script file representing your use-case activity. The actual format of the script will vary depending on the toolset used and may be a short program in C++ or C# or a more abstract document object model (DOM) presentation.

Once the script has been created, you will need to prep it by addressing any data requirements (discussed in the next section) and making any other changes required for the script to replay correctly. After you complete this task, your final step is to validate script replay. As part of the validation process, you will need to do the following:

Verify single user replay
: Always check that each script replays correctly using whatever validation functionality is provided by your performance testing tool. Obvious enough, you might think, but it is easy to wrongly assume that—because no obvious errors are present—your script has worked OK. Your performance testing tool should make it easy to verify successful replay by providing a trace of every client request and server response. I have seen many occasions where a performance testing project has been completed, only for the team to find that one or more scripted use cases were not replaying correctly, severely compromising the results.

Verify multiuser replay
: Once you are satisfied that the scripts are replaying correctly, you should always carry out a small multiuser replay to ensure that there are no problems that relate to concurrent execution. Testing for concurrency is, of course, a common performance target; however, as part of the validation process you should ensure that nothing related to script design prevents even a small number of virtual users from executing concurrently.

What to Measure

As just discussed, after identifying the key use cases, you need to record and script them using your performance tool of choice. As part of this process you must decide which parts of the use case to measure primarily in terms of response time.

You can indicate areas of interest while recording a use case simply by inserting comments, but most performance testing tools will allow you to ring-fence parts of a use case by inserting *begin* and *end* markers around individual or groups of requests, which I will refer to as *checkpoints*.

When the scripts are eventually used in a performance test, these checkpoints provide better response-time granularity than for the use case as a whole. For example, you may choose to checkpoint the login process or perhaps one or more search activities within the script. Simply replaying the whole use case without this sort of instrumentation will make it much harder to identify problem areas.

Checkpoints are really your first port of call when analyzing the results of a performance test, because they provide initial insight into any problems that may be present. For example, your total average response time for raising a purchase order may be 30 seconds, but analysis of your checkpoints shows that *Log into App Server* was taking 25 of the 30 seconds and therefore is the major contributor to use-case response time (see Figure 3-2).

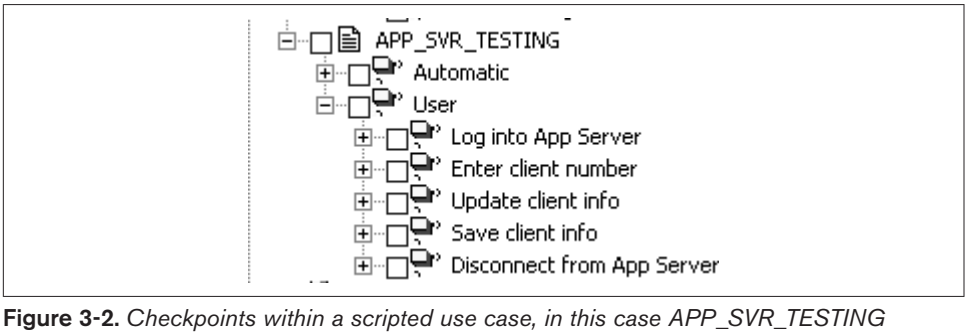


Figure 3-2. *Checkpoints within a scripted use case, in this case APP_SVR_TESTING*

To Log In or Not to Log In

Recall from our previous discussion of concurrency that the use case you script should reflect how the application will be used on a day-to-day basis. An important consideration here is the end-user *usage profile*. By this I mean whether users log in, complete an activity, and log out, or more typically log in once and remain logged in to the application during the working day.

Logging in to an application is often a load-intensive activity, so if users log in just a few times during a working day, it is unrealistic to include this step in every script iter-

ation. Most performance testing tools allow you to specify which parts of a scripted use case are repeated during test execution.

Remember that including or excluding the login-logout process in your scripts will have a significant impact on achieving your target *application* virtual user concurrency.

Peaceful Coexistence

Something else that is important to consider is whether your application will exist in isolation (very unlikely) or have to share resources with other applications. An application may perform magnificently on its own but fall down in a heap when it must coexist with others. Common examples include web traffic and email when used together or with other core company applications.

It may be that your application has its own dedicated servers but must still share network bandwidth. Simulating network effects may be as simple as factoring in enough additional traffic to approximate current bandwidth availability during testing. Creating an additional load where applications share mid-tier and database servers will be more complex, ideally requiring you to generate load for other applications to be included in performance testing. This means that you may need to identify and script use cases from other applications in order to provide a suitable level of background noise when executing performance tests.

Providing Test Data

OK, you have your performance targets and you have your scripted use cases; the next thing to consider is *data*. The importance of providing enough quality test data cannot be overstated. It would be true to say that performance testing lives and dies on the quality and quantity of the test data provided. It is a rare performance test that does not require any data to be provided as input to the scripted use cases.

Creating even a moderate amount of test data can be a nontrivial task. Most automated test tools by default take a file in comma-separated values (CSV) format as input to their scripts, so you can potentially use any program that can create a file in this format to make data creation less painful. Common examples are MS Excel and using SQL scripts to extract and manipulate data into a suitable format.

Three types of test data are critical: input data, target data, and session data.

Input Data

Input data is data that will be provided as input to your scripted use cases. You need to look at exactly what is required, how much of it you require, and—significantly—how much work is required to create it. If you have allocated two weeks for performance testing and it will take a month to produce enough test data, you need to think again!

Some typical examples of input data are as follows:

User credentials
: For applications that are designed around user sessions, this data would typically consist of a login ID and password. Many performance tests are executed with a limited number of test user credentials. This introduces an element of risk regarding multiple users with the same login credentials being active simultaneously, which in some circumstances can lead to misleading results and execution errors. Whenever possible, you should provide unique login credentials for every virtual user to be included in a performance test.

Search criteria
: There will almost always be use cases in any performance test that are designed to carry out various kinds of searches. In order to make these searches realistic, you should provide a variety of data that will form the search criteria. Typical examples include customer name and address details, invoice numbers, and product codes. You may also want to carry out wildcard searches where only a certain number of leading characters are provided as a search key. Wildcard searches typically take longer to execute and return more data to the client. If the application allows the end user to search for all customer surnames that begin with *A*, then you need to include this in your test data.

Associated files
: For certain types of performance tests, it may be necessary to associate files with use cases. This is a common requirement with document management systems, where the time to upload and download document content is a critical indicator of performance. These documents may be in a variety of formats (e.g., PDF, MS Word), so you need to ensure that sufficient numbers of the correct size and type are available.

Target Data

What about the target database? (It's rare not to have one.) This needs to be populated with realistic volumes of valid data so that your inquiries are asking the database engine to perform realistic searches. If your test database is 50 MB and the live database is 50 GB, this is a sure fire recipe for misleading results.

Let's look at the principal challenges of trying to create and manage a test database:

Sizing
: It is important to ensure that you have a realistically sized test database. Significantly smaller amounts of data than will be present at deployment provide the potential for misleading database response times, so consider this option only as a last resort. Often it's possible to use a snapshot of an existing production database, which has the added benefit of being real rather than test data. However, for new

applications this won't normally be possible, and the database will have to be populated to realistic levels in advance of any testing.

Data rollback
: If any of the performance tests that you run change the content of the data within the test database, then ideally—prior to each performance test execution—the database should be restored to the same state it was in before the start of the first performance test. It's all about minimizing the differences between test runs so that comparisons between sets of results can be carried out with confidence.

You need to have a mechanism in place to accomplish this data rollback in a realistic amount of time. If it takes two hours to restore the database, then this must be factored into the total time allowed for the performance testing project.

Session Data

During performance test execution, it is often necessary to intercept and make use of data returned from the application. This data is distinct from that entered by the user. A typical example is information related to the current user session that must be returned as part of every request made by the client. If this information is not provided or is incorrect, the server would return an error or disconnect the session. Most performance testing tools provide functionality to handle this kind of data.

In these situations, if you fail to deal correctly with session data, then it will usually be pretty obvious, because your scripts will fail to replay. However, there will be cases where a script appears to work but the replay will not be accurate; in this situation, your performance testing results will be suspect. *Always verify that your scripts are working correctly before using them in a performance test.*

Data Security

It's all very well getting your hands on a suitable test database, but you must also consider the confidentiality of the information it contains. You may need to anonymize details such as names, addresses, and bank account numbers in order to prevent personal security from being compromised by someone casually browsing through test data. Given the rampant climate of identity fraud, this is a common condition of use that must be addressed.

Ensuring Accurate Performance-Test Design

Accurate performance-test design relies on combining the requirements discussed so far into a coherent set of performance test scenarios that accurately reflect the concurrency and throughput defined by the original performance targets. The first step in this process is to understand the type of performance tests that are typically executed.

Principal Types of Performance Test

Once you've identified the key use cases and their data requirements, the next step is to create a number of different types of performance tests. The final choice will largely be determined by the nature of the application and how much time is available for performance testing. The following testing terms are generally well known in the industry, although there is often confusion over what they actually mean:

Pipe-clean test

The pipe-clean test is a preparatory task that serves to validate each performance test script in the performance test environment. The test is normally executed for a single use case as a single virtual user for a set period of time or for a set number of iterations. This execution should ideally be carried out without any other activity on the system to provide a best-case measurement. You can then use the metrics obtained as a baseline to determine the amount of performance degradation that occurs in response to increasing numbers of users and to determine the server and network footprint for each scripted use case. This test also provides important input to the transaction volume or load model, as discussed later in this chapter.

Volume test

This is the classic performance test, where the application is loaded up to the target concurrency but usually no further. The aim is to meet performance targets for availability, concurrency or throughput, and response time. Volume testing is the closest approximation of real application use, and it normally includes a simulation of the effects of user interaction with the application client. These include the delays and pauses experienced during data entry, as well as (human) responses to information returned from the application.

Stress test

This has quite a different aim from a volume test. A stress test attempts to cause the application or some part of the supporting infrastructure to fail. The purpose is to determine the capacity threshold of the SUT. Thus, a stress test continues until something breaks: no more users can log in, response time exceeds the value you defined as acceptable, or the application becomes unavailable. The rationale for stress testing is that if our target concurrency is 1,000 users, but the infrastructure fails at only 1,005 users, then this is worth knowing because it clearly demonstrates that there is very little extra capacity available. The results of stress testing provide a measure of capacity as much as performance. It's important to know your upper limits, particularly if future growth of application traffic is hard to predict. For example, the scenario just described would be disastrous for something like an airport air-traffic control system, where downtime is not an option.

Soak, or stability, test

The soak test is intended to identify problems that may appear only after an extended period of time. A classic example would be a slowly developing memory leak or some unforeseen limitation in the number of times that a use case can be executed. This sort of test cannot be carried out effectively unless appropriate infrastructure monitoring is in place. Problems of this sort will typically manifest themselves either as a gradual slowdown in response time or as a sudden loss of availability of the application. Correlation of data from the injected load and infrastructure at the point of failure or perceived slowdown is vital to ensure accurate diagnosis.

Smoke test

The definition of smoke testing is to focus only on what has changed. Therefore, a performance smoke test may involve only those use cases that have been affected by a code change.

NOTE

The term *smoke testing* originated in the hardware industry. The term derived from this practice: after a piece of hardware or a hardware component was changed or repaired, the equipment was simply powered up. If there was no smoke (or flames!), the component passed the test.

Isolation test

This variety of test is used to home in on an identified problem. It usually consists of repeated executions of specific use cases that have been identified as resulting in a performance issue.

I believe that you should always execute pipe-clean, volume, stress, and soak tests. The other test types are more dependent on the application and the amount of time available for testing—as is the requirement for isolation testing, which will largely be determined by what problems are discovered.

Having covered the basic kinds of performance test, let's now discuss how we distribute load across the use cases included in a performance test scenario.

The Load Model

The load model is key to accurate and relevant performance testing. It defines the distribution of load across the scripted use cases, and very importantly, the target levels of concurrency and throughput required for testing. The load model is initially created for a volume test and should reflect the performance targets agreed on for the SUT, after which it is varied as required for other performance test types. It is important to take into account the effects of think time and pacing (discussed in the next section), as these variables will have an important impact on throughput calculations.

Another very important but often overlooked factor is the impact that test data can have on load model design. Take, for example, a use case that represents searching for a product on an ecommerce site. If your test data to drive the script is all based on complete product names, then the search functionality tested, while consistent, is not necessarily representative of how customers will use the site. What happens, for example, if a customer searches for all black dresses instead of a specific item and this exposes a poorly configured search algorithm?

My advice is to always structure your load model to take this sort of scenario into account. While wildcard and partial searches are permitted, I tend to configure multiple use-case deployments that reflect different sizes of the (expected) result set based on the data supplied. So if we use the aforementioned example, my deployments would look as follows:

Small data model
: Test data comprises specific product names or IDs.

Medium data model
: Test data comprises partial product names.

Large data model
: Test data comprises wildcard or minimal product name content.

Using this approach makes throughput calculation a little more complex in that the medium and large data model use-case iterations will generally take longer to execute, but the advantage is that you cover yourself against missing problems with search, which is an important part of most applications' functionality.

Moving on, there are several types of load model that you can create.

The simple concurrency model

This simply states that the target concurrency is X and that each scripted use case will be allocated a %X of virtual users, the sum of which represents the target concurrency value. The configuration of load injection and throughput is important only in that it mimics as accurately as possible real use of the application. For example, all users may log in between 9 and 10 a.m. and typically stay logged in for the working day, or there are never more than 100 purchase orders processed in an hour.

Table 3-1 provides an example of 10 use cases distributed among a total application concurrency of 1,000 virtual users.

Table 3-1. *Virtual user use-case allocation*

Use case	Allocated virtual users
MiniStatement	416
CustomStatement	69
ViewStandingOrders	69
TextAlertCreateCancel	13
TransferBetweenOwnAccounts	145
BillPaymentCreate	225
BusMiniStatement	33
BusCustomStatement	4
BusTransferBetweenOwnAccounts	10
BusBillPaymentCreate	16

Decide on the number of virtual users to allocate to each use-case deployment per test. Most automated performance test tools allow you to deploy the same use case multiple times with different numbers of virtual users and different injection profiles. In fact, this would be an extremely good idea for an application that has significant numbers of users connecting over differing WAN environments. Each deployment could simulate a different WAN bandwidth, allowing you to observe the effects of bandwidth reduction on application performance under load.

You should apportion the number of virtual users per use case to represent the variety and behavior of activity that would occur during normal usage. Where possible, make use of any existing historical data to ensure that your virtual user allocation is as accurate as possible. For new applications, the allocation can be based only on expectation and thus may need refinement during test execution.

The throughput model

This type of load model is more complex to create, as it relies on reaching but not necessarily exceeding a specified level of throughput. This may be based on complete use-case iterations per time period or, as is often the case with ecommerce applications, a certain number of page views per minute or second.

For existing applications you may source this information from business intelligence services such as Google Analytics or WebTrends if implemented, where it is a straightforward process to determine volumetrics data for a given period. For new applications, this can once again be only an estimate, but it should be based on the most

accurate information available and the resulting load model agreed upon between application stakeholders.

As a starting point I would determine the iteration and throughput metrics from a single user execution for each scripted use case. However, you will almost certainly need to experiment to achieve the appropriate level of throughput, as there are too many variables to rely on simple extrapolation from a baseline.

Think Time

Think time represents the delays and pauses that any end user will introduce while interacting with a software application. Examples include pausing to speak to a customer during data entry and deciding how much to bid for an item on eBay. When recording a use case, you'd typically represent think time by inserting pause commands within the logic flow of the resulting script.

My advice is to always retain think time, ideally as recorded, with a small variation if this can be easily introduced into the scripted use case. For example, vary each instance of recorded think time by ±10%. Think time is, by default, excluded from the response-time calculations of most performance test toolsets. An important point to note is that any think time included in your scripts should be realistic and represent normal delays as part of interaction with the application. You should avoid any unrealistically long pauses within your scripts.

The important thing about think time is that it introduces an element of realism into the test execution, particularly with regard to the queueing of requests in any given script iteration.

Pacing

Pacing is the principal way to affect the execution of a performance test. Whereas think time influences the rate of execution per iteration, pacing affects overall throughput. This is an important difference that I'll take a moment to explain.

Consider this example: even though an online banking application has 6,000 customers, we know that the number of people who choose to view their statements is never more than 100 per hour even at the busiest time of day. Therefore, if our volume test results in 1,000 views per hour, it is not a realistic test of this activity.

We might want to increase the rate of execution as part of a stress test, but not for volume testing. As usual, our goal is to make the performance test as true to life as we can. By inserting a pacing value, we can reduce the number of iterations and, by definition, the throughput.

For example, if a typical *mini-statement* use case takes 60 seconds to execute, then applying a pacing value of 2 minutes will limit the maximum number per hour to 30

per virtual user. We accomplish this by making each virtual user wait 2 minutes (less the use-case execution time) before beginning the next iteration. In other words, the execution time per virtual user iteration cannot be *less* than 2 minutes.

Load injection profile

Next, decide how you will inject the application load. There are five types of injection profile that are typically used as part of a performance test configuration:

Big Bang
: This is where all virtual users start at the same time. Typically, these users will run concurrently but not in lockstep; that is, they will start executing at approximately the same time but will never be doing exactly the same thing at the same moment during test execution.

Ramp-up
: This profile begins with a set number of virtual users and then adds more users at specified time intervals until a target number is reached. This is the usual way of testing to see if an application can support a certain number of concurrent users.

Ramp-up (with step)
: In this variation of straight ramp-up, there is a final target concurrency or throughput, but the intention is to pause injection at set points during test execution. For example, the target concurrency may be 1,000 users, but we need to observe the steady-state response time at 250, 500, and 750 concurrent users; so, upon reaching each of these values, injection is paused for a period of time. Not all automated performance test tools provide this capability within a single performance test. But it is a simple matter to configure separate tests for each of the step values and mimic the same behavior.

Ramp up (with step), ramp down (with step)
: Building on the previous profile, you may want to ramp up to a certain number of concurrent users (with or without steps) and then gradually reduce the number of virtual users (with or without steps) back to zero, which may signal the end of the test. Again, not all automated performance test tools provide this capability within a single performance test.

Delayed start
: This may be combined with any of the preceding injection profiles; it simply involves delaying the start of injection for a period of time for a particular script deployment.

If you choose the ramp-up approach, aim to build up to your target concurrency for each script deployment in a realistic period of time. You should configure the injection rate for each use-case deployment to reach full load simultaneously. Remember that

the longer you take ramping up (and ramping down), the longer the duration of the performance test.

For Big Bang load injection, caution is advised because large numbers of virtual users starting together can create a fearsome load, particularly on web servers. This may lead to system failure before the performance test has even had a chance to start properly.

For example, many client/server applications have a user login period at the start of the working day that may encompass an hour or more. Replicating 1,000 users logging in at the same moment when this would actually be spread over an hour is unrealistic and increases the chance of system failure by creating an artificial period of extreme loading. This sort of injection profile choice is not recommended unless it is intended to form part of a stress test.

Deciding on performance test types

After validation and pipe-clean, I tend to structure my performance tests in the following format, although the order may differ based on your own requirements:

Volume test each use case
: Decide on the maximum number of concurrent users or throughput you want for each scripted use case; then run a separate test for each up to the limit you set. This will provide a good indication of whether there are any concurrency-related issues on a per-use-case basis. Depending on the number of concurrent users, you may want to use a straight ramp-up or the ramp-up-with-step injection profile. I tend to use the ramp-up-with-step approach, since this allows fine-grained analysis of problems that may occur at a particular level of concurrency or throughput.

Isolation-test individual use cases
: If problems do occur, then isolation testing is called for to triage the cause of the problem.

Volume test use-case groups
: Once you've tested in isolation, the next step is normally to combine all scripted use cases in a single load test. This then tests for problems with infrastructure capacity or possible conflicts between use cases such as database lock contention. Once again, you may decide to use the ramp-up or ramp-up-with-step injection profile.

Isolation test use-case groups
: As with individual use-case performance tests, any problems you find may require running isolation tests to confirm diagnoses and point to solutions.

Stress test use-case groups
: Repeat the individual use case and/or use-case group tests, but reduce the amount of pacing or increase the number of virtual users to create a higher throughput

than was achieved during volume testing. This allows you to ascertain the upper capacity limits of the hosting infrastructure and to determine how the application responds to extreme peaks of activity.

Soak test use-case groups
: Repeat the individual use-case and/or use-case group performance tests for an extended duration. This should reveal any problems that manifest themselves only after a certain amount of application activity has occurred (for example, memory leaks or connection pool limitations).

Nonperformance tests
: Finally, carry out any tests that are not specifically related to load or stress. These may include testing different methods of load balancing or perhaps testing failover behavior by disabling one or more servers in the SUT.

Load injection point of presence

As part of creating performance test scenarios, you next have to decide from where you will inject the load. This is an important consideration, because you need to ensure that you are not introducing artificially high data presentation into areas where this simply wouldn't happen in the production environment. For example, trying to inject the equivalent of 100 users' worth of traffic down a 512 Kb ADSL link that would normally support a single user is completely unrealistic—you will simply run out of bandwidth. Even if you were successful, the resulting performance data would bear no relationship to the real end-user experience (unless 100 users may at some point share this environment). Once again, the cloud offers great flexibility for remote deployment of load injectors.

If you consider how users connect to most applications, there is a clear analogy with a system of roads. The last connection to the end user is the equivalent of a country lane that slowly builds into main roads, highways, and finally freeways with progressively more and more traffic. The freeway connection is the Internet or corporate LAN pipe to your data center or hosting provider, so this is where the high-volume load needs to be injected.

In other words, you must match the load you create at any given point in your test infrastructure with the amount of load that would be present in a real environment. If you don't, you run the risk of bandwidth constraints limiting the potential throughput and number of virtual users. The result is a misleading set of performance data.

Putting it all together

After taking into consideration all the points discussed so far, you should by now have a clear idea of how your performance tests will look. Figure 3-3 provides an example of a comprehensive set of data taken from an actual performance test document. I'm

not saying that you will always need to provide this level of detail, but it's an excellent example to which we can aspire.

Ref No	Transaction	Duration	Transaction Rate as per baseline		Calculated Users as per basdline			Simulated Load (Load factors applied)			
			tpd	tph	Min VU	Dist (%)	VU	tph	VU	Ramp-up (hh:mm:ss)	Pacing (mm:ss.ms)
1.7	Movement Continuation	25.00	2000	363.64	2.53	18.43	37	1,181.82	93	0:00:17	04:41.8
11.1.10	Customer Account Enquiry - Personal	21.00	2000	363.64	2.12	15.48	31	1,181.82	78	0:00:20	03:56.1
7.2	Approval Process	15.00	1900	345.45	1.44	10.50	21	1,122.73	53	0:00:32	02:48.3
2.3	Enquire & Amend on a Partner Record	20.00	1000	181.82	1.01	7.37	15	590.91	38	0:00:43	03:48.5
11.1.10.1	Customer Account Enquiry - Group	21.00	1000	181.82	1.06	7.74	15	590.91	38	0:00:43	03:48.5
4.5	Add members to a group scheme	34.00	600	109.09	1.03	7.52	15	354.55	38	0:00:39	06:20.8
11.3.1	Process Receipts (2 payments)	54.00	600	109.09	1.64	11.94	24	354.55	60	0:00:20	10:09.2
5.1	Create & Issue a quote	77.00	400	72.73	1.56	11.35	23	236.36	58	0:00:16	14:35.8
1.8.1	Add interested Party Role	47.00	350	63.64	0.83	6.06	12	206.82	30	0:00:44	08:42.2
1.8.2	Remove Interested Party Role	28.00	350	63.64	0.49	3.61	7	206.82	18	0:01:31	05:04.6
Total			10200	1854.55	13.7	100.0	200	6,027.27	500		

Rampup Interval (hh:mm:ss)	00:30:00	Duration of business day (hrs)	5.5
Start up VU's / transaction	1	Transaction rate load factor (%)	30
Baseline VU's	200	VU load factor (%)	150

Legend

tpd = Transactions per Day
tph = Transactions per Hour
Min VU = Minimum Virtual Users
Dis (%) = Distribution Percentage
VU = Virtual Users

Figure 3-3. *Example performance test configuration*

Observe that there is a section in the figure for baseline and a separate section for simulated load. The application was already supporting 200 virtual users without issue, and data was available to determine the use-case iteration rate per day and per hour for the existing load. This is the information presented in the baseline section and used as the first target for the performance test.

The simulated load section specified the final target load for the performance test, which was 500 virtual users. An important part of achieving this increased load was ensuring that the use-case execution and throughput rates were a realistic extrapolation of current levels. Accordingly, the virtual user (distribution), ramp-up, and pacing values were calculated to achieve this aim.

Here is a line-by-line description of the figure:

- Ref No: Sections in the master document that involve the *use case*
- Use Case: Name of the application *use case*
- Duration: Amount of time in seconds that the *use case* takes to complete
- *Use-Case* Rate (per baseline)
  - tpd: *Use case* rate per day
  - tph: *Use case* rate per hour
- Calculated Users (per baseline)
  - Min VU: Minimum number of virtual users for this *use case*
  - Dist %: Distribution of this *use case* as a percentage of the total number of virtual users in this test
  - VU: Number of virtual users allocated to this *use case* based on the Dist %

- Simulated Load (load factors applied)
  - tph: Target *use-case* rate per hour
  - VU: Total number of virtual users allocated for this *use case*
  - Ramp-up (hh:mm:ss): Rate of virtual user injection for this *use case*
  - Pacing (hh:mm:ss): Rate of pacing for this *use case*

Identifying the KPIs

You need to identify the server and network key performance indicators (KPIs) that should be monitored for your application. This information is vital to achieve accurate root-cause analysis of any problems that may occur during performance test execution. Ideally, the monitoring is integrated with your automated performance test solution. However, a lack of integration is no excuse for failing to address this vital requirement.

Server KPIs

You are aiming to create a set of monitoring models or templates that can be applied to the servers in each tier of your SUT. Just what these models comprise will depend largely on the server operating system and the technology that was used to build the application.

Server performance is measured by monitoring software configured to observe the behavior of specific performance metrics or counters. This software may be included or integrated with your automated performance testing tool, or it may be an independent product.

Perhaps you are familiar with the Perfmon (Performance Monitor) tool that has been part of Windows for many years. If so, then you are aware of the literally hundreds of performance counters that could be monitored on any given Windows server. From this vast selection, there is a core of a dozen or so metrics that can reveal a lot about how any Windows server is performing.

In the Unix/Linux world there are long-standing utilities like monitor, top, vmstat, iostat, and SAR that provide the same sort of information. In a similar vein, mainframes have their own monitoring tools that can be employed as part of your performance test design.

It is important to approach server KPI monitoring in a logical fashion—ideally, using a number of layers. The top layer is what I call generic monitoring, which focuses on a small set of counters that will quickly tell you if any server (Windows Linux or Unix) is under stress. The next layer of monitoring should focus on specific technologies that

are part of the application tech stack as deployed to the web, application, and database servers. It is clearly impractical to provide lists of suggested metrics for each application technology. Hence, you should refer to the documentation provided by the appropriate technology vendors for guidance on what to monitor. To aid you in this process, many performance testing tools provide suggested templates of counters for popular application technologies.

TIP

The ideal approach is to build separate templates of performance metrics for each layer of monitoring. Once created, these templates can form part of a reusable resource for future performance tests. I provide sample templates in Appendix D.

So to summarize, depending on the application architecture, any or all of the following models or templates may be required.

Generic templates

This is a common set of metrics that will apply to every server in the same tier that has the same operating system. Its purpose is to provide first-level monitoring of the effects of load and stress. Typical metrics would include monitoring how busy the CPUs are and how much available memory is present. Appendix D contains examples of generic and other templates. If the application landscape is complex, you will likely have several versions.

In my experience, a generic template for monitoring a Windows OS server based on Windows Performance Monitor should include at a minimum the following counters, which cover the key areas of CPU, memory, and disk I/O, and provide some visibility of the network in terms of errors:

- Total processor utilization %
- Processor queue length
- Context switches/second
- Available memory in bytes
- Memory pages faults/second
- Memory cache faults/second
- Memory page reads/second
- Page file usage %
- Top 10 processes in terms of the previous counters
- Free disk space %

- Physical disk: average disk queue length
- Physical disk: % disk time
- Network interface: Packets Received errors
- Network interface: Packets Outbound errors

Web and application server tier

These templates focus on a particular web or application server technology, which may involve performance counters that differ from those provided by Microsoft's Performance Monitor tool. Instead, this model may refer to the use of monitoring technology to examine the performance of a particular application server such as Oracle WebLogic or IBM's WebSphere.

Other examples include the following:

- Apache
- IIS (Microsoft Internet Information Server)
- JBOSS

Database server tier

Enterprise SQL database technologies are provided by a number of familiar vendors. Most are reasonably similar in architecture and modus operandi, but differences abound from a monitoring perspective. As a result, each type of database will require its own unique template. Examples familiar to most include the following:

- Microsoft SQL Server
- Oracle
- IBM DB2
- MySQL
- Sybase
- Informix
- Some newer database technologies are now commonly part of application design. These include NoSQL databases such as MongoDB, Cassandra, and DynamoDB.

Mainframe tier

If there is a mainframe tier in your application deployment, you should include it in performance monitoring to provide true end-to-end coverage. Mainframe monitoring tends to focus on a small set of metrics based around memory and CPU utilization per job and logical partition (LPAR). Some vendors allow integration of mainframe performance data into their performance testing solution. Performance monitoring tools for mainframes tend to be fairly specialized. The most common are as follows:

- Strobe from Compuware Corporation
- Candle from IBM

Depending on how mainframe connectivity is integrated into the application architecture, application performance monitoring (APM) tooling can also provide insight into the responsiveness of I/O between the mainframe and other application tiers.

Hosting providers and KPI monitoring

With the emergence of cloud computing, there is an accompanying requirement to monitor the performance of hosted cloud platforms. The challenge is that the level of monitoring information provided by hosting providers pre- and post-cloud computing varies greatly in terms of historical storage, relevance, and granularity.

I am very much a fan of the self-service approach (assuming your hosting provider allows this) in that by configuring your own monitoring you can at least be sure of what is being monitored. The cloud in particular makes this relatively easy to put in place. That said, if there are monitoring services already available (such as Amazon CloudWatch), then by all means make use of them but bear in mind they are not necessarily free of charge.

Cloud also provides the opportunity to bake monitoring technology into machine image templates (or AMI, in Amazon-speak). We have a number of clients who do this with New Relic APM and it works very well. The only word of caution with this approach is that if you are regularly flexing up large numbers of virtual server instances, then you may be accidentally violating the license agreement with your software tool vendor.

Network KPIs

In performance testing, network monitoring focuses mainly on packet round-trip time, data presentation, and the detection of any errors that may occur as a result of high data volumes. As with server KPI monitoring, this capability can be built into an automated performance test tool or provided separately.

If you have followed the guidelines on where to inject the load and have optimized the data presentation of your application, then network issues should prove the least likely cause of problems during performance test execution.

For network monitoring the choice of KPIs is much simpler, as there are a small set of metrics you should always be monitoring. These include the following:

Network errors
: Any network errors are potentially bad news for performance. They can signal anything from physical problems with network devices to a simple traffic overload.

Latency
: We discussed latency a little earlier in this chapter. From a network perspective, this is any delay introduced by network conditions affecting application performance.

Bandwidth consumption
: How much of the available network capacity is your application consuming? It's very important to monitor the bytes in and out during performance testing to establish the application network footprint and whether too high a footprint is leading to performance problems or network errors.

For Windows and Unix/Linux operating systems, there are performance counters that monitor the amount of data being handled by each NIC card as well as the number of errors (both incoming and outgoing) detected during a performance test execution. These counters can be part of the suggested monitoring templates described previously.

To help better differentiate between server and network problems, some automated performance test tools separate server and network time for each element within a web page (Figure 3-4).

Name	Response Code	Total Requests	Average response time	Average server	Average network	Server / Network
https://OnlineBanking.uk/online/img/arrow.gif	200	2	2.0816	99.9881 %	0.0119 %	
https://OnlineBanking.uk/online/css/advIE.css	200	2	2.9855	31.8739 %	68.1261 %	
http://OnlineBanking.uk/js/loan.js	200	6	0.4469	99.9912 %	0.0088 %	
http://OnlineBanking.uk/img/warning.jpg	200	6	0.0219	99.863 %	0.137 %	
http://OnlineBanking.uk/img/logo.gif	200	6	0.0519	99.9554 %	0.0446 %	
http://OnlineBanking.uk/img/award.gif	200	6	0.1396	99.9756 %	0.0244 %	
http://OnlineBanking.uk/img/small.jpg	200	6	0.4033	99.9864 %	0.0136 %	

Figure 3-4. *Example server/network response-time breakdown*

Application Server KPIs

The final and very important layer involves the application server (if relevant) and shifts the focus away from counters to component- and method-level performance. Essentially, this is looking inside the application server technology to reveal its contribution to performance problems that may initially be revealed by generic server and network monitoring.

Previously you saw the term *application performance management*, or APM. This is a rapidly growing area of IT, and the use of APM tooling greatly enhances the speed and effectiveness of triaging application performance problems either in test or in production. Application components that are memory or CPU hogs can be very difficult to isolate without the sort of insight that APM brings to monitoring. I talk more about how to leverage the benefits of APM tooling in Chapter 5.

Specific application KPI areas to consider are calls from the application to services internal and external. For example, you may choose to outsource the search functionality of your ecommerce site to a third party via an API. You should always look to monitor such API calls, as they can easily become performance bottlenecks. If these metrics are not forthcoming from the service provider, then consider instrumenting your application to monitor the performance of the service or indeed any other software components crucial to performance.

Summary

In this chapter we have taken a look at the nonfunctional requirements that are the essential prerequisites of effective performance testing. In the next chapter we turn our attention to making use of these requirements in order to build an effective performance testing process.

CHAPTER FOUR

The Process of Performance Testing

I see plans within plans.

—Frank Herbert, *Dune*

AS DISCUSSED IN CHAPTER 1, MANY PERFORMANCE TESTING PROJECTS COME together as a last-minute exercise. I would happily wager that you have been involved in a project that falls into this category. In such cases, you are constrained by limited time to test and pressure from the business to deploy by a certain date, even though the application may have serious performance problems. This chapter describes a performance testing approach to follow so that any new projects you participate in are unlikely to suffer from the same pitfalls.

In Chapter 3, my intention was to cover nonfunctional requirements (NFRs) in a logical but informal way. This chapter is about using these requirements to build a test plan: a performance testing checklist divided into logical stages. We'll also look at how this plan can be applied to a couple of case studies based on real projects. Each case study will demonstrate different aspects of the performance test process and will provide some of the examples we'll cover in Chapter 5 to explore the interpretation of performance test results. Each case study features a review of how closely (or not) the performance project aligns with the NFRs discussed in Chapter 3 and the suggested approach provided in this chapter.

Activity Duration Guidelines

Before describing a suggested approach, I thought it would be helpful to provide some guidelines on how much time to allow for typical performance testing activities. Certain activities, like building and commissioning a suitable test environment, are too varied and complex to allow meaningful estimates, but for other key tasks I can offer the following guidance:

Scoping and NFR capture
: It's important to accurately determine the scope of a performance testing engagement and to allow enough time to capture all of the NFR requirements. Provided that you already have this information (and stakeholders), this activity rarely takes more than a couple of days.

Scripting performance test use cases
: Allow half a day per use case, assuming an experienced performance engineer is doing the work. The reality is that some use cases will take more time to script, and others less. But from my experience with many performance testing engagements, this is a realistic estimate. *Note that the correct tool choice can make a huge difference to success or failure of this task.*

Creating and validating performance test scenarios
: This is typically one to two days' work, provided that you have created an accurate load model. This should simply be a job of work, since you will have already defined the structure and content of each performance test as part of your requirements capture. Most of your time will likely be spent conducting dress rehearsals to ensure that the tests execute correctly.

Executing the performance test
: For all but the simplest of projects, allow a minimum of five days. The unknown in this process is how many reruns will be required to deal with any problems that surface during test execution. Make sure that you also factor in time for database restore between performance test executions. You can get a lot of performance testing done in five days, and if you don't actually need them all, then you don't *have* to charge the client.

Collecting data and uninstalling software
: Allow one day. If you are testing in-house, then it's unlikely that you'll need to uninstall software. If you are providing a service, then you may have to remove the performance testing software. Give yourself a day to do this and to collect the results of the tests and KPI monitoring.

Conducting the final analysis and reporting
: Typically this takes two to three days. Make sure you allow sufficient time to complete your analysis and to prepare a comprehensive report aligned to the performance targets for the engagement. *Skimping on this task is something that greatly reduces the value of many performance testing engagements.*

Performance Testing Approach

To set the scene, the following list demonstrates the major steps in a suggested performance testing approach. I will then expand on each step in detail. The approach

described is applicable to most performance testing projects, whether you are executing in-house or providing a service to a customer. By design, this approach does not enforce the use of any specific tools, so you can keep using your preferred project management and requirements capture solution even if it amounts to nothing more than MS Word.

- Step 1: Nonfunctional Requirements Capture
- Step 2: Performance Test Environment Build
- Step 3: Use-Case Scripting
- Step 4: Performance Test Scenario Build
- Step 5: Performance Test Execution and Analysis
- Step 6: Post-Test Analysis and Reporting

> **NOTE**
>
> The steps are repeated in Appendix B as a handy quick reference.

Step 1: Nonfunctional Requirements Capture

Your first task in any performance testing project should be to set the following in motion. Gather or elicit performance NFRs from all relevant stakeholders, as discussed in Chapter 3. You will need this information to create a successful project plan or statement of work (SOW).

> **NOTE**
>
> This is often termed a *scoping* exercise.

At the very minimum you should have the following NFRs agreed upon and signed off before undertaking anything else.

- Deadlines available to complete performance testing, including the scheduled deployment date.
- Internal or external resources to perform the tests. This decision will largely depend on time scales and in-house expertise (or lack thereof).
- Test environment design. Remember that the test environment should be as close an approximation of the live environment as you can achieve and will require longer to create than you estimate.
- A code freeze that applies to the test environment within each testing cycle.

- A test environment that will not be affected by other user activity. Nobody else should be using the test environment while performance test execution is taking place; otherwise, there is a danger that the test execution and results may be compromised.
- Identified performance targets. Most importantly, these must be *agreed to* by appropriate business stakeholders. See Chapter 3 for a discussion on how to achieve stakeholder consensus.
- Key use cases. These must be identified, documented, and ready to script. Remember how vital it is to have *correctly* identified the key use cases to script. Otherwise, your performance testing is in danger of becoming a wasted exercise.
- Parts of each use case (such as login or time spent on a search) that should be monitored separately. This will be used in Step 3 for checkpointing.
- The input, target, and session data requirements for the use cases that you select. This critical consideration ensures that the use cases you script run correctly and that the target database is realistically populated in terms of size and content. As discussed in Chapter 3, quality test data is critical to performance testing. Make sure that you can create enough test data of the correct type within the timeframes of your testing project. You may need to look at some form of automated data management, and don't forget to consider data security and confidentiality.
- A load model created for each application in scope for performance testing.
- Performance test scenarios identified in terms of the number, type, use-case content, and virtual user deployment. You should also have decided on the think time, pacing, and injection profile for each use-case deployment.
- Identified and documented application, server, and network KPIs. Remember that you must monitor the hosting infrastructure as comprehensively as possible to ensure that you have the necessary information available to identify and resolve any problems that may occur.
- Identified deliverables from the performance test in terms of a report on the test's outcome versus the agreed-upon performance targets. It's a good practice to produce a document template that can be used for this purpose.
- A defined procedure for submitting any performance defects discovered during testing cycles to the development or application vendor. This is an important consideration that is often overlooked. What happens if you find major application-related problems? You need to build contingency into your test plan to accommodate this possibility. There may also be the added complexity of involving offshore resources in the defect submission and resolution process. If your plan is to carry out the performance testing using in-house resources, then you will also need to have the following in place:

— Test team members and reporting structure. Do you have a dedicated and technically competent performance testing team? Often organizations hurriedly put together such teams by grabbing functional testers and anyone else unlucky enough to be available. To be fair, if you have only an intermittent requirement to performance test, it's hard to justify keeping a dedicated team on standby. Larger organizations, however, should consider moving toward establishing an internal center of testing excellence. Ensure at the very minimum that you have a project manager and enough testing personnel assigned to handle the scale of the project. Even large-scale performance testing projects rarely require more than a project manager and a couple of engineers. Figure 4-1 demonstrates a sample team structure and its relationship to the customer.

— The tools, resources, skills, and appropriate licenses for the performance test. Make sure the team has what it needs to test effectively.

— Adequate training for all team members in the tools to be used. A lack of testing tool experience is a major reason why many companies' introduction to performance testing is via an outsourced project.

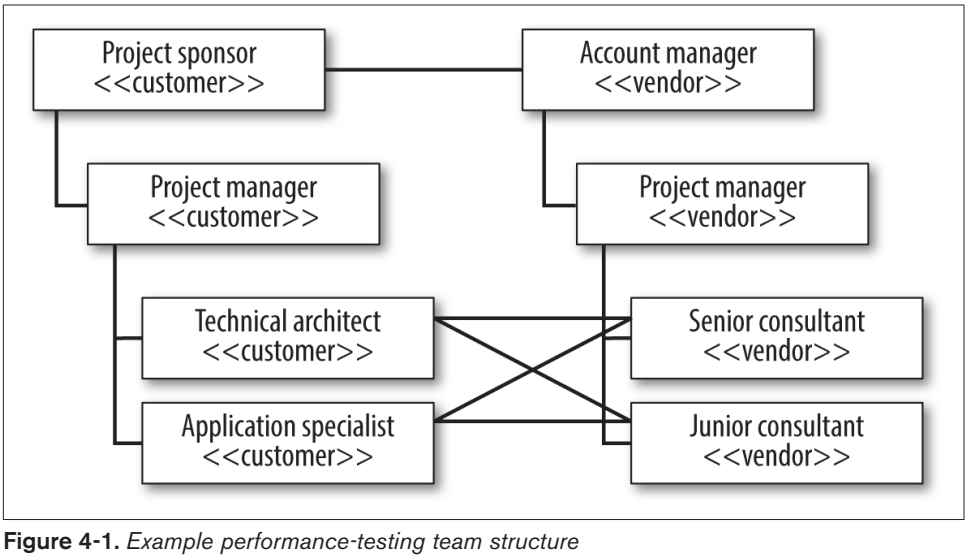


Figure 4-1. *Example performance-testing team structure*

With this information available, you can proceed with the following:

1. Develop a high-level plan that includes resources, timelines, and milestones based on these requirements.
2. Develop a detailed performance test plan that includes all dependencies and associated timelines, detailed scenarios and use cases, load models, and environment information.

3. Make sure that you include a risk assessment of not meeting schedule or performance targets just in case things don't go according to plan.

Be sure to include contingency for additional testing cycles and defect resolution if problems are found with the application during performance test execution. This precaution is frequently overlooked.

With these actions under way, you can proceed through the remaining steps. Not everything mentioned may be relevant to your particular testing requirements, but the order of events is important.

NOTE

See Appendix E for an example MS Project–based performance testing project plan.

Step 2: Performance Test Environment Build

You should have already identified the hardware, software, and network requirements for your performance test environment. Recall that you should strive to make your test environment a close approximation of the production environment. If this is not possible, then at a minimum your setup should reflect the server tier deployment of the production environment and your target database should be realistically populated in terms of both data content and sizing. This activity should be started as soon as possible because it frequently takes (much) longer than expected.

WARNING

You may have to build a business case to justify creating your performance test environment!

The performance test environment build involves the following steps:

1. Allow enough time to source equipment and to configure and build the environment.
2. Take into account all deployment models. You may need to test several different configurations over LAN and WAN environments for your application, so you need to ensure that each of these can be created in your test environment.
3. Take into account links to external systems. You should never ignore external links, since they are a prime location for performance bottlenecks. As discussed in Chapter 3, you need either to use real links or to create some kind of stub or simulation that represents external communication working as designed.

4. Provide enough load injection capacity for the scale of testing envisaged. Think about the locations from which you will need to inject load. If these are not all local to the application infrastructure, then you must ensure that the load injector machines can be managed remotely or station local personnel at each remote site. Don't forget about considering cloud-based load injectors if this simplifies your task.
5. Ensure that the application is correctly deployed into the test environment. I have been involved in more than one performance test engagement that was unfavorably affected by mistakes made when the application—supposedly ready for testing—was actually deployed.
6. Provide sufficient software licenses for the application and supporting software (e.g., Citrix or SAP licenses). You'd be surprised how often an oversight here causes problems.
7. Deploy and configure performance testing tools. Make sure that you *correctly* install and configure your performance testing solution.
8. Deploy and configure KPI monitoring. This may be part of your performance testing solution or entirely separate tooling. In either case, make sure that it is correctly configured to return the information you need.

Step 3: Use-Case Scripting

For each use case that you have selected to script, you must perform the following:

- Identify the *session* data requirements. Some of this information may be available from a proof of concept (POC) exercise, although this will have focused on only a few transactions. In many cases, you will not be able to confirm session data requirements for the remaining use cases until you begin the full scripting process.
- Confirm and apply *input* data requirements. These should have been identified as part of the pre-engagement requirements capture. See Appendix A for an example of the sort of detail you should provide, particularly with regard to data for each use case that will form part of your performance testing project.
- Decide on how you will *checkpoint* the use case in terms of what parts you need to monitor separately for response time. This is an important consideration because it provides first-level analysis of potential problem areas within the use case. You should have addressed this after identifying the key use cases during pre-engagement NFR capture.
- Identify and apply any scripting changes required for use cases to replay correctly. If you have already carried out a POC, then you should have a sense of the nature of these changes and the time required to implement them.

- Ensure that the use case replays correctly—from both a single-user and multiuser perspective—before signing it off as ready to include in a performance test. Make sure that you can verify what happens on replay either by, for example, checking that database updates have occurred correctly or examining replay logfiles.

Step 4: Performance Test Scenario Build

Making use of the load model created as part of NFR capture, consider the following points for each performance test scenario that you create:

- What kind of test will this represent—pipe-clean, volume, soak, or stress? A typical scenario is to run pipe-clean tests for each use case initially as a single user to establish a performance baseline, and then up to the target maximum currency or throughput for the use case. If problems are encountered at this stage, you may need to run isolation tests to identify and deal with what's wrong. This would be followed by a volume test combining all use cases up to target concurrency, which would in turn be followed by further isolation tests if problems are discovered. You may then want to run stress and soak tests for the final testing cycles, followed perhaps by non–performance-related tests that focus on optimizing the load balancing configuration or perhaps exploring different disaster recovery scenarios.
- Decide how you will represent think time and pacing for each use case included in the testing. You should normally include think time and pacing in all performance test types (except possibly the stress test); otherwise, you run the risk of creating an unrealistic throughput model.
- For each use case, decide how many load injector deployments you require and how many virtual users to assign to each point of presence. As already mentioned, if your load injectors are widely distributed, then be sure there's a local expert available to deal with any problems that may surface and don't forget the option of cloud-based load injectors.
- Decide on the injection profile for each load injector deployment Big Bang, ramp-up, ramp-up/ramp-down with step, or delayed start. Depending on what you are trying to achieve, your choices will likely involve a combination of Big Bang deployments to represent static load and one or more of the ramp variations to test for scalability. Figure 4-2 demonstrates a performance test plan using this approach. The darker rectangles on the bottom show the static load created by the Big Bang deployment, on top of which a variety of extra load tests are deployed in ramp-up steps.
- Will the test execute for a set period of time or rather be stopped by running out of data, reaching a certain number of use-case iterations, or user intervention? If the test will be data-driven, make sure that you have created enough test data!

- Do you need to spoof IP addresses to correctly exercise application load balancing requirements? If so, then the customer will need to provide a list of valid IP addresses, and these will have to be distributed and configured against the load injection machines.
- Do you need to simulate different baud rates? If so, then confirm the different baud rates required. Any response-time prediction or capacity modeling carried out prior to performance testing should have already given you valuable insight into how the application reacts to bandwidth restrictions.
- What runtime monitoring needs to be configured using the server and network KPIs that have already been set up? If appropriate, the actual monitoring software should have been deployed as part of the test environment build phase, and you should already have a clear idea of exactly what to monitor in the system under test (SUT).
- If this is a web-based performance test, what level of browser caching simulation do you need to provide—new user, active user, returning user? This will very much depend on the capabilities of your performance testing solution. See Chapter 9 for a discussion on caching simulation.
- Consider any effects that the application technology will have on your performance test design. For example, SAP performance tests that make use of the SAP-GUI client will have a higher resource requirement than, say, a simple terminal emulator and thus will require more load injector machines to generate a given number of virtual users. Chapter 9 discusses additional considerations for SAP and other application technologies.

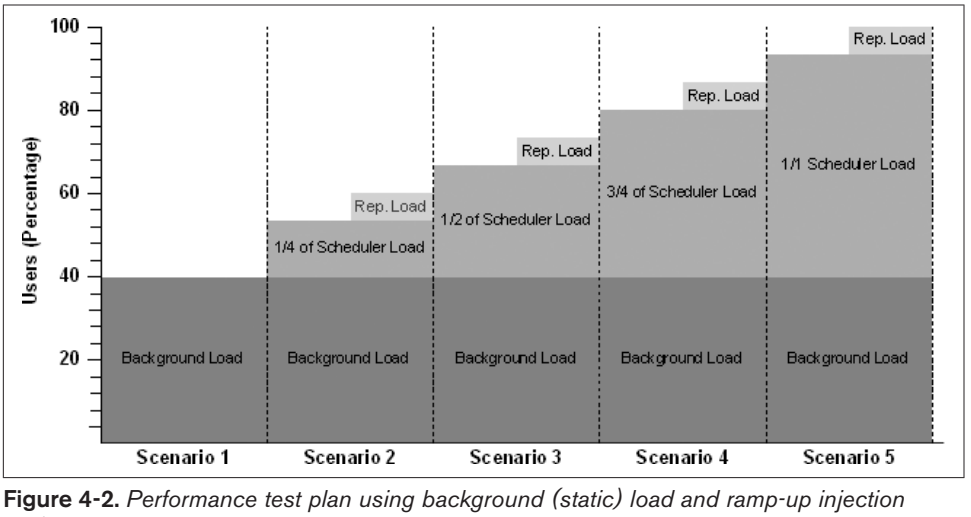


Figure 4-2. *Performance test plan using background (static) load and ramp-up injection profiles*

Step 5: Performance Test Execution

Run and monitor your tests. Make sure that you carry out a dress rehearsal of each performance test scenario as a final check that there are no problems accessing the application or with the test configuration.

This phase should be the most straightforward part of any performance testing project. You've done the hard work: preparing the test environment, creating the scripts, addressing the data requirements, and building the performance tests. In an ideal world, performance test execution would be solely a matter of validating your application performance targets. It should not become a bug-fixing exercise.

The only unknown is how many test cycles will be required before your performance testing goals are achieved. I wish I could answer this question for you, but like many things in life, this is in the hands of the gods. But if you've followed the suggested performance testing checklist religiously to this point, you're in pretty good shape to be granted a miracle!

Looking at the execution process, I recommend that you run tests in the following order:

1. Execute dress-rehearsal or pipe-clean tests as a final check on your performance test environment and test scenarios to ensure that you haven't omitted anything fundamental in your performance test configuration—for example, forgetting to include an external data file needed by a script.
2. Execute volume tests, ideally resetting target database content between test cycles. Once you have established performance baselines, the volume test is normally the next step: all scripted use cases should be apportioned among the target number of virtual users as per the application load model.
3. Execute isolation tests to explore any problems revealed by volume testing and then supply results to the developers or application vendor. This is why it's important to allow contingency time in your test plan, since even minor problems can have a significant impact on project time scales.
4. Execute stress tests, which are crucial from a capacity perspective. You should have a clear understanding of how much spare capacity remains within the hosting infrastructure at peak load to allow for future growth. You may also use stress testing to establish horizontal scalability limits for servers at a particular application tier.
5. Execute soak tests to reveal any memory leaks or problems related to high-volume executions. Although it is not always feasible, I strongly recommend you include soak testing in your performance testing plan.

6. Execute any tests that are not performance related. For example, experiment with different load balancing configurations.

Step 6: Post-Test Analysis and Reporting

The final step involves collecting and analyzing data from all test runs, creating reports and a performance baseline, and possibly retesting:

- Carry out final data collection (and possibly software uninstall if you are providing a service to a customer). Make sure that you capture and back up *all* data created as part of the performance testing project. It's easy to overlook important metrics and then discover they're missing as you prepare the project report.
- Determine success or failure by comparing test results to performance targets set as part of project requirements. Consensus on performance targets and deliverables *prior* to the project's commencement makes the task of presenting results and proving application compliance a great deal less painful. (See Chapter 3 for a discussion on gaining consensus.)
- Document the results using your preferred reporting template. The format of the report will be based on your own preferences and company or customer requirements, but it should include sections for each of the performance targets defined during NFR capture. This makes it much easier to present and *justify* the findings to the client.

Case Study 1: Online Banking

Now let's move on to the first case study, which I will refer to as Online Banking.

Online Banking is a critical customer-facing web application for a large multinational bank. It's been around for some time and provides a good service to the bank's customers. The motivation for the performance testing project that we'll discuss was an imminent expansion in terms of customer numbers and a need to explore the capacity limits of the current application infrastructure. The aim was to establish a baseline model for horizontal scaling of servers at the web, application server, and database tiers.

Application Landscape

Figure 4-3 shows the application landscape for Online Banking, which includes the following elements:

Clients
: This application provides a service to customers of the bank who connect using the Internet. This means that customers use whatever Internet browser software they prefer. Microsoft's Internet Explorer is still the most common choice, but there are many others, including Mozilla, Firefox, Opera, and Netscape. There is no requirement for the end user to have any other software installed apart from her preferred browser.

Mid-tier servers
: Because the application is web-based, the first tier is web servers. There are two quad-blade servers, each running Windows 2003 Server as the operating system and Microsoft's IIS 6 as the web server software. The servers are load-balanced through Microsoft's Network Load Balancing (NLB) technology. Behind the two web servers is a mid-tier layer of a single dual-CPU application server.

Database servers
: This is a single high-specification machine running MS SQL 2005 database software on Windows 2003 Server.

Network infrastructure
: All servers reside in a single data center with gigabit Ethernet connectivity. Internet connectivity to the data center consists of a single 8 Mb WAN link to the local Internet backbone. There is also a 100 Mb LAN connection to the company network.

Application Users

The application currently supports a maximum of approximately 380 concurrent users at the busiest point during a 24-hour period. These are mainly Internet customers, although there are likely to be a number of internal users carrying out administration tasks. Every user has a unique set of login credentials and a challenge/response security question. The services provided by the application are typical for online banking:

- Viewing statements
- Making payments
- Setting up direct deposits and standing orders
- Applying for a personal loan

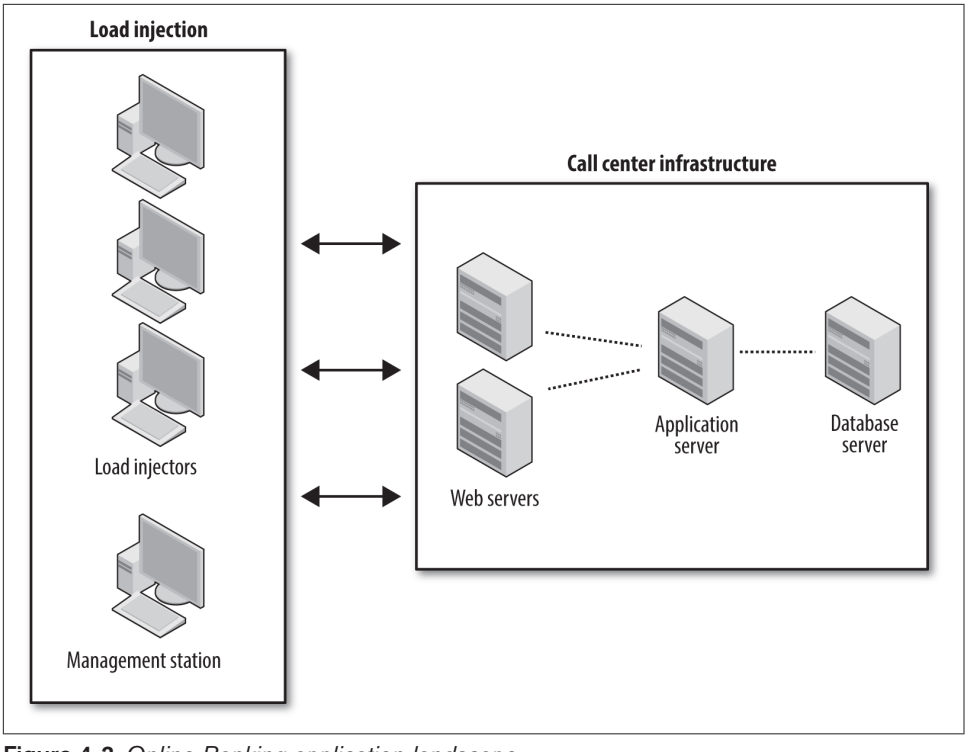


Figure 4-3. *Online Banking application landscape*

Step 1: Pre-Engagement NFR Capture

Project time scales amounted to a week of lead time and three days to carry out the testing engagement. The client decided to outsource the entire testing project because of the short schedule and lack of in-house testing expertise.

The test environment required little preparation, since testing was to be carried out using the production infrastructure. (More on this approach in Step 2.)

Performance targets for online banking were limited to availability and concurrency. The application had to be available and performant at a concurrent load of 1,000 virtual users. As mentioned, the application was coping with approximately 380 concurrent users at peak periods on a daily basis without problems, but this number was expected to increase significantly during the next year.

Ten use cases were identified as core to the performance testing project. Input data requirements amounted to a list of card numbers representing real user accounts and an accompanying PIN that provided an additional level of account security. Target data requirements were unusually straightforward in that the production application data-

base was to be used with the caveat of no insert or update use cases unless the use case backed up any changes made to the database as its final action.

The only performance test identified was a progressive ramp-up (without step) to the target concurrency of 1,000 virtual users. Had more time been available, a ramp-up with step variation would have provided a more granular view of steady-state performance.

Server and network KPIs focused on generic Windows performance metrics. There was no requirement to specifically monitor the application server or database layer. It subsequently transpired that application server monitoring would have been useful to identify poorly performing Java components whose presence had to be extrapolated from slow rendering of client-side content.

Step 2: Test Environment Build

What was unusual about this project was that there was no dedicated test environment. All performance testing was carried out against the production infrastructure. This is certainly not the norm, but neither is it a unique situation.

This case study provided some unique challenges (although an accurate test environment was not one of them). The first was dealing with an application already deployed and being actively used by customers and internal users. Any testing could be affected by other user activity, so we had to take this into account when examining test results. In addition, performance test execution would definitely have an impact on the experience of real end users, particularly if high volumes of load are being generated.

So in this rather unusual situation, the performance test environment was immediately available and consisted of the following:

- Two load-balanced web servers
- One application server
- One database server

The one element missing was the resource needed to inject load. This resource was identified as four PCs as injectors to provide a load of 1,000 virtual users (250 users per machine), with one PC also acting as the test management station. Another consideration was that the volume load had to be injected onto the 8 Mb WAN link that provided the connection from the Internet to the corporate data center. This involved distributing the load across a number of proxy servers to minimize the chance of overloading any single point of entry.

Because performance testing was carried out on the live environment, it was not possible to use integrated server and network monitoring capabilities of the chosen perfor-

mance test tool. Corporate security constraints prevented the installation onto production servers of any software that was not part of the standard build configuration. (The infrastructure was, after all, for a live banking application, so the restrictions were hardly surprising or draconian.)

This was an example of installation constraints. Internal security policies can prohibit the installation of monitoring software onto the target servers or even connecting to them remotely. In these circumstances, the only options are to dispense with infrastructure monitoring altogether (not recommended) or to rely on whatever independent monitoring software has access to the application landscape.

For Online Banking it was possible to make use of Microsoft's Performance Monitor (Perfmon) application to view server performance under load. This software is normally part of a default install on Windows server operating systems. A generic set of Windows server KPIs was instrumented with Perfmon and set to run in parallel with each performance test execution.

Step 3: Use-Case Scripting

The 10 use cases that were identified as key activities are listed in Table 4-1. Because this was a vanilla browser-based application, there were few challenges recording and preparing the scripts. The only complication concerned session data requirements that involved adding logic to the scripts to deal with the random selection of three characters from the account PIN at application login.

Table 4-1. *Online Banking use cases*

Use case	Description
MiniStatement	Log in, view a mini statement, log out
CustomStatement	Log in, view a custom statement, log out
ViewStandingOrders	Log in, view standing orders, log out
TextAlertCreateCancel	Log in, create or cancel a text alert, log out
TransferBetweenOwnAccount	Log in, transfer an amount between personal accounts, log out
BillPaymentCreate	Log in, create a bill payment to a nominated beneficiary, log out
BusMiniStatement	Log in, generate a mini statement for nominated card numbers, log out
BusCustomStatement	Log in, generate a custom statement for nominated card numbers, log out

Use case	Description
BusTransferBetweenOwnAccounts	Log in, transfer $5.00 between personal accounts for nominated card number, log out
BusBillPaymentCreate	Log in, create a $5.00 bill payment to nominated beneficiary for nominated card numbers, log out

Once logged in, users were restricted to visibility of their own account information. This limited search activity to date/account ranges and meant that the resulting scripts were navigation-driven rather than data-driven.

Step 4: Performance Test Build

For Online Banking the performance test design was based on scaling from 1 to 1,000 concurrent virtual users. The test was designed so that all users were active after a 30-minute period. Because different numbers of users were assigned to each use case, each one had its own injection rate.

As shown in Table 4-2, the number of virtual users assigned to each use case was based on an assessment of their likely frequency during a 24-hour period (this information is displayed in the table's fourth column). For example, the transaction MiniStatement was considered the most common activity, since most customers who use an online banking service will want to check their latest statement. (I know it's usually my first port of call!) As a result, 416 virtual users out of the 1,000 were allocated to this activity.

Table 4-2. *Virtual user injection profile*

Use case	Starting virtual users	Injection rate per one virtual user	Target virtual users
MiniStatement	1	4 seconds	416
CustomStatement	1	26 seconds	69
ViewStandingOrders	1	26 seconds	69
TextAlertCreateCancel	1	2 minutes, 18 seconds	13
TransferBetweenOwnAccounts	1	12 seconds	145
BillPaymentCreate	1	8 seconds	225
BusMiniStatement	1	55 seconds	33
BusCustomStatement	1	7 minutes, 30 seconds	4

Use case	Starting virtual users	Injection rate per one virtual user	Target virtual users
BusTransferBetweenOwnAccounts	1	3 minutes	10
BusBillPaymentCreate	1	1 minute, 52 seconds	16

Differing numbers of users per use case required different injection rates to ensure that all virtual users were active after a 30-minute period; this information is demonstrated in column three, which shows how the virtual users were injected into the performance test. An important consideration to performance test design involves the accurate simulation of throughput. You may recall our discussion in Chapter 3 about concurrency and how the number of concurrent virtual users does not necessarily reflect the number of users actually logged in to the application. For Online Banking it was necessary to apply specific pacing values to the performance test and transactions; this ensured that throughput did not reach unrealistic levels during test execution.

Step 5: Performance Test Execution

Performance test execution principally involved multiple executions of ramp-up (without step) from 1 to 1,000 virtual users. As already mentioned, there was no integration of KPI monitoring of server or network data. Although the appropriate KPIs were identified as part of NFR capture, this information was monitored separately by the customer. Consequently, test execution had to be synchronized with existing monitoring tools—in this case, Perfmon.

The client also monitored the number of active users independently over the period of the performance test. This raises an interesting point concerning the definition of concurrency. For an application, concurrency measures the number of users actually logged in. But automated performance tools measure concurrency as the number of active virtual users, which exclude virtual users who are logged in but kept in a waiting state. If your performance testing use cases include logging in and logging out as part of the execution flow (as was the case with Online Banking), then the number of active virtual users will be fewer than the number of users as seen by the application. Therefore, you will need to inflate the number of virtual users or (as I did with Online Banking) use a slower pacing and execution rate to extend the time that virtual users remained logged in to the application and so create the proper concurrent load.

Online Banking Case Study Review

Let's now assess the Online Banking case study in light of the suggested performance testing requirements checklist:

The test team

The test team amounted to a single individual—certainly not the ideal situation, but not uncommon. Companies that provide performance testing as a service will generally supply a minimum of a project manager and at least one testing operative. When performance testing is carried out in-house, the situation is more fluid: many organizations do not have a dedicated performance testing team and tend to allocate resources on an ad hoc basis. For medium to large enterprises that carry out regular performance testing, a dedicated performance testing resource is a necessity in my view.

The test environment

In terms of providing an accurate performance testing environment, this case *was* ideal in terms of similarity to the deployment environment (i.e., they were one and the same). This relatively unusual situation led to the challenges of other activity affecting the performance testing results and the potentially negative effect of scalability testing on real application users. If these challenges can be managed successfully, then this approach provides an ideal, albeit unusual, performance test environment.

KPI monitoring

As mentioned previously, internal constraints prevented the use of any integrated monitoring with the performance testing tool. It was therefore necessary for customers to carry out their own monitoring, after which the testing team had to manually integrate this data with that provided by the performance testing tool. These situations are far from ideal because they complicate the analysis process and make it difficult to accurately align response time and scalability data with network and server performance metrics. However, sometimes there is no choice. In this particular case, the performance data was exported to MS Excel in CSV format to allow graphical analysis. The ability to import third-party data into the analysis module of your performance testing tool is an extremely useful feature if you are unable to use integrated KPI monitoring.

Performance targets

The sole performance target was scalability based on 1,000 concurrent users. There was no formal requirement for a minimum response time, which could have been estimated fairly easily since the application was already deployed and in regular use. The 1,000-user scalability target was arrived at in a rather arbitrary manner, and there was little evidence of consensus between the application stakeholders. This can be a significant risk to any performance testing project because it opens the door to challenges to interpretation of the results. In this case, independent monitoring of application end-user activity during performance test execution allayed any fears that the stated load was not achieved. As discussed earlier, even though the performance test tool achieved 1,000 concurrent virtual users,

this on its own would not have achieved the target of 1,000 concurrent application users. The difference was addressed by several hundred real users being active during performance test execution, who effectively topped up the number of concurrent application users generated by the performance testing tool.

Use-case scripting
: Ten use cases were selected and carefully documented in order to preclude any ambiguity over what was recorded. This followed the general trend whereby it is unusual to require more than 20 unique use cases in a performance testing engagement. The only required scripting changes involved login security.

Data requirements
: Input data requirements for Online Banking were limited to a list of valid card numbers. These numbers provided initial access to the application for each virtual user. The target database was the live database, which presented no problems in terms of content or realistic sizing. However, the use cases were limited in terms of what data could be committed to the database. Although there was significant testing of user activity via database reads, there was little opportunity to test the effects of significant database write activity, and this introduced an element of risk. Session data requirements involved the identification and reuse of a set of standard challenge/response questions together with random characters selected from a range of PINs associated with the card number data used as input to the test use cases.

Performance test design
: Performance test design focused on a single test type that I would characterize as a load test. The application was already comfortably supporting 380 users at peak periods, and the stated intention was to try to achieve 1,000 users—some five times the existing capacity. As already mentioned, the test design incorporated specific pacing changes to ensure that an accurate throughput was maintained.

Case Study 2: Call Center

The second case study involves an application that is very different from Online Banking: a typical call center that provides driver and vehicle testing services to a region of the United Kingdom. The Call Center application undergoes frequent code changes as a result of government legislation changes, and there is a requirement to performance test the application prior to each new release. Extensive user acceptance testing (UAT) is carried out to eliminate any significant functional issues before the updated application is released for performance testing.

Application Landscape

The landscape for the Call Center application is demonstrated by Figure 4-4.

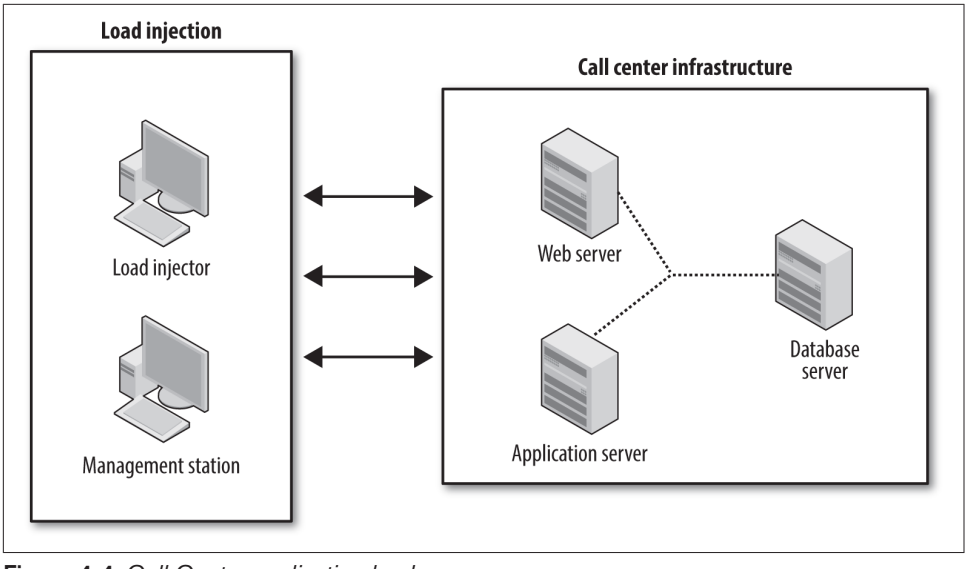


Figure 4-4. *Call Center application landscape*

Clients
: The application client is slightly unusual in that it is a traditional fat client written in Visual Basic but makes use of web services technology to connect to the application server layer. There are two versions of the fat client deployed: one for the call-center operators and another for operators at each of the regional vehicle testing centers. There is also a public-domain self-service website available with a limited set of functionality. Public users can use whatever Internet browser software they prefer.

Mid-tier servers
: The mid-tier application architecture consists of two load-balanced application servers running Windows 2003 Server OS directly connected to call-center clients and a web server layer providing connectivity for public domain users.

Database servers
: The database server is a single high-specification machine running the MS SQL database software on Windows 2003 Server.

Network infrastructure
: All servers reside in a single data center with gigabit Ethernet connectivity. All connectivity to the application servers is via 100 Mb LAN for call-center users or via ISP connection for public domain users.

Application Users

There is a typical maximum of 100 concurrent users during a normal working day in addition to an increasing number of self-service users through the 24/7 public-facing web portal. The call-center and regional test-center users each have their own login credentials, whereas web portal users provide information about their vehicle or driver's license to initiate a session. Typical transaction volumes are in the order of 1,500 transactions per 24-hour period.

Step 1: Pre-Engagement NFR Capture

Project time scales were typical, amounting to a week of lead time and five days to carry out the testing engagement. The client had outsourced the entire testing project for a number of years because it lacked in-house performance testing expertise.

A dedicated performance test environment was provided, although (as is often the case) the number and specification of servers differed from the live environment.

Performance targets for the Call Center case study were availability, concurrency, and response time. The application had to be available and performant at a concurrent load of 100 virtual users. In terms of response-time targets, performance at the web service level had to match or exceed that of the previous release.

Five use cases were identified as core to the performance testing project. Input data requirements were relatively complex and involved the following types of data.

Call-center use IDs
: A range of call-center login credentials was needed to provide variation in user sessions. For the target concurrency of 100 users, 20 sets of credential were provided.

Test centers
: Drivers can book appointments at a number of regional vehicle testing centers, so a list of all possible test centers was required for a realistic simulation.

Vehicle registration numbers
: Because appointments could be booked by vehicle registration number, a large number of valid registration numbers needed to be made available.

Driver's license numbers
: Appointments could also be booked on behalf of the driver, so the vehicle registration numbers had to be accompanied by a list of valid driver's license numbers.

The target database was a recent copy of the live database, so there were no problems with inadequate data sizing affecting performance test results. Although normally desirable, in this case the test database was rarely restored between performance test runs (it would have required a fresh cut of live data).

Performance tests used the ramp-up-with-step approach, reaching the target of 100 concurrent users. The tests would start focusing on baseline performance for each use case, which ran with one virtual user in steady state for 10 minutes. Ramp-up then came in increments of 25 users with a steady-state period of 15 minutes at each step. The 15-minute period of stability allowed incremental observation of steady-state performance.

Server and network KPIs focused once again on generic Windows performance metrics, although this time it was possible to use the integrated monitoring capability of the performance testing tool.

The testing team for this case study was more conventional in that a project manager and testing consultant were assigned to the engagement. As mentioned before, I see this as the typical staffing requirement for the majority of performance testing projects.

Step 2: Test Environment Build

In Chapter 3 we discussed the need to set up the performance test environment correctly. What is important is to retain the deployment architecture in terms of the different types of server tiers. The test environment for our case study manages to retain the tier deployment, but it has only a single application server whereas the live environment has two that are load-balanced. This implies less capacity, which we must take into account when building our performance tests.

As shown earlier, the performance test environment for the call center comprised the following:

- One web server
- One application server
- One database server

Load injection was provided by two workstations, one to inject the load and another to act as the management station.

Network analysis was carried out using packet-level sniffer technology to determine the amount of data presentation per use case and to model application performance in a WAN environment. This is a valuable exercise to carry out prior to performance testing because it enables additional tuning and optimization of an application, potentially removing a layer of problems that may affect performance and scalability.

Step 3: Use-Case Scripting

For the Call Center case study, five use cases were identified that could all be considered active. Since the call-center and regional test-center clients were similar, the test use cases were selected only from the call center. Session data requirements were com-

plex, requiring the extraction and reuse of information relating to the appointment booking selection. It was necessary to add code to the scripts to extract the required information. Unforeseen scripting challenges can have a significant impact on performance testing projects; it is therefore extremely important to identify and resolve them in advance of the scripting phase.

To ensure that the target concurrency was achieved, login and logout were not included as iterative activities during performance test execution. Each virtual user logged in at the start of the performance test and remained logged in until test conclusion, when the final action was to log out. Table 4-3 lists the use cases tested.

Table 4-3. *Call Center use cases*

Use case	Description
DriverBookingSearch	Log in, search for first available driver booking slots, log out
FindSlotFirstAvailable	Log in, search for first available vehicle booking slots, log out
FindVRN	Log in, find vehicle by registration number, log out
SearchForAppointment	Log in, find appointment by appointment ID number, log out
VehicleBooking	Log in, make a vehicle booking, log out

Step 4: Performance Test Scenario Build

The Call Center application architecture is based on web services that invoke stored procedure calls on the MS SQL database. Historical problems with changes to web services and stored procedure calls between releases highlighted the importance of timing each web service call and monitoring the impact on the database server. Thus, timing checkpoints were inserted between all web service calls to allow granular timing of each request.

Step 5: Performance Test Execution

Performance test execution involved the following steps:

1. Pipe-clean test execution for each use case, executing as a single user for a period of 10 minutes to establish a performance baseline.
2. Then ramp-up with step execution for each use case in increments of 25 virtual users, with a steady-state period of 15 minutes at each increment.

Since the use cases in the Call Center case study did not include login and logout within their execution loop, there were no challenges with maintaining true application user concurrency (as had occurred in the Online Banking case study).

Call Center Case Study Review

Let's now assess the Call Center case study in terms of our checklist:

The test team

The test team consisted of a project manager and a single test operative, meeting what I consider to be typical requirements for a performance testing project. Because the application was not developed in-house, any problems that arose had to be fed back to the application vendor. This meant that test execution had to be suspended while the problems were investigated, which led on more than one occasion to a postponement of the target deployment date.

The test environment

The environment was typical: dedicated hardware isolated from the production infrastructure in its own network environment. Application tier deployment largely matched that of the production application; the one major difference was the single application server versus the load-balanced pair of the production environment. Any significant difference in the number of application servers between test and production could lead to a bottleneck in the mid-tier. In such cases you should approach any stress-testing requirements with caution. An excessively aggressive injection profile might result in premature application server overload and misleading results.

KPI monitoring

Because a dedicated testing environment was available, there were no obstacles to using the integrated server and network monitoring capability of the chosen performance testing tool. We obtained server information by integrating the testing tool with Microsoft's Perfmon application. This is the preferred situation, and it allowed automated correlation of response time and monitoring data at the conclusion of each performance test execution—an important benefit of using automated performance testing tools.

Performance targets

Performance targets were supporting a concurrency of 100 users and ensuring that response time did not deteriorate from that observed during the previous round of testing. We used checkpointing to focus on web service performance and to provide an easy mechanism for comparing releases.

Scripting

Five use cases were selected from the call-center client to be used as the basis for performance testing the application. Scripting challenges focused on the unusual nature of the application architecture, which involved large numbers of web service requests, each with an extensive set of parameters. These parameters would frequently change between releases, and the accompanying documentation was

often unclear. This made it difficult to identify exactly what changes had occurred. As a result, scripting time varied considerably between testing cycles and sometimes required creation of a completely new set of scripts (rather than the preferred method of copying the scripts from the previous testing cycle and simply modifying the code). This shows the vulnerability of scripted use cases to code being invalidated by changes introduced in a new release.

Data requirements

Input data requirements for the Call Center were more complex than those for Online Banking. Large numbers of valid booking and vehicle registration numbers were needed before testing could proceed. We handled this task by extracting the relevant information from the target database via SQL scripts and then outputting the information in CSV format files. The target database was a recent copy of the live database, which once again presented no problems in terms of realistic content or sizing. Since this was a true test environment, there were no restrictions on the type of end-user activity that could be represented by the scripted use cases. The one negative factor was the decision not to restore the database between test runs. I always advise that data be restored between test executions in order to avoid any effects on performance test results that are caused by changes in database content from previous test runs. Session data requirements were rather complex and involved the interception and reuse of several different data elements, which represent server-generated information about the vehicle booking process. This made creating the original scripts a nontrivial exercise requiring the involvement of the application developers.

Performance test design

This test design used the ramp-up-with-step approach, which allowed observation of steady-state response time of the application at increments of 25 virtual users for intervals of approximately 15 minutes.

Summary

I hope that the case studies in this chapter have struck some chords and that the checklists provided outline a practical way for you to approach application performance testing. The next step is to look at test execution and try to make sense of the information coming back from your performance testing toolset and KPI monitoring.

CHAPTER FIVE

Interpreting Results: Effective Root-Cause Analysis

> Statistics are like lamp posts: they are good to lean on, but they often don't shed much light.
>
> —Anonymous

OK, SO I'M RUNNING A PERFORMANCE TEST—WHAT'S IT TELLING ME? THE correct interpretation of results is obviously vitally important. Since we're assuming you've (hopefully) set proper performance targets as part of your testing requirements, you should be able to spot problems quickly during the test or as part of the analysis process at test completion.

If your application concurrency target was 250 users, crashing and burning at 50 represents an obvious failure. What's important is having all the necessary information at hand to detect when things go wrong and diagnose what happened when they do. Performance test execution is often a series of false starts, especially if the application you're testing has significant design or configuration problems.

I'll begin this chapter by talking a little about the types of information you should expect to see from an automated performance test tool. Then we'll look at real-world examples, some of which are based on the case studies in Chapter 4.

NOTE

For the purposes of this chapter you can define *root-cause analysis* as a method of problem solving that tries to identify the root causes of (performance) problems. This is distinct from a causal factor that may affect an event's outcome but is not a root cause. (Courtesy of Wikipedia.)

The Analysis Process

Depending on the capabilities of your performance test tooling, analysis can be performed either as the test executes (in real time) or at its conclusion. Let's take a look at each approach in turn.

Real-Time Analysis

Real-time analysis is very much a matter of what I call *watchful waiting*. You're essentially waiting for something to happen or for the test to complete without *apparent* incident. If a problem does occur, your KPI monitoring tools are responsible for reporting the location of the problem in the application landscape. If your performance testing tool can react to configured events, you should make use of this facility to alert you when any KPI metric is starting to come off the rails.

> **NOTE**
>
> As it happens, *watchful waiting* is also a term used by the medical profession to describe watching the progress of some fairly nasty diseases. Here, however, the worst you're likely to suffer from is boredom or perhaps catching cold after sitting for hours in an overly air-conditioned data center. (These places are definitely not designed with humans in mind!)

So while you are sitting there bored and shivering, what should you expect to see as a performance test is executing? The answer very much depends on the capabilities of your performance testing tool. As a general rule, the more you pay, the more sophisticated the analysis capabilities are. But at an absolute minimum, I would expect to see the following:

- You need response-time data for each use case in the performance test in tabular and graphical form. The data should cover the complete use case as well as any subparts that have been separately marked for analysis. This might include such activities as the time to complete login or the time to complete a search.
- You must be able to monitor the injection profile for the number of users assigned to each script and a total value for the overall test. From this information you will be able to see how the application reacts in direct response to increasing user load and throughput.
- You should be able to monitor the state of all load injectors so you can check that they are not being overloaded.
- You need to monitor data that relates to any server, application server, or network KPIs that have been set up as part of the performance test. This may involve integration with other monitoring software if you're using a prepackaged performance testing solution rather than just a performance testing tool.

- You need a display of any performance thresholds configured as part of the test and an indication of any breaches that occur.
- You need a display of all errors that occur during test execution that includes the time of occurrence, the virtual users affected, an explanation of the error, and possibly advice on how to correct the error condition.

Post-Test Analysis

All performance-related information that was gathered during the test should be available at the test's conclusion and may be stored in a database repository or as part of a simple file structure. The storage structure is not particularly important so long as you're not at risk of losing data and it's readily accessible. At a minimum, the data you acquired during real-time monitoring should be available afterward. Ideally, you should also have additional information available, such as error analysis for any virtual users who encountered problems during test execution.

This is one of the enormous advantages of using automated performance testing tools: the output of each test run is stored for future reference. This means that you can easily compare two or more sets of test results and see what's changed. In fact, many tools provide a templating capability so that you can define in advance the comparison views you want to see.

As with real-time analysis, the capabilities of your performance testing tool will largely determine how easy it is to decipher what has been recorded. The less expensive and open source tools tend to be weaker in the areas of real-time analysis and post-test analysis and diagnostics.

Make sure that you make a record of what files represent the output of a particular performance test execution. It's quite easy to lose track of important data when you're running multiple test iterations.

Types of Output from a Performance Test

You've probably heard the phrase "Lies, damned lies, and statistics." Cynicism aside, statistical analysis lies at the heart of all automated performance test tools. If statistics are close to your heart, that's well and good, but for the rest of us I thought it a good idea to provide a little refresher on some of the jargon to be used in this chapter. For more detailed information, take a look at Wikipedia or any college text on statistics.

Statistics Primer

What follows is a short discussion on the elements of statistics that are relevant to performance testing:

Mean and median

Loosely described, the mean is the average of a set of values. It is commonly used in performance testing to derive average response times. It should be used in conjunction with the *n*th percentile (described later) for best effect. There are actually several different types of mean value, but for the purpose of performance testing we tend to focus on what is called the *arithmetic mean*.

For example: to determine the arithmetic mean of 1, 2, 3, 4, 5, 6, simply add them together and then divide by the number of values (6). The result is an arithmetic mean of 3.5.

A related metric is the median, which is simply the middle value in a set of numbers. This is useful in situations where the calculated arithmetic mean is skewed by a small number of outliers, resulting in a value that is not a true reflection of the average.

For example: the arithmetic mean for the number series 1, 2, 2, 2, 3, 9 is 3.17, but the majority of values are 2 or less. In this case, the median value of 2 is thus a more accurate representation of the true average.

Standard deviation and normal distribution

Another common and useful indicator is standard deviation, which refers to the average variance from the calculated mean value. It's based on the assumption that most data in random, real-life events exhibits a *normal distribution*—more familiar to most of us from high school as a *bell curve*. The higher the standard deviation, the farther the items of data tend to lie from the mean. Figure 5-1 provides an example courtesy of Wikipedia.

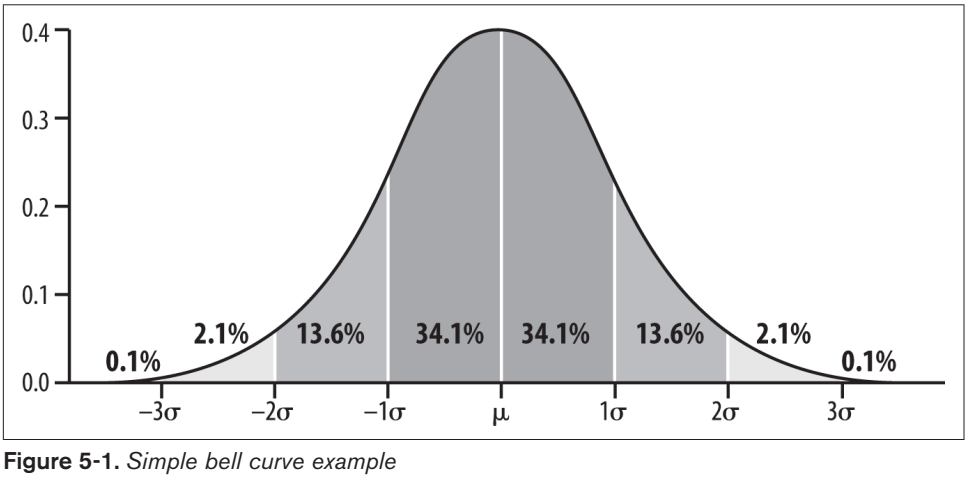


Figure 5-1. *Simple bell curve example*

+ In performance testing terms, a high standard deviation can indicate an erratic end-user experience. For example, a use case may have a calculated mean response time of 40 seconds but a standard deviation of 30 seconds. This would mean that an end user has a high chance of experiencing a response time as low as 25 and as high as 55 seconds for the same activity. You should seek to achieve a small standard deviation.

nth percentile

Percentiles are used in statistics to determine where a certain percent of results fall. For instance, the 40th percentile is the value at or below which 40 percent of a set of results can be found. Calculating a given percentile for a group of numbers is not straightforward, but your performance testing tool should handle this automatically. All you normally need to do is select the percentile (anywhere from 1 to 100) to eliminate the values you want to ignore.

For example, let's take the set of numbers from our earlier skewed example (1, 2, 2, 2, 3, 9) and ask for the 90th percentile. This would lie between 3 and 9, so we eliminate the high value 9 from the results. We could then apply our arithmetic mean to the remaining five values, giving us the much more representative value of 2 (1 + 2 + 2 + 2 + 3 divided by 5).

Response-time distribution

Based on the normal distribution model, this is a way of aggregating all the response times collected during a performance test into a series of groups, or buckets. This distribution is usually rendered as a bar graph (see Figure 5-2), where each bar represents a range of response times and what percentage of transaction iterations fell into that range. You can normally define how many bars you want in the graph and the time range that each bar represents. The y-axis is simply an indication of measured response time.

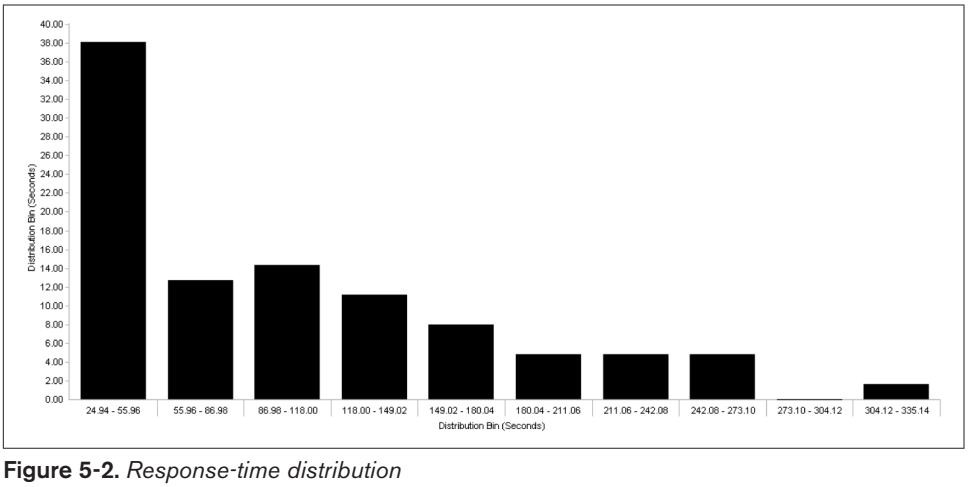


Figure 5-2. *Response-time distribution*

Response-Time Measurement

The first set of data you will normally look at is a measurement of application—or, more correctly, *server*—response time. Automated performance test tools typically measure the time it takes for an end user to submit a request to the application and receive a response. If the application fails to respond to a client request in the required time, the performance tool will record some form of time-out error. If this situation occurs frequently, it may be indicative of an overload condition somewhere in the application landscape. We then need to check the server and network KPIs to help us determine where the overload occurred.

> **TIP**
>
> An overload doesn't always represent a problem with the application. It may simply mean that you need to increase one or more time-out values in the performance test configuration.

Any application time spent exclusively on the client is rendered as periods of think time, which represents the normal delays and hesitations that are part of end-user interaction with a software application. Performance testing tools generally work at the middleware level—that is, under the presentation layer—so they have no concept of events such as clicking on a combo box and selecting an item unless this action generates traffic on the wire.

User activity like this will normally appear as a period of inactivity or sleep time and may represent a simple delay to simulate the user digesting what has been displayed on the screen. If you need to time such activities separately, you may need to combine functional and performance testing tools as part of the same performance test (see Chapter 9).

These think-time delays are not normally included in response-time measurement, since your focus is on how long it took for the server to send back a complete response after a request is submitted. Some tools may break this down further by identifying at what point the server started to respond and how long it took to complete sending the response.

Moving on to some examples, the next three figures demonstrate typical response-time data that would be available as part of the output of a performance test. This information is commonly available both in real time (as the test is executing) and as part of the completed test results.

Figure 5-3 depicts simple use-case response time (y-axis) versus the duration of the performance test (x-axis). On its own this metric tells us little more than the response-time behavior for each use case over the duration of the performance test. If there are

any fundamental problems with the application, then response-time performance is likely to be bad regardless of the number of active virtual users.

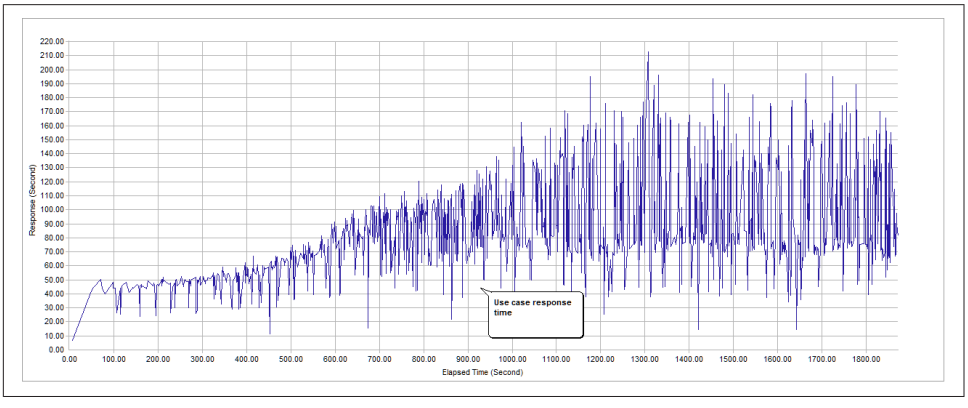


Figure 5-3. *Use-case response time for the performance test duration*

Figure 5-4 shows response time for the same test but this time adds the number of concurrent virtual users. Now you can see the effect of increasing numbers of virtual users on application response time. You would normally expect an increase in response time as more virtual users become active, but this should not be in lock step with increasing load.

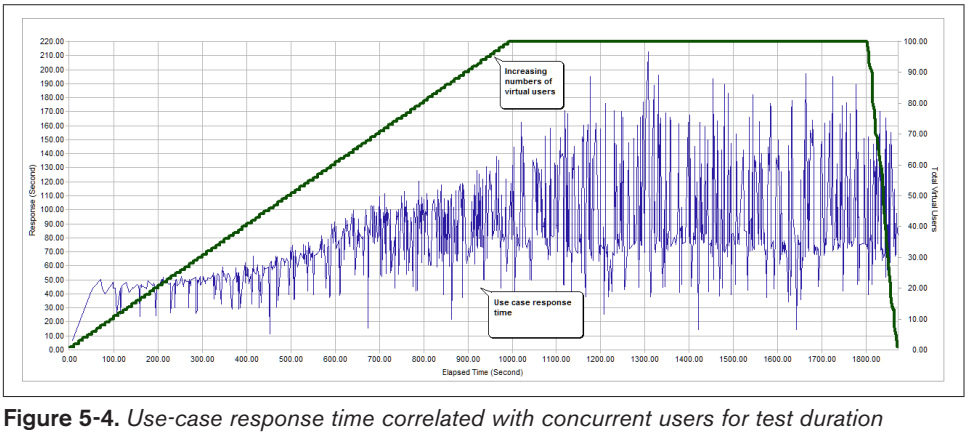


Figure 5-4. *Use-case response time correlated with concurrent users for test duration*

Figure 5-5 builds on the previous two figures by adding response-time data for the checkpoints that were defined as part of the use case. As mentioned in Chapter 3, adding checkpoints improves the granularity of response-time analysis and allows correlation of poor response-time performance with the specific activities within a use case. The figure shows that the spike in use-case response time at approximately 1,500 seconds corresponded to an even more dramatic spike in checkpoints but did *not* correspond to the number of active virtual users.

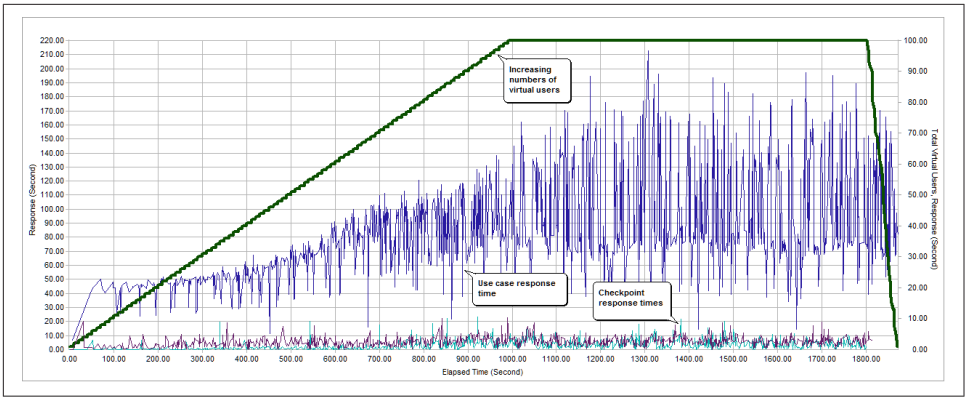


Figure 5-5. *Use-case and checkpoint response time correlated with concurrent users*

In fact, the response-time spike at about 1,500 seconds was caused by an invalid set of login credentials supplied as part of the input data. This demonstrates the effect that inappropriate data can have on the results of a performance test.

Figure 5-6 provides a tabular view of response-time data graphed in the earlier examples. Here we see references to mean and standard deviation values for the complete use case and for each checkpoint.

Name	Min	Mean	Max	Last	Std.Dev.
Transaction Response Time	24.9380	86.0258	335.1410	202.7350	61.5771
Total Running Virtual Users	10.0000	458.7383	913.0000	913.0000	261.8142
Login Response Time	0.7660	7.3413	24.2350	7.1870	5.3617
Security Question Response Time	0.5930	6.7524	20.6250	6.9690	5.0116
Click Custom Statements Response Time	0.8120	6.0654	35.0470	4.7970	5.2828
Show Statement Response Time	0.7970	6.0582	15.9690	8.4840	4.0250
Logoff Response Time	1.2030	15.1568	246.0940	10.7820	32.2105

Figure 5-6. *Table of response-time data*

Performance testing tools should provide us with a clear starting point for analysis. For example, Figure 5-7 lists the 10 worst-performing checkpoints for all the use cases within a performance test. This sort of graph is useful for highlighting problem areas when there are many checkpoints and use cases.

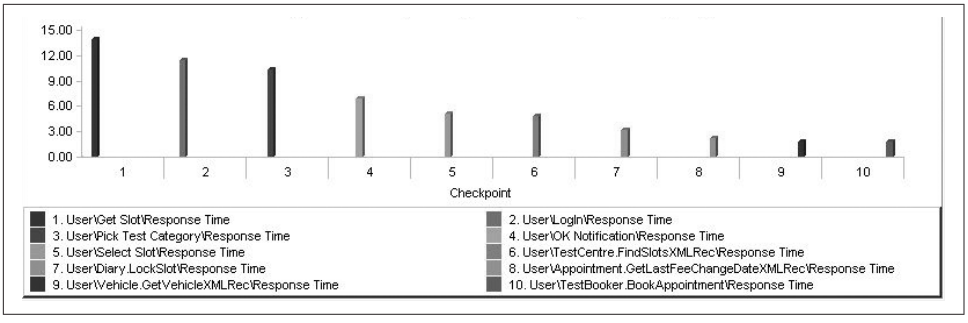


Figure 5-7. *Top 10 worst-performing checkpoints*

Throughput and Capacity

Along with response time, performance testers are usually most interested in how much data or how many use cases can be handled simultaneously. You can think of this measurement as throughput to emphasize how fast a particular number of use cases are handled, or as capacity to emphasize how many use cases can be handled in a particular time period.

Figure 5-8 illustrates use-case throughput per second for the duration of a performance test. This view shows when peak throughput was achieved and whether any significant variation in throughput occurred at any point.

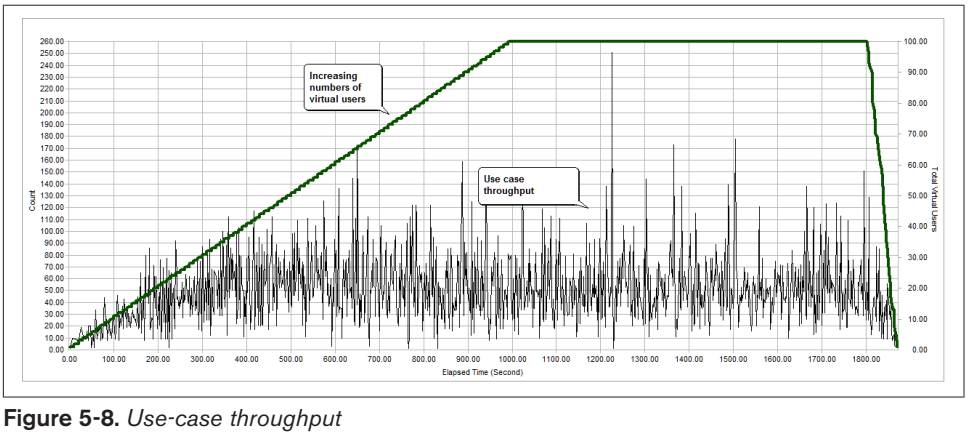


Figure 5-8. *Use-case throughput*

A sudden reduction in throughput invariably indicates problems and may coincide with errors encountered by one or more virtual users. I have seen this frequently occur when the web server tier reaches its saturation point for incoming requests. Virtual users start to stall while waiting for the web servers to respond, resulting in an attendant drop in throughput. Eventually users will start to time out and fail, but you may find that throughput stabilizes again (albeit at a lower level) once the number of active users is reduced to a level that can be handled by the web servers. If you're

really unlucky, the web or application servers may not be able to recover and all your virtual users will fail. In short, reduced throughput is a useful indicator of the capacity limitations in the web or application server tier.

Figure 5-9 looks at the number of GET and POST requests for active concurrent users during a web-based performance test. These values should gradually increase over the duration of the test. Any sudden drop-off, especially when combined with the appearance of virtual user errors, could indicate problems at the web server tier.

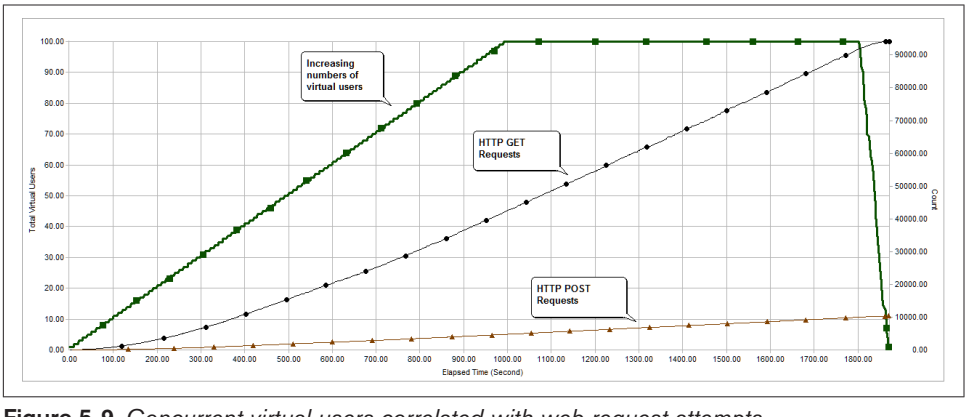


Figure 5-9. *Concurrent virtual users correlated with web request attempts*

Of course, the web servers are not always the cause of the problem. I have seen many cases where virtual users timed out waiting for a web server response, only to find that the actual problem was a long-running database query that had not yet returned a result to the application or web server tier. This demonstrates the importance of setting up KPI monitoring for *all* server tiers in the system under test (SUT).

Monitoring Key Performance Indicators

As discussed in Chapter 3, you can determine server and network performance by configuring your monitoring software to observe the behavior of key generic and application-specific performance counters. This monitoring software may be included or integrated with your automated performance testing tool, or it may be an independent product. Any server and network KPIs configured as part of performance testing requirements fall into this category.

You can use a number of mechanisms to monitor server and network performance, depending on your application technology and the capabilities of your performance testing solution. The following sections divide the tools into categories, describing the most common technologies in each individual category.

Remote monitoring

These technologies provide server performance data (along with other metrics) from a remote system; that is, the server being tested passes data over the network to the part of your performance testing tool that runs your monitoring software.

The big advantage of using remote monitoring is that you don't usually need to install any software onto the servers you want to monitor. This circumvents problems with internal security policies that prohibit installation of any software that is not part of the standard build. A remote setup also makes it possible to monitor many servers from a single location.

That said, each of these monitoring solutions needs to be activated and correctly configured. You'll also need to be provided with an account that has sufficient privilege to access the monitoring software. You should be aware that some forms of remote monitoring, particularly SNMP or anything using remote procedure calls (RPC), may be prohibited by site policy because they can compromise security.

Common remote monitoring technologies include the following:

Windows Registry
: This provides essentially the same information as Microsoft's Performance Monitor (Perfmon). Most performance testing tools provide this capability. This is the standard source of KPI performance information for Windows operating systems and has been in common use since Windows 2000 was released.

Web-Based Enterprise Management (WBEM)
: Web-Based Enterprise Management is a set of systems management technologies developed to unify the management of distributed computing environments. WBEM is based on Internet standards and Distributed Management Task Force (DMTF) open standards: the Common Information Model (CIM) infrastructure and schema, CIM-XML, CIM operations over HTTP, and WS-Management. Although its name suggests that WBEM is web-based, it is not necessarily tied to any particular user interface. Microsoft has implemented WBEM through its Windows Management Instrumentation (WMI) model. Its lead has been followed by most of the major Unix vendors, such as SUN (now Oracle) and HP. This is relevant to performance testing because Windows Registry information is so useful on Windows systems and is universally used as the source for monitoring, while WBEM itself is relevant mainly for non-Windows operating systems. Many performance testing tools support Microsoft's WMI, although you may have to manually create the WMI counters for your particular application and there may be some limitations in each tool's WMI support.

Simple Network Monitoring Protocol (SNMP)

A misnomer if ever there was one: I don't think *anything* is simple about using SNMP. However, this standard has been around in one form or another for many years and can provide just about any kind of information for any network or server device. SNMP relies on the deployment of management information base (MIB) files that contain lists of object identifiers (OIDs) to determine what information is available to remote interrogation. For the purposes of performance testing, think of an OID as a counter of the type available from Perfmon. The OID, however, can be a lot more abstract, providing information such as the fan speed in a network switch. There is also a security layer based on the concept of *communities* to control access to information. Therefore, you need to ensure that you can connect to the appropriate community identifier; otherwise, you won't see much. SNMP monitoring is supported by a number of performance tool vendors.

Java Monitoring Interface (JMX)

Java Management Extensions is a Java technology that supplies tools for managing and monitoring applications, system objects, devices (such as printers), and service-oriented networks. Those resources are represented by objects called MBeans (for managed beans). JMX is useful mainly for monitoring Java application servers such as IBM WebSphere, ORACLE WebLogic, and JBOSS. JMX support is version-specific, so you need to check which versions are supported by your performance testing solution.

Rstatd

This is a legacy RPC-based utility that has been around in the Unix world for some time. It provides basic kernel-level performance information. This information is commonly provided as a remote monitoring option, although it is subject to the same security scrutiny as SNMP because it uses RPC.

Installed agent

When it isn't possible to use remote monitoring—perhaps because of network firewall constraints or security policies—your performance testing solution may provide an agent component that can be installed directly onto the servers you wish to monitor. You may still fall foul of internal security and change requests, causing delays or preventing installation of the agent software, but it's a useful alternative if your performance testing solution offers this capability and the remote monitoring option is not available.

Server KPI Performance

Server KPIs are many and varied. However, two that stand out from the crowd are how busy the server CPUs are and how much virtual memory is available. These two metrics on their own can tell you a lot about how a particular server is coping with

increasing load. Some automated tools provide an expert analysis capability that attempts to identify any anomalies in server performance that relate to an increase in the number of virtual users or use-case response time (e.g., a gradual reduction in available memory in response to an increasing number of virtual users).

Figure 5-10 demonstrates a common correlation by mapping the number of concurrent virtual users against how busy the server CPU is. These relatively simple views can quickly reveal if a server is under stress. The figure depicts a ramp-up-with-step virtual user injection profile.

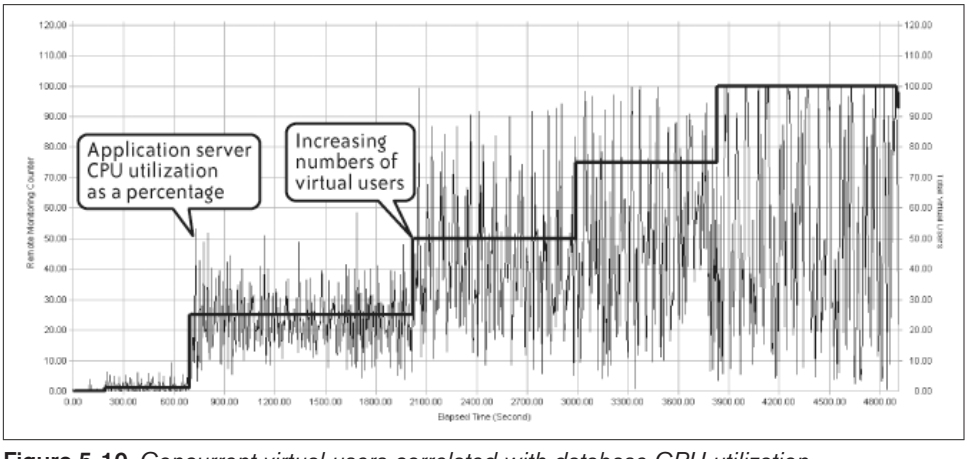


Figure 5-10. *Concurrent virtual users correlated with database CPU utilization*

A notable feature is the spike in CPU usage right after each step up in virtual users. For the first couple of steps the CPU soon settles down and handles that number of users better, but as load increases, the CPU utilization becomes increasingly intense. Remember that the injection profile you select for your performance test scripts can create periods of artificially high load, especially right after becoming active, so you need to bear this in mind when analyzing test results.

Network KPI Performance

As with server KPIs, any network KPIs instrumented as part of the test configuration should be available afterward for post-mortem analysis. Figure 5-11 demonstrates typical network KPI data that would be available as part of the output of a performance test.

Figure 5-11 also correlates concurrent virtual users with various categories of data presented to the network. This sort of view provides insight into the data footprint of an application, which can be seen either from the perspective of a single use case or single user (as may be the case when baselining) or during a multi-use-case performance

test. This information is useful for estimating the application's potential impact on network capacity when deployed.

In this example it's pretty obvious that a lot more data is being received than sent by the client, suggesting that whatever caching mechanism is in place may not be optimally configured.

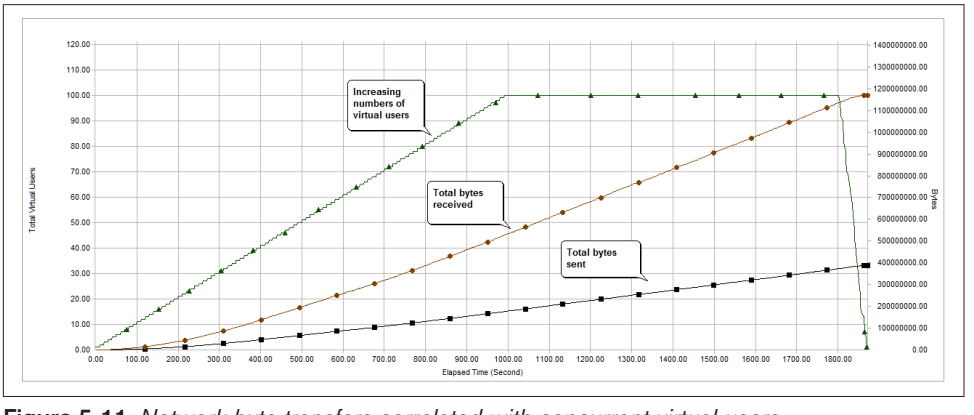


Figure 5-11. *Network byte transfers correlated with concurrent virtual users*

Load Injector Performance

Every automated performance test uses one or more workstations or servers as load injectors. It is very important to monitor the stress on these machines, as they create increasing numbers of virtual users. As mentioned in Chapter 4, if the load injectors themselves become overloaded, your performance test will no longer represent real-life behavior and so will produce invalid results that may lead you astray. Overstressed load injectors don't necessarily cause the test to fail, but they could easily distort the use case and data throughput as well as the number of virtual user errors that occur during test execution. Carrying out a dress rehearsal in advance of full-blown testing will help ensure that you have enough injection capacity.

Typical metrics you need to monitor include the following:

- Percent of CPU utilization
- Amount of free memory
- Page file utilization
- Disk time
- Amount of free disk space

Figure 5-12 offers a typical runtime view of load injector performance monitoring. In this example, we see that load injector CPU increases alarmingly in response to increasing numbers of virtual users suggesting we need more load injector resource.

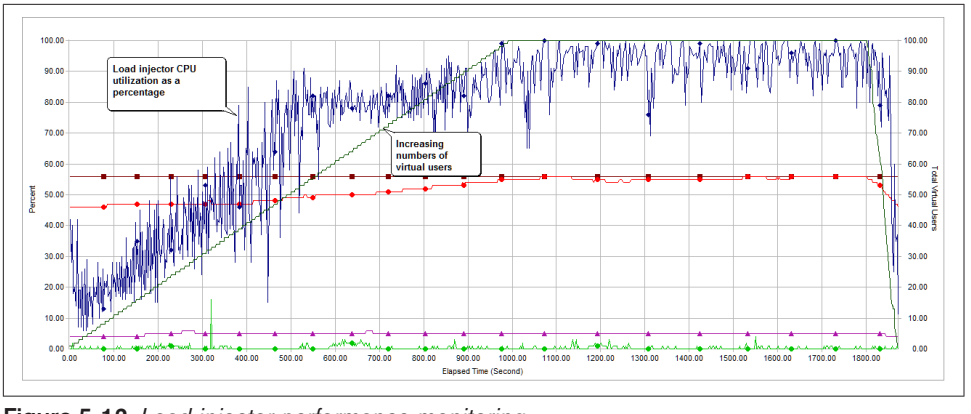


Figure 5-12. *Load injector performance monitoring*

Root-Cause Analysis

So what are we looking for to determine how our application is performing? Every performance test (should) offer a number of KPIs that can provide the answers.

TIP

Before proceeding with analysis, you might want to adjust the time range of your test data to eliminate the startup and shutdown periods that can denormalize your statistical information. This also applies if you have been using a ramp-up-with-step injection profile. If each ramp-up step is carried out en masse (e.g., you add 25 users every 15 minutes), then there will be a period of artificial stress immediately after the point of injection that may influence your performance stats. After very long test runs, you might also consider *thinning* the data—reducing the number of data points to make analysis easier. Most automated performance tools offer the option of data thinning.

Scalability and Response Time

A good model of scalability and response time demonstrates a moderate but acceptable increase in mean response time as virtual user load and throughput increase. A poor model exhibits quite different behavior: as virtual user load increases, response time increases in lockstep and either does not flatten out or starts to become erratic, exhibiting high standard deviations from the mean.

Figure 5-13 demonstrates good scalability. The line representing mean response time increases to a certain point in the test but then gradually flattens out as maximum concurrency is reached. Assuming that the increased response time remains within

your performance targets, this is a good result. (The spikes toward the end of the test were caused by termination of the performance test and don't indicate a sudden application-related problem.)

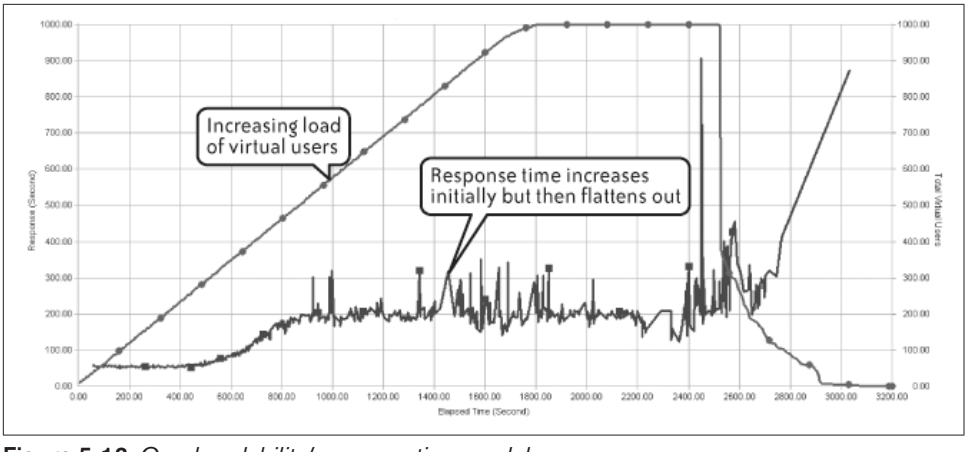


Figure 5-13. *Good scalability/response-time model*

The next two figures demonstrate undesirable response-time behavior. In Figure 5-14, the line representing mean use-case response time closely follows the line representing the number of active virtual users until it hits approximately 750. At this point, response time starts to become erratic, indicating a high standard deviation and a potentially bad end-user experience.

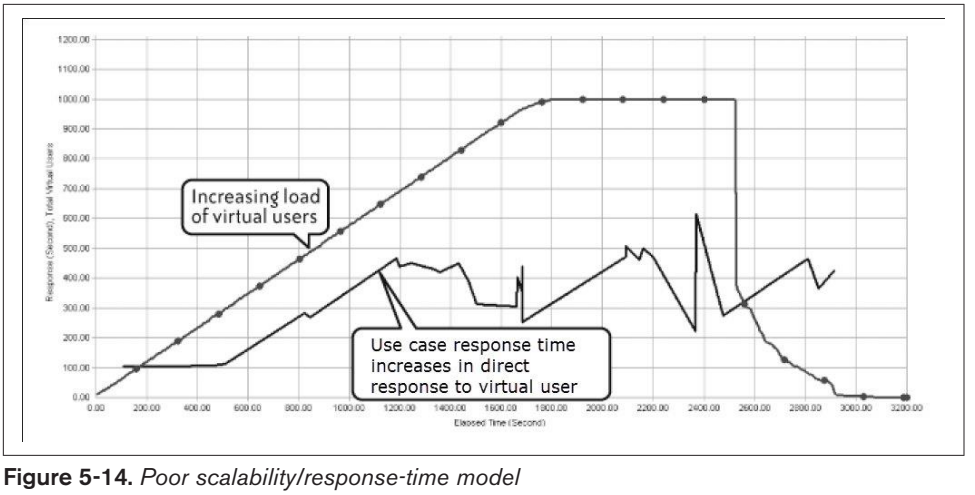


Figure 5-14. *Poor scalability/response-time model*

Figure 5-15 demonstrates this same effect in more dramatic fashion. Response time and concurrent virtual users in this example increase almost in lockstep.

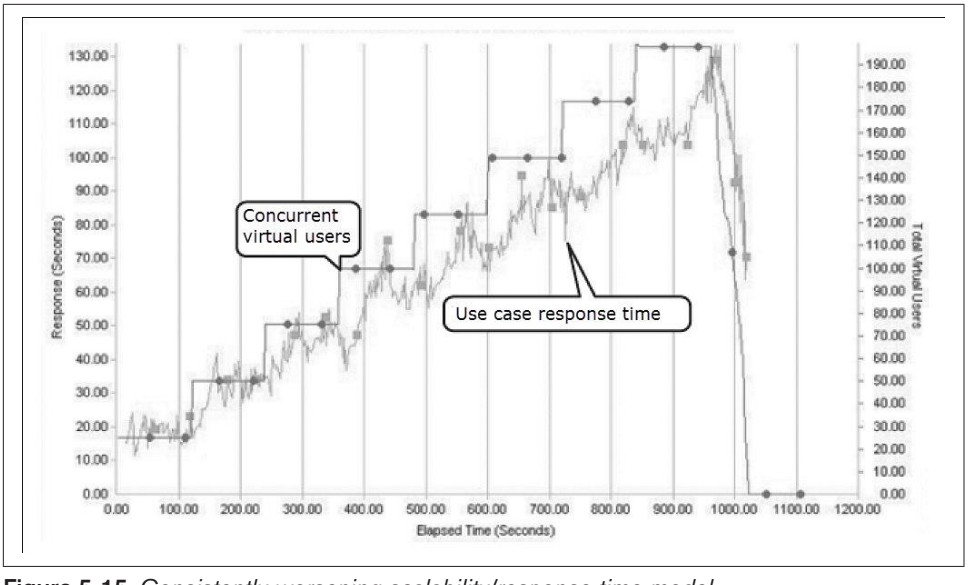


Figure 5-15. *Consistently worsening scalability/response-time model*

Digging Deeper

Of course, scalability and response-time behavior is only half the story. Once you see a problem, you need to find the cause. This is where the server and network KPIs come into play.

Examine the KPI data to see whether any metric correlates with the observed scalability/response-time behavior. Some performance testing tools have an autocorrelation feature that provides this information at the end of the test. Other tools require a greater degree of manual effort to achieve the same result.

Figure 5-16 demonstrates how it is possible to map server KPI data with scalability and response-time information to perform this type of analysis. It also builds on the previous examples, adding the Windows Server KPI metric called *context switches per second*. This metric is a good indicator on Windows servers of how well the CPU is handling requests from active threads. This example shows the CPU quickly reaching a high average value, indicating a potential lack of CPU capacity for the load being applied. This, in turn, has a negative impact on the ability of the application to scale efficiently and thus on response time.

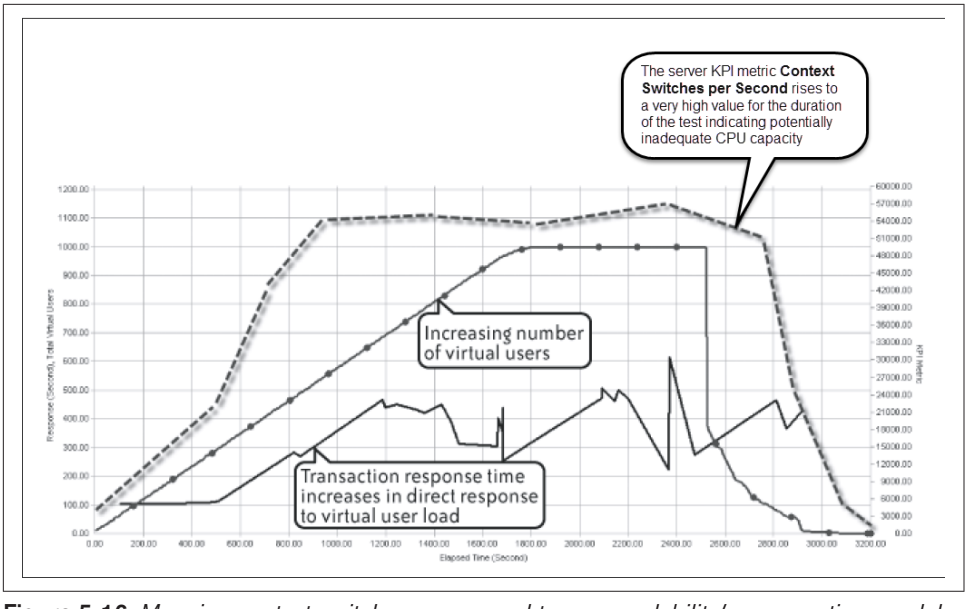


Figure 5-16. *Mapping context switches per second to poor scalability/response-time model*

Inside the Application Server

In Chapter 3 we discussed setting appropriate server and network KPIs. I suggested defining these as layers, starting from a high-level, generic perspective and then adding others that focus on specific application technologies. One of the more specific KPIs concerned the performance of any application server component present in the application landscape. This type of monitoring lets you look inside the application server down to the level of class and methods.

Simple generic monitoring of application servers won't tell you much if there are problems in the application. In the case of a Java-based application server, all you'll see is one or more *java.exe* processes consuming a lot of memory or CPU. You need to discover which specific component calls are causing the problem.

You may also run into the phenomenon of the *stalled thread*, where an application server component is waiting for a response from another internal component or from another server such as the database host. When this occurs, there is usually no indication of excessive CPU or memory utilization, just slow response time. The cascading nature of these problems makes them difficult to diagnose without detailed application server analysis.

Figures 5-17 and 5-18 present examples of the type of performance data that this analysis provides.

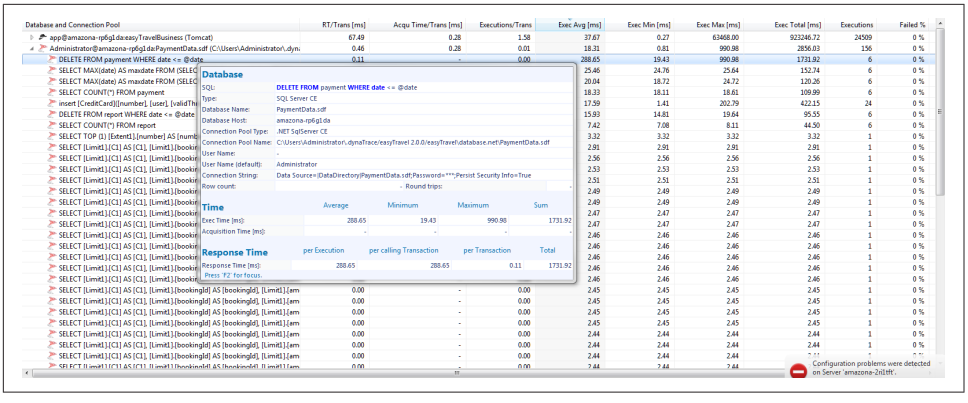


Figure 5-17. *Database hotspots*

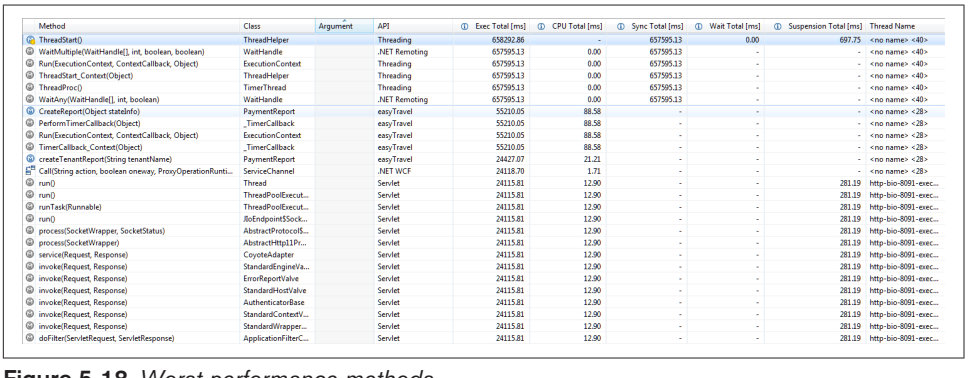


Figure 5-18. *Worst performance methods*

Looking for the Knee

You may find that, at a certain throughput or number of concurrent virtual users during a performance test, the performance testing graph demonstrates a sharp upward trend—known colloquially as a *knee*—in response time for some or all use cases. This indicates that some capacity limit has been reached within the hosting infrastructure and has started to affect application response time.

Figure 5-19 demonstrates this effect during a large-scale, multi-use-case performance test. At approximately 260 concurrent users, there is a distinct knee in the measured response time of all use cases. After you observe this behavior, your next step should be to look at server and network KPIs at the same point in the performance test. This may, for example, reveal high CPU utilization or insufficient memory on one or more of the application server tiers.

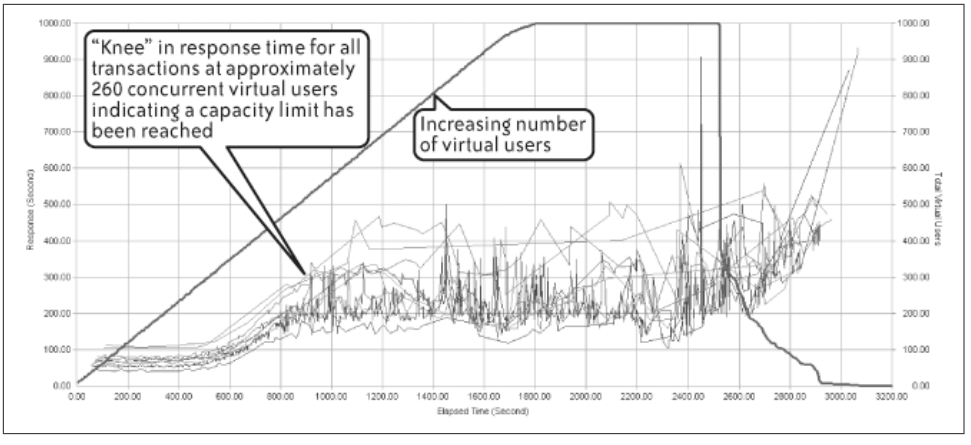


Figure 5-19. *Knee performance profile indicating that capacity limits have been reached*

In this particular example, the application server was found to have high CPU utilization and very high context switching, indicating a lack of CPU capacity for the required load. To fix the problem, the application server was upgraded to a more powerful machine. It is a common strategy to simply throw hardware at a performance problem. This may provide a short-term fix, but it carries a cost and a significant amount of risk that the problem will simply reappear at a later date. In contrast, a clear understanding of the problem's cause provides confidence that the resolution path chosen is the correct one.

Dealing with Errors

It is most important to examine any errors that occur during a performance test, since these can also indicate hitting some capacity limit within the SUT. By *errors* I mean virtual user failures, both critical and (apparently) noncritical. Your job is to find patterns of when these errors start to occur and the rate of further errors occurring after that point. A sudden appearance of a large number of errors may coincide with the knee effect described previously, providing further confirmation that some limit has been reached. Figure 5-20 adds a small line to the earlier graph that demonstrated poor performance, showing the sudden appearance of errors. The errors actually start before the test shows any problem in response time, and they spike about when response time suddenly peaks.

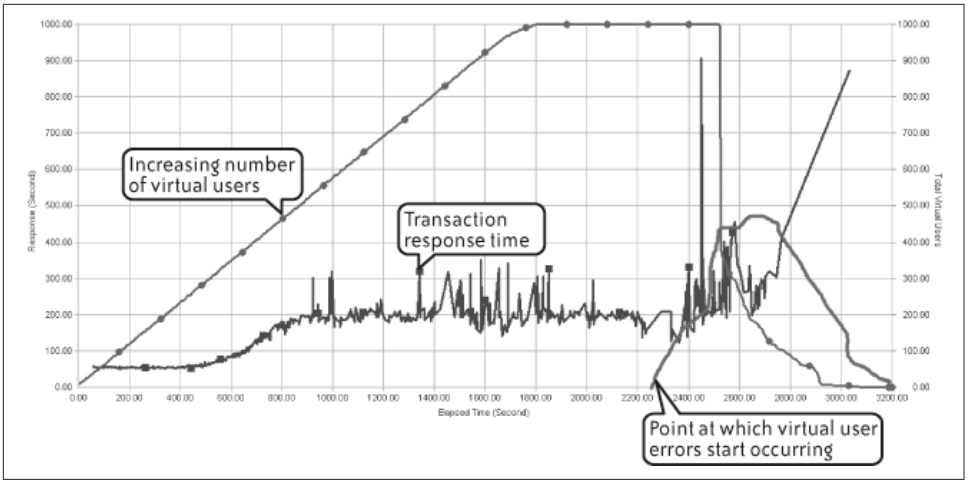


Figure 5-20. *Example of errors occurring during a performance test*

Baseline Data

The final output from a successful performance testing project should be baseline performance data that can be used when you are monitoring application performance after deployment. You now have the metrics available that allow you to set realistic performance SLAs for client, network, and server (CNS) monitoring of the application in the production environment. These metrics can form a key input to your information technology service management (ITSM) implementation, particularly with regard to end-user experience (EUE) monitoring (see Chapter 7).

Analysis Checklist

To help you adopt a consistent approach to analyzing the results of a performance test both in real time and after execution, here is another checklist. There's some overlap with the checklist from Chapter 4, but the focus here is on analysis rather than execution. For convenience, I have repeated this information as part of the quick-reference guides in Appendix B.

Pre-Test Tasks

- Make sure that you have configured the appropriate server, application server, and network KPIs. If you are planning to use installed agents instead of remote monitoring, make sure there will be no obstacles to installing and configuring the agent software on the servers.

- Make sure that you have decided on the final mix of performance tests to execute. As discussed in Chapter 4, this commonly includes pipe-clean tests, volume tests, and isolation tests of any errors found, followed by stress and soak tests.
- Make sure that you can access the application from your injectors! You'd be surprised how often a performance test has failed to start because of poor application connectivity. This can also be a problem during test execution: testers may be surprised to see a test that was running smoothly suddenly fail completely, only to find after much headscratching that the network team has decided to do some unscheduled housekeeping.
- If your performance testing tool provides the capability, set any thresholds for performance targets as part of your test configuration. This capability may simply count the number of times a threshold is breached during the test, and it may also be able to control the behavior of the performance test as a function of the number of threshold breaches that occur—for example, more than 10 breaches of a given threshold could terminate the test.
- If your performance testing tool provides the capability, configure autocorrelation between use-case response time, concurrent virtual users, and server or network KPI metrics. This extremely useful feature is often included in recent generations of performance testing software. Essentially, it will automatically look for (undesirable) changes in KPIs or response time in relation to increasing virtual user load or throughput. It's up to you to decide what constitutes *undesirable* by setting thresholds for the KPIs you are monitoring, although the tool should provide some guidance.
- If you are using third-party tools to provide some or all of your KPI monitoring, make sure that they are correctly configured *before* running any tests. Ideally, include them in your dress rehearsal and examine their output to make sure it's what you expect. I have been caught running long duration tests only to find that the KPI data collection was wrongly configured or even corrupted!
- You frequently need to integrate third-party data with the output of your performance testing tool. Some tools allow you to automatically import and correlate data from external sources. If you're not fortunate enough to have this option, then you'll need to come up with a mechanism to do it efficiently yourself. I tend to use MS Excel or even MS Visio, as both are good at manipulating data. But be aware that this can be a very time-consuming task.

Tasks During Test Execution

At this stage, your tool is doing the work. You need only periodically examine the performance of your load injectors to ensure that they are not becoming stressed. The dress rehearsal carried out before starting the test should have provided you with con-

fidence that enough injection capacity is available, but it's best not to make assumptions.

- Make sure you document every test that you execute. At a minimum, record the following information:
  - The name of the performance test execution file and the date and time of execution.
  - A brief description of what the test comprised. (This also goes into the test itself, if the performance testing tool allows it.)
  - If relevant, the name of the results file associated with the current test execution.
  - Any input data files associated with the performance test and which use cases they relate to.
  - A brief description of any problems that occurred during the test. It's easy to lose track of how many tests you've executed and what each test represented. If your performance testing tool allows you to annotate the test configuration with comments, make use of this facility to include whatever information will help you to easily identify the test run. Also make sure that you document which test result files relate to each execution of a particular performance test. Some performance testing tools store the results separately from other test assets, and the last thing you want when preparing your report is to wade through dozens of sets of data looking for the one you need. (I speak from experience on this one!) Finally, if you can store test assets on a project basis, this can greatly simplify the process of organizing and accessing test results.
- Things to look out for during execution include the following:

  The sudden appearance of errors

  This frequently indicates that some limit has been reached within the SUT. If your test is data-driven, it can also mean you've run out of data. It's worth determining whether the errors relate to a certain number of active virtual users. Some performance tools allow manual intervention to selectively reduce the number of active users for a troublesome use case. You may find that errors appear when, say, 51 users are active, but go away when you drop back to 50 users.

NOTE

Sudden errors can also indicate a problem with the operating system's default settings. I recall a project where the application mid-tier was deployed on a number of blade servers running Sun Solaris Unix. The performance tests persistently failed at a certain number of active users, although there was nothing to indicate a lack of capacity from the server KPI monitoring we configured. A search through system logfiles revealed that the problem was an operating system limit on the number of open file handles for a single user session. When we increased the limit from 250 to 500, the problem went away.

A sudden drop in throughput
: This is a classic sign of trouble, particularly with web applications where the virtual users wait for a response from the web server. If the problem is critical enough, the queue of waiting users will eventually exceed the time-out threshold for server responses and the test will exit. Don't immediately assume that the web server layer is the problem; it could just as easily be the application server or database tier. You may also find that the problem resolves itself when a certain number of users have dropped out of the test, identifying another capacity limitation in the hosting infrastructure. If your application is using links to external systems, check to ensure that none of these links is the cause of the problem.

An ongoing reduction in available server memory
: You would expect available memory to decrease as more and more virtual users become active, but if the decrease continues after all users are active, then you may have a memory leak. Application problems that hog memory should reveal themselves pretty quickly, but only a soak test can reveal more subtle problems with releasing memory. This is a particular problem with application servers, and it confirms the value of providing analysis down to the component and method level.

Panicked phone calls from infrastructure staff
: Seriously! I've been at testing engagements where the live system was accidentally caught up in the performance testing process. This is most common with web applications, where it's all too easy to target the wrong URL.

Post-Test Tasks

- Once a performance test has completed—whatever the outcome—make sure that you collect all relevant data for each test execution. It is easy to overlook important data, only to find it missing when you begin your analysis. Most performance tools collect this information automatically at the end of each test execution, but if you're relying on other third-party tools to provide monitoring data, then make sure you preserve the files you need.

- It's good practice to back up all testing resources (e.g., scripts, input data files, test results) onto a separate archive, because you never know when you may need to refer back to a particular test run.
- When producing your report, make sure that you map results to the performance targets that were set as part of the pre-test requirements capture phase. Meaningful analysis is possible only if you have a common frame of reference.

Summary

This chapter has served to demonstrate the sort of information provided by automated performance testing tools and how to go about effective root-cause analysis. In the next chapter, we discuss how the emergence of the mobile client has impacted application performance testing.

CHAPTER SIX

Performance Testing and the Mobile Client

> The mobile phone…is a tool for those whose professions require a fast response, such as doctors or plumbers.
>
> —Umberto Eco

THE RISE OF THE MOBILE DEVICE HAS BEEN RAPID AND REMARKABLE. THIS YEAR (2014) will see the mobile device become the greatest contributor to Internet traffic, and is unlikely to easily relinquish this position. As a highly disruptive technology, the mobile device has impacted most areas of IT, and performance testing is no exception. This impact is significant enough in my opinion that I have included this new chapter to provide help and guidance to those approaching mobile performance testing for the first time. I have also made reference to mobile technology where relevant in other chapters. I will first discuss the unique aspects of mobile device technology and how they impact your approach to performance testing, including design considerations, and then examine mobile performance testing challenges and provide a suggested approach.

What's Different About a Mobile Client?

Depending on your point of view, you could say, "not much" or "a whole lot." As you might be aware, there are four broad types of mobile client:

The mobile website
: This client type differs the least from traditional clients in that as far as the application is concerned it is really just another browser user. The technology used to build mobile sites can be a little specialized (i.e., HTML5), as can the on-device rendering, but it still uses HTTP or HTTPS as its primary protocol so can usually be recorded and scripted just like any other browser client. This is true for functional and performance testing. You might have to play around with the proxy setting

configuration in your chosen toolset, but this is about the only consideration over and above creating a scripted use case from a PC browser user. Early implementations of mobile sites were often hosted by SaaS vendors such as UsableNet; however, with the rapid growth of mobile skills in IT, those companies that choose to retain a mobile, or *m.*, site are generally moving the hosting in-house. There is a definite shift away from mobile sites to mobile applications, although both are likely to endure at least in the medium term.

The mobile application
: The mobile application is more of a different animal, although only in certain ways. It is first and foremost a fat client that communicates with the outside world via one or more APIs. Ironically, it has much in common with legacy application design, at least as far as the on-device application is concerned. Mobile applications have the advantage of eliminating the need to ensure cross-browser compliance, although they are, of course, still subject to cross-device and cross-OS constraints.

The hybrid mobile application
: As the name suggests, this type of application incorporates elements of mobile site and mobile application design. A typical use case would be to provide on-device browsing and shopping as an application and then seamlessly switch to a common browser-based checkout window, making it appear to the user that he is still within the mobile application. While certainly less common, hybrids provide a more complex testing model than an m. site or mobile application alone.

The m. site designed to look like a mobile application
: The least common commercial option, this client was born of a desire to avoid the Apple App Store. In reality this is a mobile site, but it's designed and structured to look like a mobile application. While this client type is really relevant only to iOS devices, there are some very good examples of noncommercial applications, but they are increasingly rare in the commercial space. If you wish to explore this topic further, I suggest reading *Building iPhone Apps with HTML, CSS, and JavaScript* by Jonathan Stark (O'Reilly).

Mobile Testing Automation

Tools to automate the testing of mobile devices lagged somewhat behind the appearance of the devices themselves. Initial test automation was limited to bespoke test harnesses and various forms of device emulation, which allowed for the testing of generic application functionality but nothing out-of-the-box in terms of on-device testing. The good news is that there are now a number of vendors that can provide on-device automation and monitoring solutions for all the principal OS platforms. This includes

on-premise tooling and SaaS offerings. I have included a list of current tool vendors in Appendix C.

Mobile Design Considerations

As you may know, there are a number of development IDEs available for mobile:

- BlackBerry
- iOS for Apple devices
- Android
- Windows Phone for Windows Phone devices

Regardless of the tech stack, good mobile application (and mobile site) design is key to delivering a good end-user experience. This is no different from any other kind of software development, although there are some unique considerations:

Power consumption
: Mobile devices are battery-powered devices. (I haven't yet seen a solar-powered device, although one may exist somewhere.) This imposes an additional requirement on mobile application designers to ensure that their application does not make excessive power demands on the device. The logical way to test this would be to automate functionality on-device and run it for an extended period of time using a fully charged battery without any other applications running. I haven't personally yet had to do this; however, it seems straightforward enough to achieve. Another option to explore would be to create a leaderboard of mobile applications and their relative power consumption on a range of devices. While this is probably more interesting than useful, it could be the next big startup success!

The cellular WiFi connection
: Mobile devices share the cellular network with mobile phones, although phone-only devices are becoming increasingly rare. Interestingly, with the proliferation of free metropolitan WiFi, a lot of urban mobile comms doesn't actually use the cellular network anymore. Cellular-only traffic is pretty much limited to calls and text messages unless you are using Skype or similar VoIP phone technology. Cellular and WiFi impose certain characteristics on device communication, but in performance testing terms this really comes down to taking into account the effect of moderate to high latency, which is always present, and the reliability of connection, which leads nicely into the next point.

Asynchronous design
: Mobile devices are the epitome of convenience: you can browse or shop at any time in any location where you can get a cellular or wireless connection. This fact alone dictates that mobile applications should be architected to deal as seamlessly

as possible with intermittent or poor connections to external services. A big part of achieving this is baking in as much async functionality as you can so that the user is minimally impacted by a sudden loss of connection. This has less to do with achieving a quick response time and more to do with ensuring as much as possible that the end user retains the perception of good performance even if the device is momentarily offline.

Mobile Testing Considerations

Moving on to the actual testing of mobile devices, we face another set of challenges:

Proliferation of devices
: Mainly as a result of mobile's popularity, the number of different devices now has reached the many hundreds. If you add in different browsers and OS versions, you have thousands of different client combinations—basically a cross-browser, cross-device, cross-OS nightmare. This makes ensuring appropriate test coverage, and by extension mobile application and m. site device compatibility, a very challenging task indeed. You have to focus on the top device (and browser for mobile site) combinations, using these as the basis for functional and performance testing. You can deal with undesirable on-device functional and performance defects for other combinations by exception and by using data provided by an appropriate end user experience (EUE) tool deployment (as we'll discuss in the next chapter).

API testing
: For mobile applications that communicate using external APIs (which is the vast majority), the volume performance testing model is straightforward. In order to generate volume load, you simply have to understand the API calls that are part of each selected use case and then craft performance test scripts accordingly. Like any API-based testing, this process relies on the relevant API calls being appropriately documented and the API vendor (if there is one) being prepared to provide additional technical details should they be required. If this is not the case, then you will have to reverse-engineer the API calls, which has no guarantee of success—particularly where the protocol used is proprietary and/or encrypted.

Mobile Test Design

Incorporating mobile devices into a performance testing scenario design depends on whether you are talking about a mobile *m.* site, a mobile application, or indeed both types of client. It also depends on whether you are concerned about measuring on-device performance or just about having a representative percentage of mobile users as part of your volume load requirement. Let's look at each case in turn.

On-Device Performance Not in Scope

In my view, on-device performance really should be in scope! However, that is commonly not the case, particularly when you are testing a mobile site rather than a mobile application.

Mobile site

Test design should incorporate a realistic percentage of mobile browser users representing the most commonly used devices and browsers. You can usually determine these using data provided by Google Analytics and similar business services. The device and browser type can be set in your scripts using HTTP header content such as `User agent`, or this capability may be provided by your performance tooling. This means that you can configure virtual users to represent whatever combination of devices or browser types you require.

Mobile application

As already mentioned, volume testing of mobile application users can really only be carried out using scripts incorporating API calls made by the on-device application. Therefore, they are more abstract in terms of device type unless this information is passed as part of the API calls. As with mobile site users, you should include a representative percentage to reflect likely mobile application usage during volume performance test scenarios. Figure 6-1 demonstrates the sort of mobile data available from Google Analytics.

	Mobile Device Info	Acquisition			Behaviour		
		Sessions ↓	% New Sessions	New Users	Bounce Rate	Pages / Session	Avg. Session Duration
		730,196 % of Total: 56.94% (1,282,467)	46.65% Site Avg: 52.53% (-11.19%)	340,648 % of Total: 50.56% (673,700)	44.62% Site Avg: 43.12% (3.47%)	4.90 Site Avg: 5.53 (-11.41%)	00:03:06 Site Avg: 00:03:21 (-7.65%)
	1. Apple iPhone	**337,586** (46.23%)	44.05%	148,711 (43.66%)	47.31%	3.73	00:02:30
	2. Apple iPad	**285,262** (39.07%)	47.87%	136,567 (40.09%)	40.40%	6.42	00:03:47
	3. Samsung GT-I9505 Galaxy S IV	**15,756** (2.16%)	47.95%	7,555 (2.22%)	47.18%	4.46	00:02:55
	4. Samsung GT-I9300 Galaxy S III	**11,161** (1.53%)	44.57%	4,975 (1.46%)	47.24%	4.32	00:03:11
	5. (not set)	**4,459** (0.61%)	67.86%	3,026 (0.89%)	51.38%	3.65	00:02:34
	6. Google Nexus 7	**3,682** (0.50%)	64.53%	2,376 (0.70%)	57.20%	3.96	00:02:23
	7. Samsung GT-I9195 Galaxy S4 Mini	**3,474** (0.48%)	52.19%	1,813 (0.53%)	45.60%	4.27	00:02:25
	8. Samsung GT-I8190N Galaxy S III Mini	**3,275** (0.45%)	48.46%	1,587 (0.47%)	47.88%	4.26	00:02:58
	9. Samsung SM-N9005 Galaxy Note 3	**3,259** (0.45%)	44.71%	1,457 (0.43%)	45.57%	5.04	00:03:37
	10. Amazon KFTT Kindle Fire HD 7	**2,945** (0.40%)	51.17%	1,507 (0.44%)	41.97%	7.15	00:04:04

Figure 6-1. *Google Analytics mobile device analysis*

On-Device Performance Testing Is in Scope

Again, it's certainly my recommendation for on-device performance testing to be in scope, but it's often challenging to implement depending on the approach taken. It is clearly impractical to incorporate the automation of hundreds of physical devices as part of a performance test scenario. In my experience a workable alternative is to combine a small number of physical devices, appropriately automated with scripts working at the API or HTTP(S) level, that represent the volume load. The steps to achieve this are the same for mobile sites and mobile applications:

1. Identify the use cases and test data requirements.
2. Create API level and on-device scripts. This may be possible using the same toolset, but assume that you will require different scripting tools.
3. Baseline the response-time performance of the on-device scripts for a single user with no background load in the following sequence:
   - Device only. Stub external API calls as appropriate.
   - Device with API active.
4. Create a volume performance test scenario incorporating where possible the on-device automation and API-level scripts.
5. Execute only the on-device scripts.
6. Finally execute the API-level scripts concurrently with the on-device scripts.

As API level volume increases, you can observe and compare the effect on the on-device performance compared to the baseline response times. Measuring on-device performance for applications was initially a matter of making use of whatever performance-related metrics could be provided by the IDE and OS. Mobile sites were and still are easier to measure, as you can make use of many techniques and tooling common to capturing browser performance. I remain a fan of instrumenting application code to expose a performance-related API that can integrate with other performance monitoring tooling such as APM. This is especially relevant to the mobile application, where discrete subsets of functionality have historically been challenging to accurately time. As long as this is carefully implemented, there is no reason why building in *code-hook* should create an unacceptable resource overhead. It also provides the most flexibility in the choice of performance metrics to expose. As an alternative, the major APM tool vendors and others now inevitably offer monitoring solutions tailored for the mobile device. I have included appropriate details in Appendix C.

Summary

Now that we've discussed the challenges of mobile client performance testing, the next chapter looks at how monitoring the end-user experience is closely aligned with application performance requirements.

CHAPTER SEVEN

End-User Experience Monitoring and Performance

Now you see it—now you don't.

—Anonymous

HAVING DISCUSSED A BEST-PRACTICE APPROACH TO WHAT COULD BE CONSID-ered static performance testing in the preceding chapters, I want to now discuss the importance of measuring and understanding the end-user experience. To aid in this process I have called on my friend and colleague, Larry Haig, to provide much of this chapter's content. Larry's many years of experience with customer engagements, both while working for one of the major performance vendors and as an independent consultant, have made him one of the industry's leading lights in end-user experience monitoring.

Traditional performance testing is designed to compare a system with itself, using a variety of infrastructure and application KPIs (as discussed earlier in this book). If effectively designed and executed, this will provide a degree of performance assurance. However, crucially, this approach gives no absolute extrapolation to the experience of end users of the tested application. This chapter and the next address the subject of external monitoring, both in its wider aspects and as an extension to performance testing. They can be read as a standalone guide to effective end-user experience management, looking at the wider situation of ongoing application performance assurance, as well as an exploration of end-user satisfaction as a component of performance testing (the core subject of this book).

There is no shortage of external monitoring products available, and I have listed some of the more well-known in Appendix C. License costs vary from nothing at all to hundreds of thousands of dollars or more over a calendar year. All seek to answer the question, *how well is my site performing outside my data center*?, and all will provide you with an answer—of a sort. How relevant the answer is depends upon the nature of the question, and the stakes are high particularly for ecommerce: £15.9 billion was spent

online during Christmas 2012 in the United Kingdom alone (retailresearch.org), and the research shows that poor performance is a key determinant of site abandonment.

End-user performance, not the number of green lights in the data center, is what determines whether your investment in application delivery is a success. Further, environmental simplicity is a distant memory, at least as far as major corporate applications are concerned. Effective strategies are required for you to understand external application performance and obtain timely, relevant data to support effective intervention. This chapter aims to supply those, and to help performance professionals design appropriate monitors for all types of content, whether native HTML, multimedia, SMS, affiliate-based content, or anything else. The importance of understanding key user behaviors and devices, as well as being aware of device trends, will be emphasized.

Ultimately, as this book has hopefully demonstrated, performance is not an absolute. Key performance metrics need to be determined relative to user expectation—which is increasing all the time. A recent review of historic benchmark performance trends by my own company, Intechnica, showed an average increase in landing page performance across tested sectors of one-third. This was based on retail, media, and banking website performance over an average of the three years from September 2010 to October 2013. Customer expectation follows this trend, so standing still is not an option. You should bear this important factor in mind for all web applications, although it is especially true of ecommerce sites, given their direct revenue implications.

This chapter is designed to ensure that the crucial importance of end-user delivery is not forgotten as a component of application performance. It provides a bird's-eye view of external monitoring, from first principles to some more detailed considerations, with (hopefully) useful further reading suggested at the end. Given the prominence of Internet browser-based applications, they are the primary area of consideration, although the underlying principles apply equally to nonweb situations. Some thoughts about relevant tools and techniques are also included.

What Is External Monitoring?

A good working definition of external monitoring is simply *the use of tools and techniques to provide and interpret IT application response metrics from outside the edge servers of the core application delivery infrastructure* (that is, the data center). At first glance, the simplicity of this definition is immediately compromised by the inherent complexity of most modern web-based applications.

As an example, my company recently examined 10 major UK ecommerce sites (see Table 7-1). These contained between them over 70 distinct third-party affiliates, even after we excluded content delivery network (CDN) providers that were directly associated with delivery performance. Further, core site content is often distributed. Examples include the external delivery of static content by a specialist provider (e.g., Adobe

Scene 7), the use of cloud-based applications (often for reasons of high scalability), or the deployment of high-redundancy distributed environments to provide disaster recovery capability.

Table 7-1. *Affiliate load range of third-party inclusions from 10 major UK ecommerce sites*

Site	# Non–performance-related affiliates (hosts)	End-user performance overhead (total response time, search ATB transaction, in seconds)
Very	43	18.7
Tesco	1	0.2
Sainsbury	27	11.6
Ralph Lauren	11	1.0
Next	1	0.5
New Look	53	23.4
Marks and Spencer	27	10.3
John Lewis	19	3.2
Debenhams	2	15.9
ASOS	49	10.9

Why Monitor Externally?

The power and sophistication of application and infrastructure monitoring has increased greatly in recent years, much as a result of the wholesale move to distributed web deployment of core applications. One argument is that such on-premise monitoring is sufficient for effective risk management of application performance; however, external monitoring has a vital role to play.

Consider modern developments—service-oriented architectures, distributed cloud-based hosting—together with the plethora of third-party components included in many sites that must be considered performance related (CDNs, application delivery controllers) and revenue centric (personalization, ad servers, web analytics). Anything that is provided from or includes delivery components outside the application's edge servers requires management, and external monitoring enables it.

The various types of external monitoring will be considered in "External Monitoring Categories" on page 130, but the key benefits of an outside-in testing approach can be summarized as follows:

- Impact analysis: external testing, particularly with an end-user dimension, provides visibility as to the business relevance of a particular issue—that is, its likely revenue impact. Such insight enables operations teams to effectively prioritize performance interventions.
- Predictive understanding: synthetic external monitoring enables testing of application performance in the absence of baseline traffic—for example, new sites (or prelaunch upgrades), new markets, and new products.
- Related to the preceding point is the ability to obtain proactive data on the performance of particular functions or processes. This is particularly valuable when previous difficulties have led to significant reductions in relevant visitor traffic (e.g., failure with a major browser version or wireless carrier).
- Careful test design enables objective (rather than inferential) data to be obtained —at the transaction, page, individual object, and subobject level.
- Active monitoring permits contextual understanding and goal setting (e.g., competitive benchmarking).
- Finally, this testing approach helps you obtain a detailed understanding of all external factors involved in delivery to end users; for example:
  - —Third-party service-level management
  - —Internet service provider (ISP) peerage effects
  - —Validation/assurance of key user performance

A final point regarding end-user understanding: it is sometimes argued that there is no point in understanding performance in true end-user conditions (wireless mobile connectivity, limiting conditions of bandwidth, ISP peerage, consumer PC system constraints, and similar). The logical extension of that thinking is that testing should be undertaken only in completely stable, clean-room conditions, as otherwise you may not see the forest for the trees.

There is certainly an important place for best-case testing, and test conditions should always be consistent and understood to enable effective results interpretation. However, the important point is that your application lives in the real world, which is by its nature heterogeneous. Understanding whether an issue affects a particular class of user enables you to make appropriate interventions.

In some cases, these interventions can be as simple as putting a text message on the site advising particular users that their performance will be compromised (or instituting some form of elegant degradation of functionality).

Even if issues are outside the ability of operations teams to address directly, access to objective data will enable them to be addressed through their third party (i.e., ISP, CDN provider).

In summary, the essential function of external monitoring is to act as a "canary in a coal mine" (Figure 7-1). Nineteenth-century miners needed to know when they were facing a problem—in their case, odorless (and deadly) carbon monoxide gas—the canary provided the understanding that: (a) a problem existed, and (b) the miners needed to exit stage right!

Figure 7-1. *External monitoring can be likened to the proverbial canary in a coal mine*

External monitoring will not, except in certain specific edge cases, isolate the root cause of an issue, but if used appropriately it will provide impact-based understanding and isolate the area in which the issue exists. A single relevant example: applications exhibiting poor performance at high-traffic periods of the business demand cycle are strong candidates for diagnostic performance load testing. In short, external monitoring is essential to relative, contextual management of application performance.

The increasingly plural nature of modern web applications and components—whether mobile devices, server push technologies, service worker delivery, adaptive elements, or similar—are likely to present challenges to effective external monitoring in the future, making a new "best practice" an increasingly important and evolving requirement. However, this chapter provides some pointers on approaches to use in the current situation.

Many tools exist to provide end-user performance visibility. However, regardless of the competing claims of the individual vendors, there are some fundamental differences that should be understood. Any single type of testing has limitations as well as advantages, and one size most certainly does not fit all.

External Monitoring Categories

In the parlance of external analysis, all monitoring may be grouped into one of two core types:

Active
: Also known as synthetic monitoring, active monitoring is effectively prescheduled, repetitive automated testing from known testing nodes (or node categories such as "Chicago-based end users" in some cases).

  Active monitoring is delivered by tooling that replays scripted use cases much in the manner of performance test automation. The key difference is that replay is normally from the perspective of a single virtual user per device deployment and usually involves a complete UI or browser session—i.e., a headed replay.

  This tooling typically presents as a web-based management console to manage the deployment of multiple remote agents, a scripting component to record and create the use cases for replay, and an analysis dashboard to monitor and interpret results. It is not uncommon for a range of integrations to be available, and this capability is now part of many application performance management (APM) toolsets. You can find a list of leading vendors in Appendix C.

Passive
: Also known as real-user monitoring (RUM), user experience monitoring (UEM), and some other similar acronyms, passive monitoring relies on analysis of visitor traffic from the operation of code snippets residing either in the headers of instrumented pages, or in certain cases, dynamically injected into client browsers by the host application server.

Active and passive monitoring vary in sophistication depending upon the tooling employed.

Active Monitoring

The most basic (and original) active monitoring was simply to ping a website and listen and time the return of a test byte, and this still constitutes the entry level of basic availability monitoring. However, while this approach is inexpensive (or free), the actionable visibility it provides is extremely limited.

In the first place, these tools are only testing the availability of the base HTML object; they offer no visibility of the other page components that make up a full download (and therefore successful user experience), let alone those of complex multistep transactions. A large white space with a red cross in it where your banner image should be is unlikely to inspire confidence in your brand.

The second key limitation is linked to the first. Such tools can rapidly tell you that your site is down, but what do you do then? Without effective object-level information, rapid diagnosis and issue resolution is extremely difficult. Such a basic level of analysis is far less timely in any case than advanced predictive diagnostics, which will often alert you to an impending failure before it occurs (which should be a goal of your performance monitoring strategy).

It is important to emphasize that however it is obtained, some understanding of application availability is essential. This understanding cannot be obtained from other types of passive analysis, which are, by definition, visitor analyses. All of the potential users who fail to reach a site due to the dozens of potential external issues (e.g., DNS resolution failure, ISP peerage) are by definition *not* visitors. Most professional active monitoring tooling today provides for full object-level visibility, combined with the ability to script transactions through the site. Table 7-2 summarizes the test perspectives.

Table 7-2. *Test perspectives*

Test origin	Purpose
Tier 1 data center, industrial-capacity test node	"Clean room" testing, (theoretically) free of bandwidth or test system constraints. For trend/iterative goal management, competitor comparison, third-party SLA management (excepting CDNs).
End user (PC, mobile device)	"Dirty" but real-world data. Provides insights into performance or limiting conditions (e.g., of bandwidth). For CDN, performance assurance and quality of service monitoring in existing and new markets.
Private peer	Testing from known locations (e.g., specific high-value customers or departments/locations within a corporate organization).
Site visitor performance analysis	Also known as real user or user experience monitoring. Also known as EUM or RUM. Records and analyzes the performance (possibly together with other metrics, e.g., transaction abandonment) of all successful visitors to a site.

Test origin	Purpose
Native mobile application	Captures performance and associated data (e.g., crash metrics) of the users of native mobile applications. Typically cross-device but operating-system (iOS, Android) specific.

Figure 7-2 illustrates the use of multiple active test perspectives to provide a prescheduled heartbeat test. These results, particularly when combined with the visitor base metrics listed in the next section, can provide a comprehensive model of end-user response.

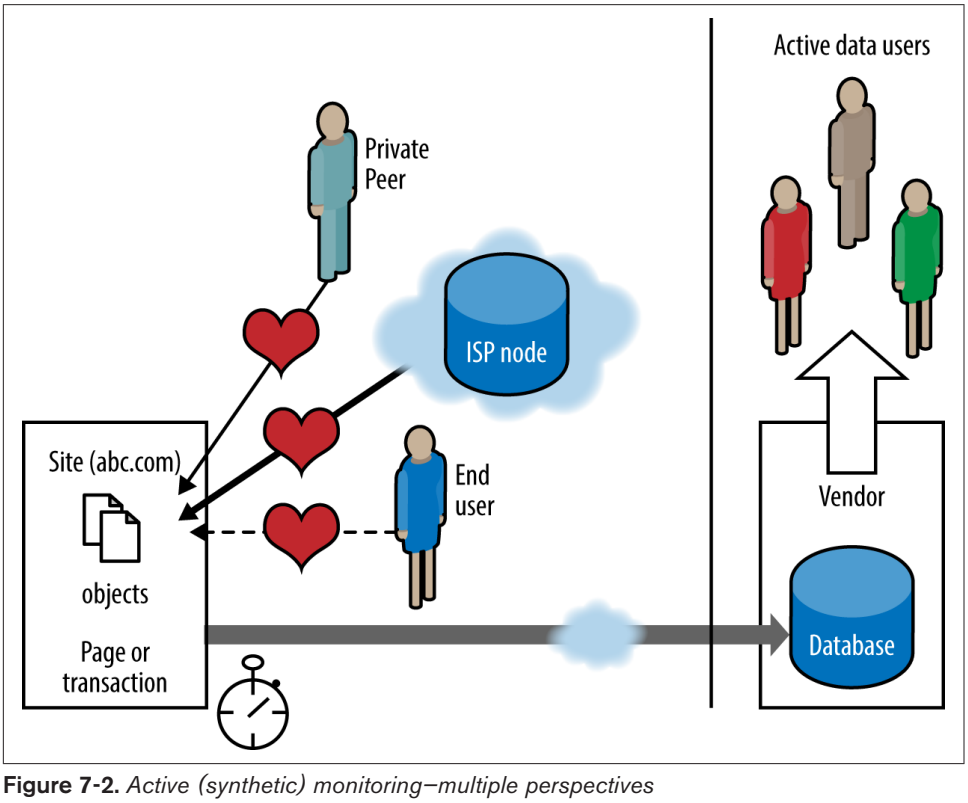


Figure 7-2. *Active (synthetic) monitoring—multiple perspectives*

Output Metrics

Active monitoring includes the following key metrics:

Availability
: The key metric from active monitoring. Because such monitoring is typically prescheduled, the "fate" of every active test in terms of success (access to the test site)

or failure is known. In contrast, passive monitoring (RUM) analyzes the response of successful visits to the site only.

Total response time
: It is important to understand what the page load or response time reported by active monitoring systems actually represents. Such tooling operates by effectively listening to the network traffic associated with a web page request. Thus it records the total time for page delivery—that is, the elapsed time between the receipt of the HTTP request and the delivery of the final byte of data associated with the page. This response time and the underlying component-level data is extremely valuable in understanding comparative performance, and in the isolation of some categories of application issues. Although page response times will be longer in end-user test conditions that make use of consumer-grade devices over tertiary ISP or wireless carrier links, this metric differs from the browser fill time experienced by an actual site user. This latter metric, also known as the *perceived render time* or some variant thereof, is recorded by some of the more sophisticated passive monitoring tools. Figure 7-3 illustrates the difference.

ISP Testing Best Practices

When configuring ISP-based testing, you should consider the following points:

Connectivity
: This class of testing is designed to be best case. It is crucial to ensure that no constraints or variations in the test environment exist, either of connectivity or test node. The major vendors of this type of monitoring will either deploy multiple nodes within tier-one ISP data centers close to the major confluences of the Web, or locate testing within LINX exchanges. The latter approach ensures that a range of ISPs are selected in successive tests, while the former enables specific peerage issues to be screened (at least among the ISPs for whom test nodes are available). Avoid tooling based in tertiary data centers where high levels of minimum bandwidth cannot be assured.

Triangulation
: Undertake testing from across a variety of ISPs and/or test nodes. Doing so ensures that any performance disparities reported are effectively isolated to the test site concerned (assuming that the issue is detected across multiple nodes/tests), rather than potentially being an ISP- or test-node-specific issue—which could be the case if testing was undertaken from a single node. Ideally, you should use three or four nodes, distributing testing evenly across each.

Location
: When selecting vendors, and in designing tests, test from multiple locations with a similar latency. It is not essential for all test nodes to be from the same country as

the target, but take care to ensure that, if not, they are in regional locations on the primary backbone of the Internet.

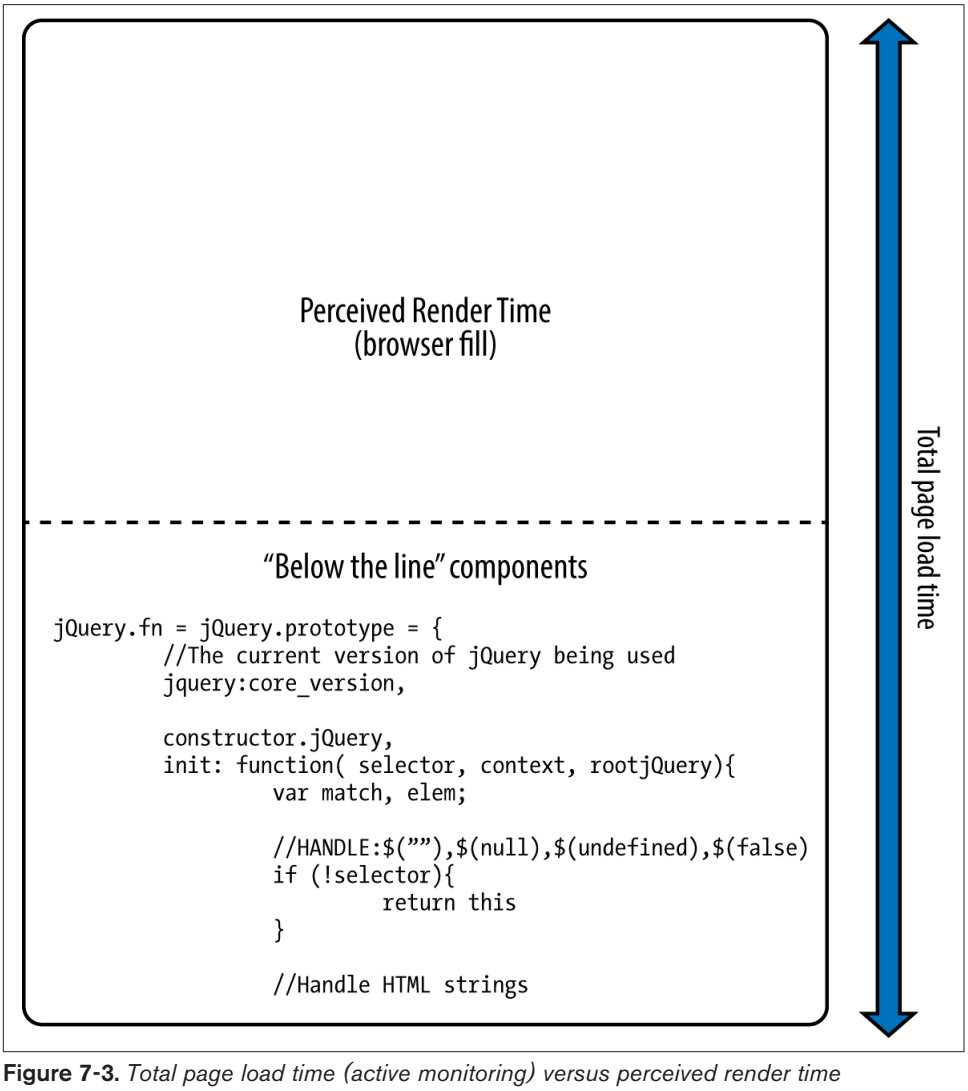


Figure 7-3. *Total page load time (active monitoring) versus perceived render time*

Frequency

With the advent of passive monitoring (RUM), which by its nature is (or should be) continuous, the requirement for high-frequency active monitoring has declined. A distinction should be made between ping (availability, aka up/down testing) where test frequencies tend to be very high and strategic, and full object-level or use-case monitoring. The key is to have recent data and trend information available to drill into when a potential issue is flagged. Thus, a heartbeat test fre-

quency of, say, one or two tests per hour from four nodes giving a nominal frequency of one test every 7.5 minutes should suffice. Test frequency can always be increased during active issue investigation if required.

Agent type
: As a general rule, the primary browser agent you use should reflect the predominant browser in the country being tested (or the browser/version with the highest penetration among your customer base if different). Browser usage changes significantly over time, and some entrenched patterns of usage (such as that of Internet Explorer 6 in China, for example) can be expected to change rapidly following withdrawal of support and malware protection by the vendor.

Analysis
: Automated analysis is a useful feature of some tooling. Even if present, it provides no more than a shortcut to some of the more common factors to consider. A structured issue isolation model should be adopted and followed.

Synthetic End-User Testing Best Practices

Synthetic end-user testing differs from backbone in that it is essentially about quality of service (errors and performance) rather than ideal, comparative external tests. It is not an alternative to active ISP-based and passive (RUM) testing, but rather a supplement. Some important differences include the following:

- Public peers have the advantage of multiple tertiary ISPs. Consider black/white listing and PC specifications.
- Private peers are useful for testing from known locations. They can be particularly useful in testing in closed intranet application situations, or from specific partner organizations, for example. Private peer-agent deployment has also been successfully used for scheduled testing from specific high-net-worth customers (e.g., in the gaming industry).
- Note that end-user response data is always inconsistent—but so is the real world. Artificially throttled testing from data center test nodes is cleaner, but does not represent the heterogeneity of actual usage conditions. It also tests from very few, fixed locations with constant conditions of ISP connectivity.
- Connectivity—mapping to typical/desired users and use cases—helps you to understand site performance in limiting conditions. It is particularly important for large-footprint sites or those delivered into regions with poor Internet infrastructure.
- Mode, or wireless connectivity, is essential to understand the performance of mobile users, both from a device and network perspective. Hardwired ISP testing of mobile users should *never* be used other than as a reference point, because the

results cannot be extrapolated to the real world. Synthetic testing of mobile end users has limitations, however, as it typically occurs from fixed test nodes. This has the advantage of more replicable testing in terms of signal strength but may suffer from saturation of the local cellular mast, which introduces distortion into the results. Triangulation of results (as with backbone tests) is particularly important, as is careful attention to the connection speed: 3G has a maximum theoretical bandwidth of more than 6 Mbps, but connection speeds of less than 100 Kbps are not uncommon in remote areas and situations of high traffic.

- Consider the browser agent used, as it is not usually possible to read across backbone and last-mile tests measured using different browsers (particularly if one of them is IE).
- Peer size is an important factor. Due to the inherent variability of testing from end-user machines, you must test from sufficiently large clusters to enable average smoothing of the results. You must carefully consider the peer results selected for object-level analysis, and engage in progressive "pruning" by blacklisting any specific test peers that consistently produce erroneous results. Take care to operate blacklisting using rational exclusion criteria in addition to poor performance; otherwise, you may be actively removing evidence of a fault that you should be dealing with. If testing from public end-user machines, adopt a minimum peer cluster size of 7 to 10 nodes as a best practice to enable meaningful baselining. Peer groups should be selected from machines and connectivity bandwidths of similar characteristics. Results should be further filtered where possible to remove outliers due to rogue low (or high) connectivity tests.

All these caveats can lead you to think that scheduled end-user monitoring is of no value in a passive (RUM)-based world. Although continuous scheduled end-user monitoring may be of limited utility (for reasons, among others, of cost), this class of monitoring can prove invaluable in isolating or excluding unusual issues (e.g., tertiary ISP peerage, DNS resolution issues, CDN assurance). The ability to study object-level data from end users and error codes provides a useful view of quality of service to end users (see Figure 7-4), particularly in areas where base traffic is low (e.g., new markets).

Passive Monitoring

Passive monitoring is typically based on the capture of performance metrics from end-user browsers, although custom instrumentation is possible for any client type. Unlike active monitoring, which is essentially proactive in nature, passive monitoring requires visitor traffic, as you can see in Figure 7-5. It is also known as real-user monitoring (RUM), user-experience monitoring (EUM), and other similar names. I use *passive (RUM)* going forward.

Figure 7-4. *Quality of service–fatal errors generated by UK ecommerce site landing page (30-day test window)*

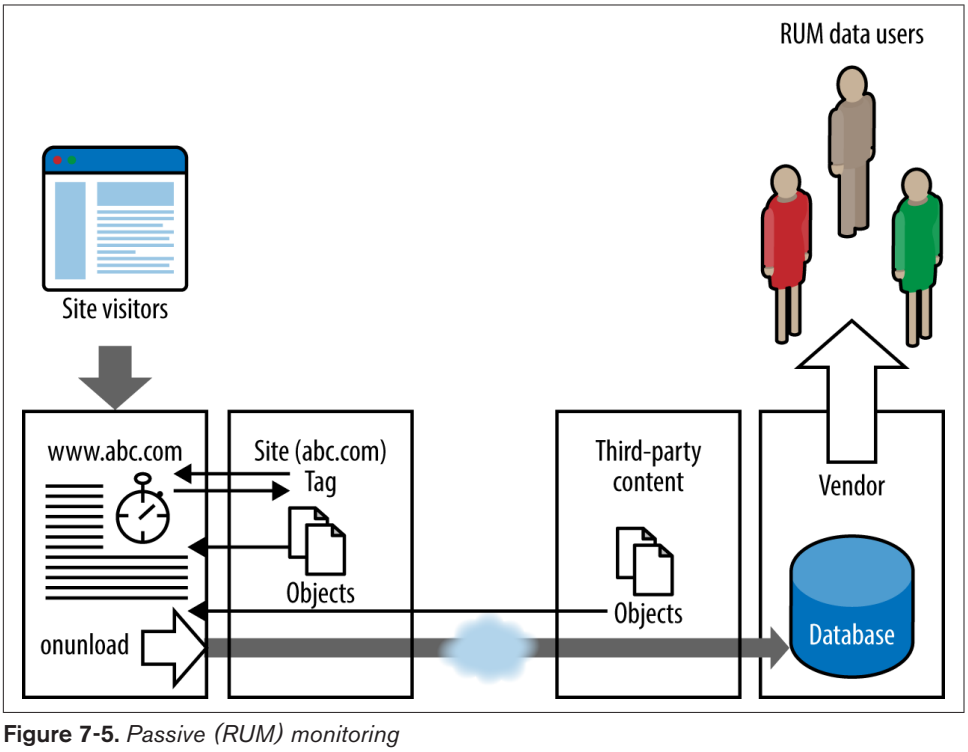


Figure 7-5. *Passive (RUM) monitoring*

How Passive Monitoring Works

The most common technique for browser clients is to place a JavaScript tag or beacon in web page headers designed to detect user traffic. In most cases, the tag permanently resides on instrumented pages, placed via a simple cut-and-paste or a global *include* statement. EUE toolset vendors are increasingly offering the option of dynamic injection of JavaScript into the visitor browser by the site application servers during the page request.

The code executes, writing cookies as needed to capture browser information, navigation metrics, and potentially other data such as connection speed or ISP. After the onunload step, data is passed back using a GET request via HTTP or HTTPS to the vendor database. This data is then made available via appropriate dashboards in the product client.

Passive (RUM) tooling is typically provided either as a standalone product or as part of an integrated end-to-end visibility extension to APM tools. While the mode of action is similar, APM tooling has the advantage of associating browser-side performance metrics with those of the delivery infrastructure, whether application (often to code method level) or hardware. A rapid growth in recent years has followed the publication (by W3C) and adoption by many browser vendors of standard navigation and resource metrics. Although broadly similar from a basic technical standpoint, passive (RUM) tools do differ fairly substantially. Some key considerations are listed here:

Sophistication/coverage
: As previously mentioned, many passive (RUM) products are based on the standard W3C navigation metrics, which means they are not supported in all browsers, primarily older versions and Safari. In certain cases, basic performance data is collected from these metrics to supplement the core metrics. Key aspects of sophistication include the following:

  - Ability to record user journeys: Less evolved products act at the individual page level.
  - Ability to capture and report individual session-level data: This includes reporting on business-relevant metrics, such as transaction abandonment and shopping cart conversion by different categories of user.
  - Detailed reporting: This includes reporting on bounce rate, stickiness (time on site), abandonment, and similar.
  - Ability to record above-the-line performance: Browser fill or perceived render time.

This metric is rarely estimated by active monitoring tools (the only one of which I am aware is WebPagetest). Therefore, passive (RUM) tooling supporting this metric provides a useful additional perspective into end-user satisfaction.

Real-time reporting

Tools vary in two principal ways with regard to data handling:

- How long captured data is stored. As with other monitoring, the problem for vendors storing customer data is that they rapidly become data storage rather than monitoring companies. However, the ability to view trend data over extended periods is extremely useful, so individual vendor strategies to manage that requirement are relevant. This problem is exacerbated if object-level metrics are captured.
- The frequency of customer data updates. This can vary from 24 hours to less than 5 minutes. Near-real-time updates are relevant to active operations management, while daily information has limited historic value only.

All traffic or traffic sampling

Because passive (RUM) data is inferential in nature, it is important to capture all visitor traffic rather than a sample. Some tooling offers the option of user-defined sampling, often to reduce license costs. This is unlikely to be good practice except possibly in the case of extremely high-traffic sites. Within Europe, this situation is exacerbated by EU legislation enabling individual users to opt for do-not-send headers, which restrict the transmission of tag-based data.

API access

Passive (RUM) tooling will always provide some form of output charting or graphing. You can derive additional value by integrating passive (RUM) data with the outputs from other tooling. This is particularly true for those products that do not report on session-level data, such as conversion and abandonment rates. In such cases, it may be advantageous to combine such data from web analytics with passive (RUM)-based performance metrics.

Page or object level

Although, theoretically, all products could be extended to capture object-level rather than page-delivery metrics, this is not the case with every passive (RUM) offering.

User event capture

This is the ability to record the time between two events (e.g., mouse clicks). Such subpage instrumentation is valuable in supporting design and development decisions.

Extensibility
: This is the ability to capture and integrate nonperformance user data. Examples include associating user login details with session performance, and collecting details of the originating application or database server.

Reporting
: This metric refers to the extent, type, and ability to customize built-in reporting.

The strength of passive monitoring (RUM) lies in the breadth of its reach in terms of browsers/versions, geography, ISP, and similar. High-level patterns of performance—for example, regional CDN performance—and national performance delivery deficits in challenging markets can be identified. At a more granular level, patterns of performance between browsers or ISPs can readily be determined, which in the case of smaller players would be unlikely through active monitoring.

Passive monitoring (RUM) has two primary weaknesses:

The inferential nature of the results
: Passive (RUM) data, being based on visitors, is essentially reactive. It does not support the proactive testing of particular markets, devices, or functionality. Different patterns of usage in different markets could be a reflection of cultural differences or of functional/nonfunctional issues, requiring active monitoring to accurately determine. By definition, prospective site visitors who fail to reach the site are not visitors, so the concept (and metric) of availability is not valid.

The absence of object data (in the majority of cases)
: This is extremely limiting if you are seeking to isolate the cause of a performance issue.

Pros and Cons of Active Versus Passive Monitoring

This section summarizes the key advantages and disadvantages of active (synthetic) and passive (RUM) external monitoring.

Active Pros

- Enables you to target monitoring to issues, browsers, location, conditions, content, services (e.g., multimedia, payment service provider)
- Offers round-the-clock (if required) visibility in periods of low base traffic
- Provides consistent comparative data (internal or competitive)
- Allows you to monitor any public site (benchmarking)
- Is easily repeatable
- Can capture host-level data

- Allows full-object capture and single-object testing
- Enables availability testing

Active Cons

- Is not real world
- Provides only a subset of true conditions
- Can be conducted only with limited frequency (depending upon a realistic budget)

Passive Pros

- Provides true visitor understanding
- Allows discovery of origin-related issues
- Provides an aggregate understanding of application components (CDN performance, for example)

Passive Cons

- Is potentially limited in new markets because visibility depends on visitor traffic
- Is inferential—that is, not based on proactive test design
- Can be counterfactual (e.g., poor quality of service producing low regional traffic volumes)
- Offers limited understanding in that results are not (usually) object based, can be limited by browser coverage, and can be skewed by cookie blocking from end users

Tooling for External Monitoring of Internet Applications

Tooling choices for external monitoring are numerous, and need to be considered from multiple perspectives. These include the following:

- Technology to be monitored: native mobile, RIA, server push
- User characteristics: location, browsers, mobile devices
- Application features and components: third-party content, multimedia, SMS

- Performance interventions: for example, CDNs, ADC (ability to filter, A/B test as required)
- In-house skills and resources: outsourcing (tooling, agency) versus self-service
- Application development strategy and goals
- Performance management strategy: in-house or outsourced
- Realistic budget, taking account of both opportunities and risks
- Any givens with regard to technology or tooling

As reiterated during this chapter, we would strongly recommend considering tooling holistically, both across the DevOps life cycle (code profiling, continuous integration) and in the context of an integrated data management approach. This approach, for example, could seek to deliver a combination of tooling with minimal products and feature overlap to support, among others:

- Performance testing
- External monitoring
- Application performance management (all tiers, including mainframe if present)
- Network management
- Database support
- Runbook automation
- Reporting and dashboarding

The ability to check many boxes is usually the prerogative of the larger vendors. However, given that much extensive tooling is obtained by successive acquisition, take care to ensure that multiple products are integrated in practice, not just at brand level. It is highly recommended that you undertake a structured comparative proof-of-concept trial to expose the various salespersons' claims to the harsh light of practical usage. Gartner Inc.'s Magic Quadrant assessments can provide a useful stepping-off point for navigating the plethora of products and vendors in the marketplace. A few words of caution, however: some of Garner's product classifications can feel somewhat contrived, its underlying selection criteria do change over time (and should be considered with your particular situation in mind), and the rate of introduction and consolidation in the tooling world can outpace the best efforts of even the most experienced to call it.

Tool Selection Criteria

A number of professional tools exist for external monitoring. As with any other competitive market, the cost and detailed characteristics of individual tools will vary. You

should make your selections with knowledge of these details, in the light of the needs and constraints of your particular business. While several tools may fit the bill, mixing more than one of the same (e.g. active testing) is not good practice. Differences in test agents, design, test location, and more can make comparing results a difficult undertaking. At the very least, you should undertake a detailed parallel test study to determine the offsets between the tools, although these may not be consistent or linear.

In selecting suitable tooling, here are the key considerations:

Technical sophistication
: Ability to detect and isolate issues, and to support evolving site components such as RIA technologies (e.g., Flex); server push functionality; scripting filtering and custom coding; and validation for logins, message-based components (SMS), and multimedia stream monitoring (encryption codices, formats, and adaptive streams). Specific features within some tooling of potential benefit include the following:

  - Automatic analysis and scoring against key performance factors and page components
  - Alert sophistication and integration with third-party alert management tools
  - Screen capture and trace route on error or success
  - Ability to parse and trace individual (e.g., third-party) object performance
  - Ability to test complex transactions (e.g., select random or *n*th product from a list, select only in-stock product)

Ease of use of product with regard to scripting, test commissioning, and reporting as well as the speed of response of vendor support and script modification.
: The relative importance of these factors will depend on the time criticality and dynamic nature of the application(s) monitored and the extent of in-house skills.

  API
  : Ability to integrate data, alerts, and so forth with external systems

  Report flexibility
  : Ability to compare multiple perspectives (e.g., traffic, RUM, mobile, backbone) on single chart

Cost
: As with all tooling, total cost of ownership should be considered—licenses, personnel, training, and support. Dependent upon the flexibility of the tooling and the licensing model, strategies to mitigate cost include the following:

  - Maintenance windows (change test frequency outside core business hours)

- "Follow the sun" testing
- Continuous and active heartbeat testing

Flexibility/range of test perspectives (relevant to your particular site)

It is important to bear in mind that (a) all monitoring tools will provide a response metric of some kind, but (b) as performance analysts—or at least consumers of this data—we are required to draw business-relevant insights from them, and the measurements can and do vary wildly between vendors and across the various products offered by given providers (see Figure 7-6). The first key to effective external monitoring is to understand the question you wish to ask and select tooling that will enable you to ask it.

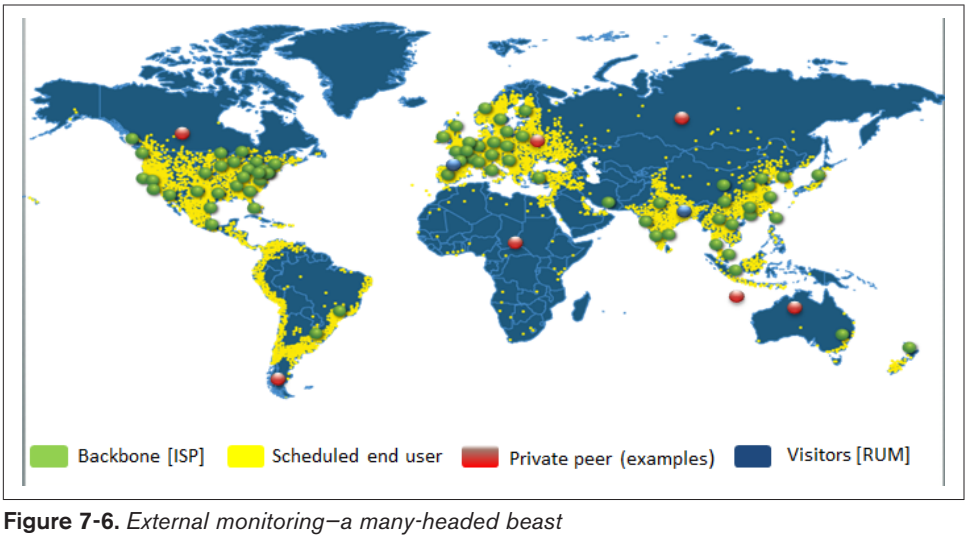


Figure 7-6. *External monitoring—a many-headed beast*

Tool selection considerations relevant to active (synthetic) and passive monitoring (RUM) differ, as we'll discuss next.

Active Monitoring Tooling

There are a relatively small number of active monitoring solutions to choose from. The predominance of web technology and the relative ease of passive monitoring (RUM) deployment makes RUM the easy option. As we will discover, active (synthetic) monitoring provides unique insights that RUM cannot offer, but is more complex to deploy and configure and is not intended for discrete monitoring of end-user performance. Synthetic monitoring typically has capability beyond just browser clients into thin-client technology, such as Citrix, and legacy fat clients such as SAP GUI.

- Clean-room (ISP backbone) testing.

- Test node location and class, connectivity/assurance.
- Number of nodes in relevant regions (minimum three at same round-trip latency).
- Number and type of test agents—native browser agents (and type) versus custom agents.

- End-user testing.
  - Consumer-grade end user versus fixed-bandwidth data center testing.
  - Number, quality, and control of test peers. It can be important to test from particular tertiary ISPs or at specific bandwidths:
    - Mobile device testing—emulation versus real devices.
    - Wireless carrier/known bandwidth versus hardwired ISP connectivity.

- The importance of controlled test conditions. Figure 7-7 illustrates the effect of wireless connectivity conditions. Paired 3G wireless carrier and WiFi-based tests for a panel of major UK ecommerce sites are shown. Note the high (and variable) discrepancy between the readings.

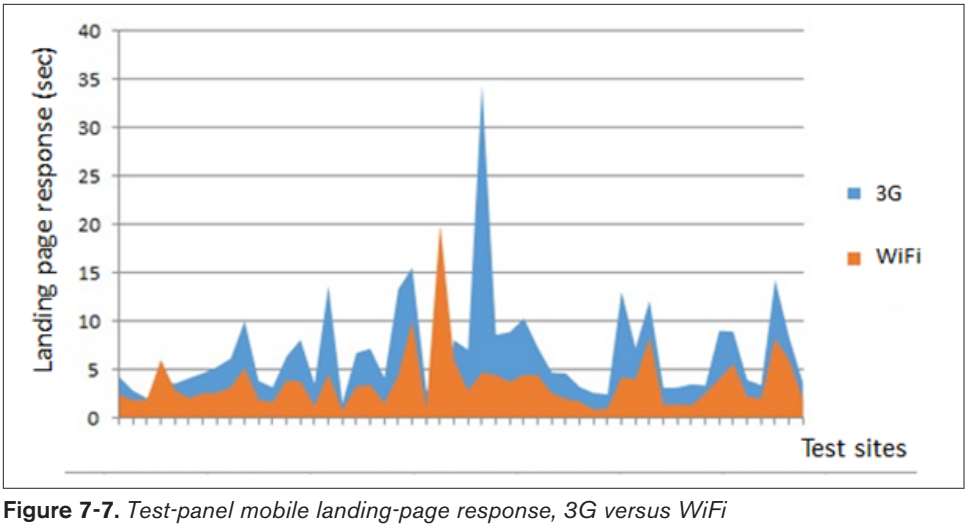


Figure 7-7. *Test-panel mobile landing-page response, 3G versus WiFi*

Passive Monitoring Tooling

At the time of writing there are over 50 passive (RUM) products available. Their large growth has been driven by the W3C's publication, and the subsequent adoption by many browser vendors, of a number of performance-related metrics. The metrics most relevant to this discussion are for navigation and resources.(see Figure 7-8).

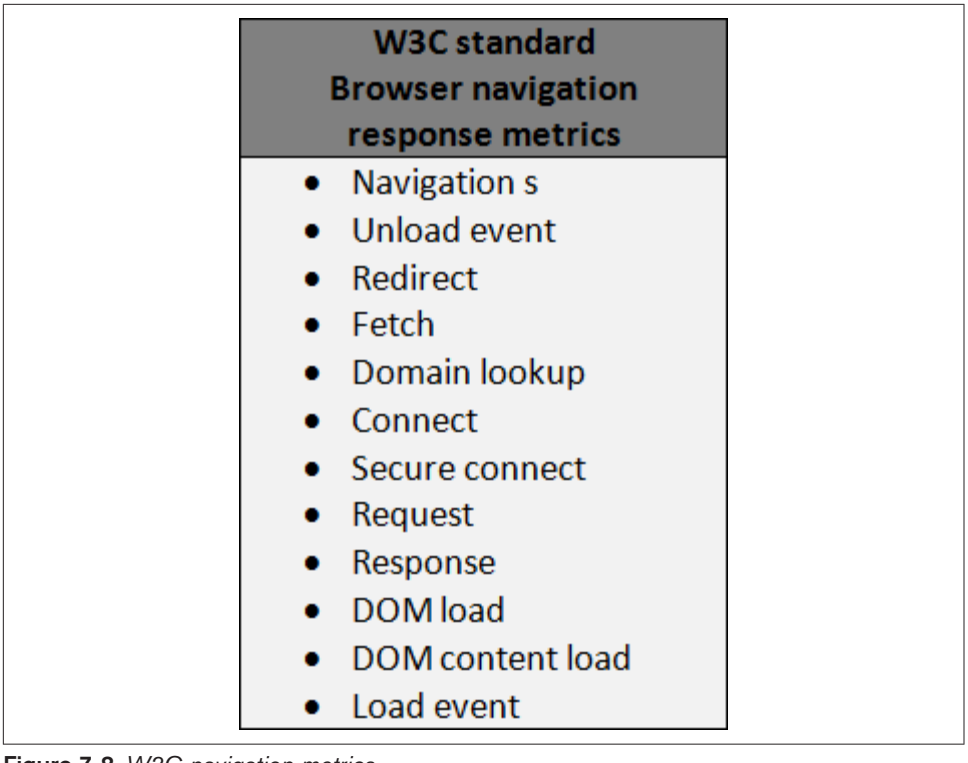


Figure 7-8. *W3C navigation metrics*

Although some convergence in functionality is becoming apparent (at least in terms of stated vendor roadmaps, even if not always currently available), the following aspects are worth considering when you are selecting a passive (RUM) product.

- Standalone or integrated (e.g., with APM tooling)
- Real-time reporting (can vary from less than 5 minutes to 24 hours).
- All traffic or traffic sampling.
- API availability and granularity.
- Alert capability; that is, the ability to customize and export to existing alert management tools.
- Browser and device coverage—versions, types (e.g., Safari).
- Manually instrumented or dynamically injected (this has a bearing on cost of ownership).
- Page or object level.
- User event capture.

- Complete use case or single page.
- Extensibility, that is, the ability to associate and report by specific session metrics (e.g., user login).
- Business metrics, including abandonment, conversion rate, bounce rate, and time on site.
- Reporting, including standard and customizable, email-based (e.g., *.pdf*), dashboard outputs.
- License model and economics (e.g., per domain, by traffic).

Creating an External Monitoring Testing Framework

Now that we've briefly described the various approaches to external monitoring, including their relative strengths and weaknesses, it is important to emphasize the importance of an effective test framework design. At its core, external monitoring must complement and extend the insights gained by infrastructure-based tooling (e.g., APM, server monitoring, network monitoring). To do this effectively while supporting the requirements of the business, external monitoring must provide information that reliably reflects the usage of the application and therefore maps closely to its effectiveness in meeting business goals.

As mentioned in the introduction, any monitoring will generate a result. Gaining accurate, actionable insights requires a clear understanding of the question that is being asked. A common error is either not to accurately reflect usage conditions, or (perhaps for reasons of perceived economy) to treat results from different tools as equivalent.

A Salutary Lesson

A good example of not accurately reflecting usage conditions was seen recently at a major UK online retailer. We were engaged to review its external monitoring strategy. It had deployed an impressive array of external tests, including a range of mobile end user tests using one of the major vendor toolsets. Its response results were superficially good, although a disparity between active monitoring and passive (RUM) was apparent. A more detailed examination of test configuration revealed that its mobile end-user synthetic monitoring was all set up using hardwired tier-one ISP nodes.

This option, which was included by the tool vendor solely to provide a comparison with wireless carrier–derived tests, provided a false sense of security that was rapidly punctured when wireless-based testing revealed a significant peerage issue affecting subscribers to one of the major networks (Figure 7-9).

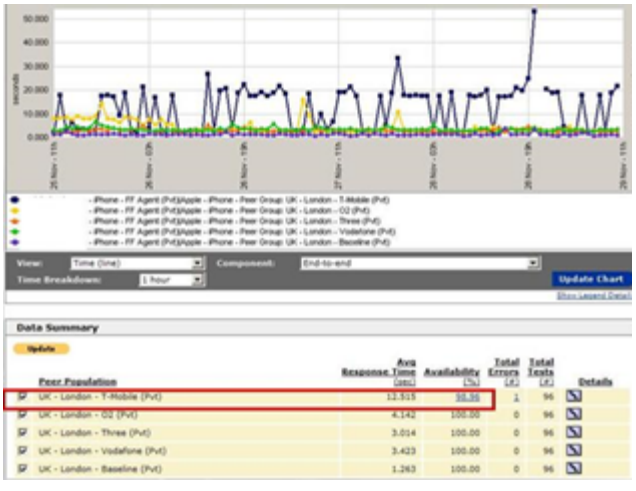


Figure 7-9. Silent issue affecting major UK wireless carrier

Building Blocks of an Effective Testing Framework

Best practices for external monitoring in many ways mirrors that discussed elsewhere in relation to performance testing; that is, it begins with consideration of the business rationale for, and characteristics of, the application to be monitored:

- Why does it exist?
- Who uses it?
- What are the relevant trends and strategic vision, and when is it anticipated that these will be realized?

Begin with good information on existing usage. The following questions illustrate this approach:

- Where are application users located—now and anticipated?
- How does behavior vary in different user subsets? (This may suggest the types of test transaction to monitor.)
- What is the pattern of traffic throughout the day, week, or month? This can vary in different markets. Modeling test frequencies and monitoring windows can make big differences to the license costs of some tools.
- What differences exist in browser/version and mobile devices? Popular browsers vary widely by region (e.g., Opera in Russia). Particularly in emerging markets, versions that are long obsolete in the UK and US may still have high market share.

For example, until recently, Internet Explorer 6 was the most popular browser in China.

- What is the strategic direction of the business with regard to new markets?
- What is the strategic direction with regard to mobile device usage and/or development of native applications for iOS and Android?
- Who are the key digital competitors (international or local) serving the target demographic in all key markets?

It is not necessary (although desirable, if somewhat impractical) to proactively monitor all regions and devices all the time, but the monitoring regime should seek to avoid major blind spots and to validate and extend data obtained from passive sources (i.e., passive tooling).

As you answer these questions, it is essential to work from objective sources of information (such as web analytics), rather than assumptions. The absence of promotion to a particular region does not necessarily imply that it is irrelevant from a revenue perspective. Prior to the aggressive internationalization of more recent years, many high street retailers experienced an English-speaking diaspora effect in terms of site visits, both from areas with a high population of English immigrants (such as southern Spain) as well as more permanent ex-colonial outposts. The world is not homogeneous; consider cultural/behavioral characteristics affecting devices, buying behavior, and use cases in general. As an example, consider the window shopping behavior of Japanese mobile users scanning Q codes. Other factors include the quality of the local infrastructure (developing world), legal/regulatory restrictions (Cuba, China), and the nature of user devices such as PC capacity and mobile devices (Africa).

Having determined the objective data required, and the key pages and use you need to understand, you should next consider the type(s) of monitoring to be employed. Key aspects are as follows:

- Multiple perspectives (triangulation)
- APM integration (root-cause isolation)
- Integration. (external, APM, and performance testing)
- Pre/per/post-load test results (memory leakage, cache effects, and similar)

Specific Design Aspects of Active Monitoring

When considering the design of a synthetic monitoring solution, you should take into account the following points:

Triangulation

It is important to test from multiple locations, and ideally from a range of core service providers. Backbone testing is designed to provide consistent, best-case external results, and test design should enable this. Ideally, a minimum of three ISP test nodes should be used per key location (e.g., country or region). If three nodes are not available within a specific country, either due to lack of availability from your chosen vendor, or to the requirement to test smaller economic areas, then locations with approximately similar round-trip latencies should be used. Political geography does not matter, but core Internet connectivity does, and this should inform your choice. Use key Internet hubs (in Northern Europe: London, Amsterdam, Frankfurt, Paris) rather than secondary backwater locations where possible.

Avoid tier-two locations (i.e., local data center) where possible—this is clean-room testing requiring consistent test conditions without test node system or communication bandwidth constraints. Failure to adhere to this suggestion will lead to problems with interpreting long-term trend data, and will affect your ability to set and monitor effective goals.

Frequency

Key considerations around test frequency include coverage, the nature of the output required, and cost management. A common approach is to set up active monitoring at a relatively low frequency, used in concert with continuous monitoring using passive (RUM) and/or APM monitoring. The detailed object level (and availability) data provided by the active testing supports pattern analysis and rapid issue isolation/confirmation of potential issues suggested by passive (RUM) results. Detailed test configuration will depend upon the tooling used, but I tend to work with groups of four nodes, each testing at hourly intervals (i.e., providing a nominal test interval of 15 minutes). Frequencies can, of course, be increased during investigation of issues, and often the tooling itself will provide additional confirmatory testing automatically when alert thresholds are breached.

If using active testing to support dashboard displays, you must ensure that test frequencies are matched to the refresh/display rate of the dashboard. Failure to do this will lead to long latencies in flagging changes in performance status and/or misleading results.

If you are using tooling where frequency of testing is a component of cost, then it is worthwhile to devise a test plan that is modeled to the patterns of usage of the application. Taking the time to introduce maintenance windows, or reduce test frequency outside core business hours across different global time zones, can lead to major savings in license costs.

Agent

In the early days of external testing, many vendors developed their own proprietary test agents. These were designed to provide good average visibility of site response across the (very limited) numbers of major browsers then in use, while increasing the detailed information available compared to the black-box limitation of native browser testing. Modern tooling almost exclusively prefers the data relevance provided by native browser-based test agents. Ideally, you should undertake testing using a variety of major browsers. This does not necessarily need to form part of ongoing monitoring, but periodic cross-browser studies enable you to capture any gross issues of incompatibility, and take into account indicative performance offsets.

Browsers differ in their detailed mode of operation, and it is important to review site design to account for these differences across browser types and versions. Such differences include the number of parallel connections, JavaScript handling, and the sophistication (or otherwise) of script prefetching. Site modifications to work around earlier browser limitations, if not reviewed, may lead to suboptimal performance with later modified browsers.

As a general rule, where possible, active monitoring should seek to match the browser agent used to the predominant end-user browser and version in the market concerned. In cases where this is not available, some vendors will enable spoofing by setting header and document object model (DOM) identifier information on a different underlying browser engine. This approach has variable benefits depending upon the nature of any underlying site issues, but it can be useful in identifying intractable problems. Such a hybrid approach is also commonly used in mobile emulation testing —that is, setting specific device headers on an underlying PC browser engine. This is flexible, and it provides consistency of performance. However, it will not uncover issues due to client-side system limitations. For these, real-device testing should be used.

Specific Design Aspects of Passive Monitoring

The key points with passive monitoring are to understand any blind spots and to compensate for them. Many passive (RUM) tools (those based on the W3C navigation metrics) cannot collect detailed metrics on Safari or older browsers. Supplementing with appropriate active testing (e.g., during periods of low base traffic) will ensure rapid response should issues occur outside core trading periods.

Integration with APM tooling will also serve to extend this understanding across the full range of visitor usage. Advanced passive (RUM) tools supporting individual session-level information are particularly useful. Such products support more advanced business—providing relevant metrics such as the relationship between performance and *bounce rate* or transaction abandonment.

When interpreting passive (RUM) outputs, bear in mind that the results relate to successful visitors to the site. Visit requests that were not successful will not appear. Potential visitors (or visitor subsets, such as mobile users of a particular carrier or device) who persistently experience problems may not seek to access the site in the future. For these reasons, it is advisable to approach passive (RUM) data interpretation from the perspective of expected results. Significant deviation from expectation (e.g., the absence or very low conversion rates for a popular device or certain region) should prompt you to perform validation testing using active monitoring and visual fidelity testing to offer confirmatory assurance of performance.

Isolating and Characterizing Issues Using External Monitoring

The approach detailed in this section is one that I have used for many years. It was designed for active monitoring (i.e., using scheduled synthetic testing) and hence it would typically be used to validate and/or investigate potential issues flagged by continuous visitor passive (RUM) outputs.

The ability to perform the testing will depend upon the tooling available.

1. Start with clean room (backbone results).
    - Examine average backbone trace (ideally for a minimum of two weeks).
        - What are the patterns of performance degradation? One-off, periodic, systemic?
        - What is the most likely correlation (e.g., traffic patterns, periods of maintenance)? If possible, examine passive (RUM)-based traffic versus performance.
        - Is there any progressive deterioration/sudden improvement (memory leakage, caching issues)?
    - Look for regular periods of poor performance in direct response to load (sine wave). Typically, this indicates an infrastructure capacity deficit, but other explanations exist.
    - Look for random outlying results.
    - Compare cross-browser results: are they the same for all? Focus on major browsers for the relevant country or region and/or the browsers reported by customer services.
        - Validate passive (RUM) results using active monitoring (set browser/device headers as required).

- — Review any automated analysis (consider the likely extent/significance of any red flags).
- — Analyze by ISP node:
  - — If problems point to the same node, select that node and chart on it alone.
  - — If multiple nodes are implicated, view a scattergram of performance by node to determine any patterns.
- — If there is a drop in availability, view using a test-by-time scatterchart to show the timing of the failed tests. Examine for cause of failure.
- — Validate availability by payload (byte count): is there any drop in content or evidence of errors in page delivery? Examine passive (RUM) data if error page(s) are instrumented.
- — Characterize the error (details will depend upon the tooling deployed):
  - — Implement screen capture on error if this is supported by the tooling.
  - — Is there evidence of content failure?
  - — Are there content match errors?

2. Focus on areas with performance degradation.

- Split by page (if transactional).
- Split page by component (e.g., core, third party). Consider the following aspects of individual page components: DNS resolution, connection time, first byte time, content delivery time.
- Compare performance relative to reference baseline data (for the site under test or top-performing competitors).
  - — Consider median and 95th percentile response values, together with a measure of variation (e.g., MAD, MADe).
- Review the pattern of results using overall scattergram patterns and by associating slow delivery times with spikes in performance of the relevant page or use case.
- Compare average results from multiple perspectives—ISP backbone, scheduled end user (PC and mobile/wireless carrier), passive.
  - — Correlate key parameters (e.g., traffic, ISP, end-user device).
- From the backbone, drill down to object waterfall view if supported by the tooling. Compare three or four good and poor results: what is the difference?
- View objects by host.

- Plot suspect object(s) over time: do any patterns emerge?
- Undertake a rapid object audit of page(s): number, type, size, header settings.

3. Look at the quality of service.
   - Generate an end-user error chart for the period.
     - — View by type (pie chart).
     - — View by time.
     - — Drill down on particular error(s).
       - — View report: are any patterns to objects/hosts/speeds/ISPs involved?
       - — Beware of "rogue" errors caused by limitations of connection speed—set (or filter) for known bandwidth.
   - Look at visitor passive (RUM) reporting.
     - — Where is the slow performance (geography, ISP, browser version, etc.)?
     - — Is there any association between traffic and errors or abandonment?
4. Capture, summarize, and report findings for future comparison.
   - Ensure that the data you capture is retained and easily accessible for baselining and comparison purposes.

Monitoring Native Mobile Applications

Internet-based delivery to mobile is evolving in two dimensions. First, mobile is increasingly becoming the primary retail digital purchase channel (anecdotally, several major retailers report close to 40 percent of their digital revenue from this source). The trendlines between PC-based purchase (declining) and mobile (increasing) are set to cross within the next 12 months (i.e., by early 2015), as you can see in Figure 7-10.

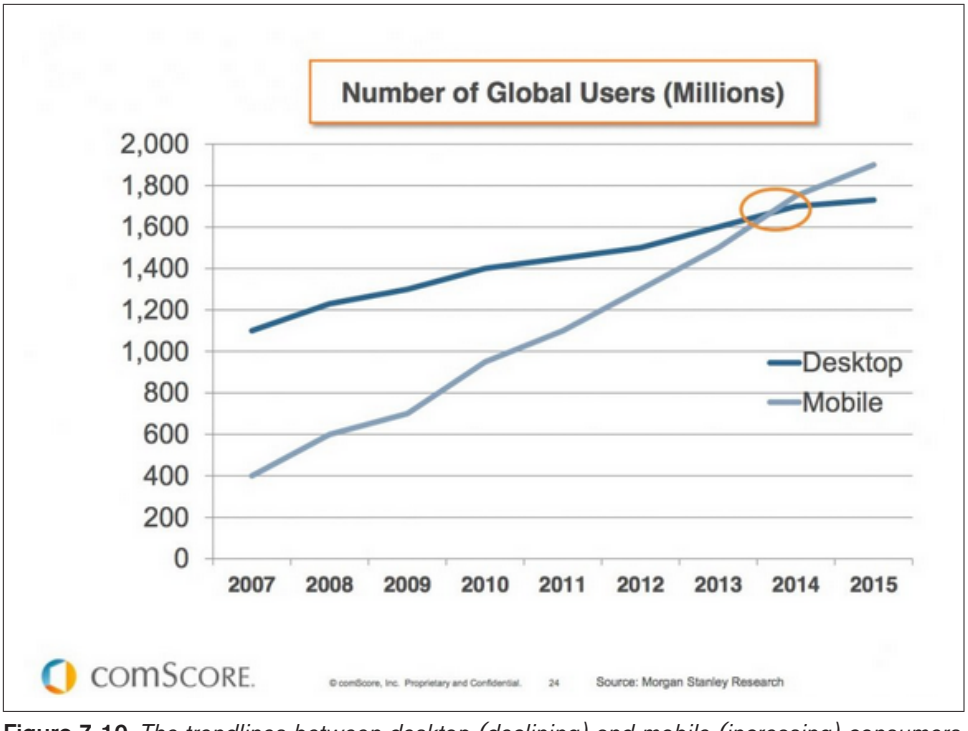


Figure 7-10. *The trendlines between desktop (declining) and mobile (increasing) consumers are set to cross by early 2015*

The second dynamic is the migration from relatively simple *m.* (mobile-enabled) websites to native mobile applications, although the advent of web apps, particularly when supported by some of the emerging network-independent components, may prove disruptive to this in the future. Setting aside web apps for now, native mobile applications have a number of advantages:

- They are network independent. You can use them in train tunnels and other areas where Internet connectivity is poor or nonexistent.
- They are productive at all times. In other words, you can store key information such as documentation and product SKUs locally.
- They are (or can be) faster and richer in functionality. Developers have access to a rich, closed OS environment, and therefore do not have to make cross-device compromises.

These strengths present difficulties, however, from an external monitoring perspective. These are complete, compiled applications, so the concept of the web page does not exist. They are also not necessarily HTTP/HTTPS based, so tooling designed for the Web may be unsuitable anyway.

Monitoring native mobile applications requires different approaches, which I will term *explicit* and *inherent*. Explicit monitoring uses a software development kit (SDK) to instrument the application code and link these together to create logical user journeys. Typically, iOS Views and Android Activities are referenced using the SDK. Data from application traffic (either actual end users or from specific VNC-linked devices) is collected and reported.

The ability to relate application performance to device variables (for example, memory, battery state, and signal strength) is particularly valuable. Crash metrics may additionally be reported in case of application failure. This approach is primarily used by mobile test tool vendors such as SOASTA (Touchtest).

Inherent monitoring is similar, but is easier to manage in that it requires a one-time instrumentation of the code with a compiled component. This is injected automatically at runtime, enabling capture of all user activity without requiring use-case scripting.

This is the route typically favored by native mobile application monitoring extensions of APM tooling (for example, New Relic, shown in Figure 7-11). The ability to compare performance across devices and OS versions is extremely useful, particularly if elements are used to feed composite dashboards (via the tool API) relating performance to business operation metrics (e.g., order rate).

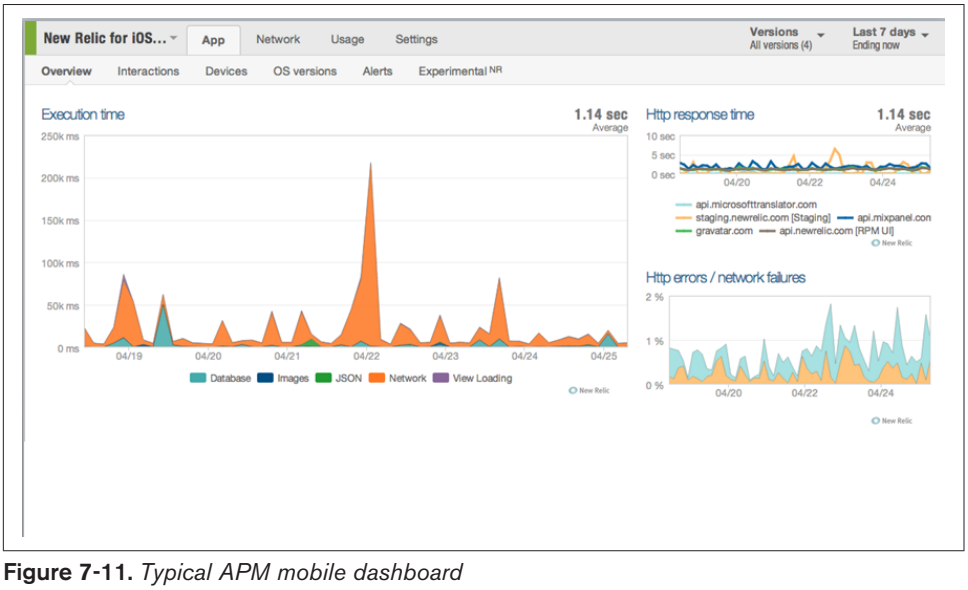


Figure 7-11. *Typical APM mobile dashboard*

From the point of view of external monitoring, the principal disadvantage of native mobile applications is that it is not possible to monitor apps without access to the code. Thus, it is not possible to monitor competitors, and sector benchmarks do not exist (at

the time of writing). This makes it challenging to appropriately define target performance.

Essential Considerations for CDN Monitoring

External content caching, or CDN, acceleration is one of the more established and widely used of the many performance-centric interventions now available. A recent (2014) study by my company Intechnica on the top 20 UK retail sites identified more than 75 percent of third-party services being represented by one major independent CDN provider alone. CDN adoption is far from a one-size-fits-all decision. The market itself is both large and fairly volatile, with the contending capabilities and claims of independents (Akamai, CD Networks, Limelight, Fastly, and many others), platform-based offerings (e.g., Amazon Cloudfront), and ISP-based services—which are often themselves delivered through licensing agreements with independents (e.g., Deutsche Telecom and EdgeCast). The broad choice is less relevant than the inherently dynamic nature of the problem. At its most basic level, CDN usage is far from binary; a given site may well employ several CDNs either directly or indirectly (through third-party content). So what are some of the core considerations?

- What are you accelerating? Depending upon the nature of the site and its usage, all or part may be accelerated. Is CDN caching applied just to large static images (for example), or all content?
- Where is it going? What are your key global markets? CDN providers differ widely in their capacity across the globe (indeed, some are regional niche players).
- Which primary regional ISPs are involved in web delivery?
- What is the nature of your core content and to whom is it principally delivered? Web HTTP and/or video (total download, adaptive streaming, etc.), PC browser, mobile device, or other (gaming console)?

Specific considerations apply in optimizing each of these areas. These may or may not be embraced by a particular product or service. As an example, delivery optimization to mobile devices may involve test compression (to reduce overall payload), video pacing, and TCP acceleration. All of these factors should influence test design and tooling approaches. An effective CDN-based strategy begins with initial selection, informed by some of the aforementioned considerations. As we have seen, determining optimal return on investment repays a thorough approach, and some form of comparative proof-of-concept trial is likely to pay major dividends. Following the purchase decision and deployment, key ongoing questions remain, if you are to maintain best value:

- Where are the priorities for usage optimization?

- Is there an end-user monitoring strategy in place across all key (and potential) markets?
- Do target KPIs (absolute or relative to local competitors) exist for all markets and usage categories?
- Is a service-level agreement (SLA) with defined interventions and remedies in place with the selected provider?
- How do these stack up, both during business as usual and across peak demand events?

These questions should be answered by an ongoing monitoring program supported by a best-practice-based testing framework. Having defined which areas (functional, geographic, and demographic) would potentially benefit from the use of CDN approaches, and assuming other interventions (e.g., appliance based, FEO) have been discounted, you should then ask the following:

- Is the CDN accelerating all appropriate components?
  - Some "entry-level" providers are restricted to caching static content such as images. If other elements are required (e.g., login and search), are these also included and effectively measured?
- Test design. Where does caching occur? At a single POP/ISP domain, via multiple hosts, or at cellular network mast?
  - Testing should reflect any potential constraints implied by the design of a particular CDN providers network.
  - For maximum effectiveness, conduct prescheduled, active (synthetic) testing from appropriate end users (PC and/or mobile) in the country concerned.
  - You should employ a parallel testing methodology comparing performance directly to the origin servers with that delivered locally by the CDN provider.

This approach will ensure not only that end-user performance is satisfactory, but also that the extent of benefit delivered to (and purchased from) the CDN is maintained.

An example may help here:

Comparative monitoring of cached content to PC end users in a key regional market (known connectivity conditions), origin versus local cache

Figure 7-12 shows the overall benefit delivered (some 7 seconds on average). The highlighted area illustrates a convergence between the traces (i.e., transient failure of the CDN to deliver performance acceleration).

Figure 7-12. *Timebase: 1 hour*

Importance of test location

Figure 7-13 compares the number of CDN hosts used to deliver content (i.e., be monitored) over a one-week period by a major CDN provider. Testing from the tier-one ISP cloud provisions (left image) cached content from 2 CDN host groups (16 individual hosts).

The image on the right shows results from the same geography when tested from a wide range of PC end users (using a variety of tertiary ISPs). Ninety-six individual hosts were employed.

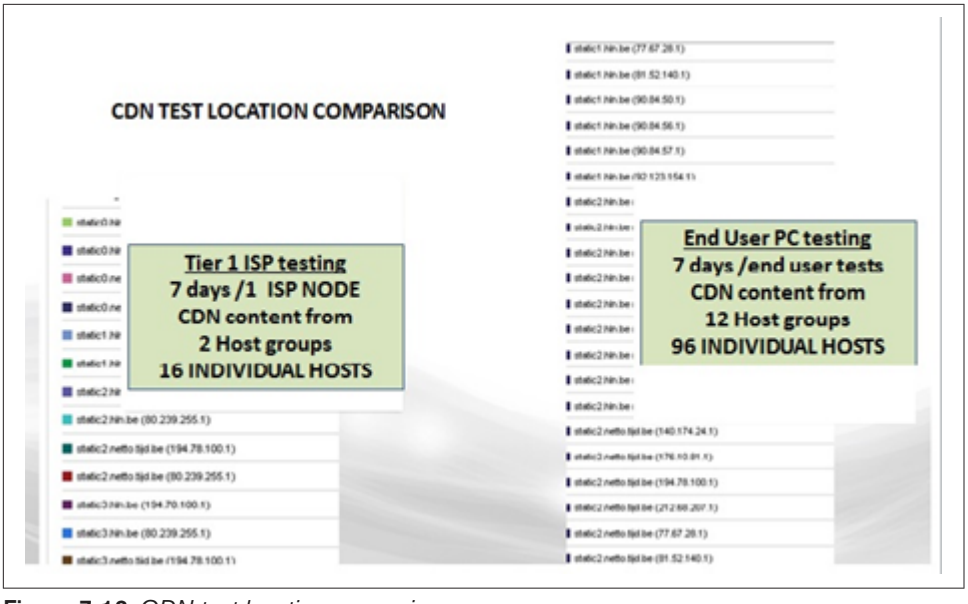


Figure 7-13. *CDN test location comparison*

Relevance

Failure or poor performance by any of the 80 hosts not detected by ISP-level monitoring would not be detected, while being paid for, and impacting all end users of that particular host.

Figure 7-14 illustrates the range of performance at the individual CDN host level during a five-day test window.

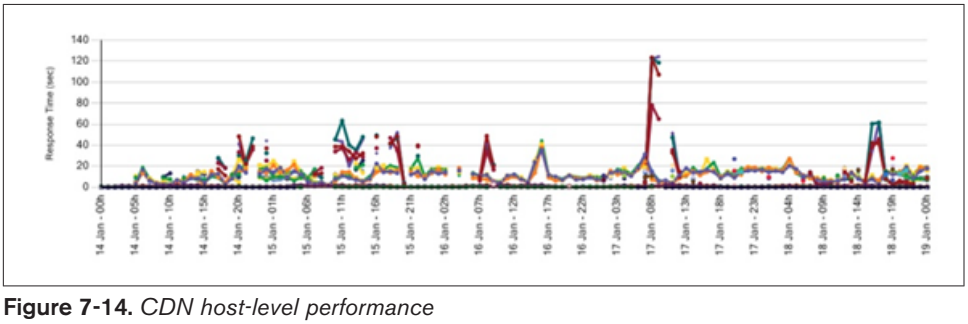


Figure 7-14. *CDN host-level performance*

Note that the selection of CDN delivery node location will be influenced by several factors. These include existence of a point-of-presence (POP) in the region to be optimized. Absence of a local POP need not be disastrous, but it will mean that round-trip times to the source of the cached data (e.g., UK to US) will be extended.

Even if a local POP exists, it is important that end-user traffic is mapped efficiently to the CDN host that will most efficiently deliver the cached content. Such mapping is typically effected using the IP location of the DNS resolver rather than that of the end user. In the case of most consumer end users, DNS resolution will be undertaken by their ISP at a central in-country location. Corporate or institutional users may, on occasion, use DNS proxies distant from their location (e.g., on another continent). This may lead to inefficiencies due to incorrect CDN location mapping, with commensurate round-trip latency. Whether or not this occurs, you should certainly correct for it when undertaking CDN assurance screening, by identifying and blacklisting such end-user peers.

Finally, having selected your CDN provider, ensure that configuration is optimized, both in terms of origin/CDN interaction, and ensuring that headers are set appropriately at the individual object level. Failure to do (and police this) can result in performance being compromised by origin rush, whereby the individual CDN delivery nodes, rather than serving locally cached content, are constrained to request all or part of the content from the origin servers, thus defeating the objective of the exercise. Such misconfiguration can have relatively dramatic effects on site performance even if a relative small proportion of total content is misconfigured, particularly if the origin infrastructure is prone to capacity constraints.

Performance Results Interpretation

Effective performance management requires an understanding of ongoing performance, both in absolute terms, and relative to key competitors and visitor expectations. A wide range of tooling exists for external monitoring of application performance. All will produce a series of results. The validity of these results, and therefore the quality of the business inferences drawn from them, can vary significantly. This section highlights some pitfalls and suggested approaches to maximizing beneficial outcomes from investment in performance.

You must take care when selecting appropriate tooling, taking into account the nature of the application, its detailed technical characteristics, the nature of visitor traffic, the location of key global markets (existing or planned), and many other considerations. Having developed a test matrix for external monitoring (with regard to established best practice), you should collect and inspect data over a full monthly business cycle. This will provide a set of reference baseline values across a range of business-as-usual conditions.

A given application will typically exhibit recurrent patterns of performance behavior across a weekly demand cycle. Such patterns often provide an initial high-level indication of overall application characteristics, such as capacity deficits or gross issues in

serving particular visitor groups (e.g., mobile users). These preliminary findings can inform your subsequent approaches to the performance strategy.

Assuming that overall application performance is reasonably stable, it is then appropriate to determine the KPIs and any associated SLAs that will be used for performance understanding and management.

Key Performance Indicators for Web-Based Ecommerce Applications

KPI determination should reference many factors. In summary, these include the following:

- Best-practice values derived from top performers in a relevant business sector, "bellwether" sites (i.e., places where your customers are visiting if not on your site), and expert advice.
- Performance of key competitors.
- Historic performance of your site.
- Key use cases—that is, those with maximum business relevance. These fall into three categories:
  - Direct revenue sources (e.g., purchase transactions).
  - Activities that reduce business overhead (e.g., web-based customer service chat versus telephone-based support).
  - Functions that are important to brand perception and/or deferred revenue. Examples include color simulator for paint/car manufacturer; booking sales representative appointments.
- Key pages. Use case should always be deconstructed to examine the performance of each page or logical step, as an exclusive focus on end-to-end response times may mask poor performance at a key step. Poor performance at individual key steps has been shown to be highly correlated with abandonment.
- Subpage performance—that is, response and consistency of individual objects/links, either core or third party.

The purpose of KPIs and any SLAs based on them is to ensure and optimize business outcomes. These outcomes can take many forms, but those most relevant to ecommerce are revenue, customer satisfaction, and brand perception. Site/application behavior has both a direct and indirect impact on performance. Direct factors include lack of availability and high levels of delayed page response and inconsistency.

From an end-user-experience perspective, key performance-related metrics are response time (seconds), availability (%), and consistency (standard deviation or

median absolute deviation). You need to monitor and understand the influence (if any) of performance on conversion rate, visitor abandonment, bounce rate, and stickiness (time on site).

Performance has been shown to have a less direct effect on other key revenue metrics (e.g., conversion and abandonment). These factors are influenced by many variables, including site design, user demographics, and aspects of the offer (e.g., shipping costs). Setting and managing performance KPIs while controlling for other variables has been shown to have a positive effect on digital revenue. Visitor metrics should be analyzed using robust statistics to mitigate the distorting effect of outliers. Once thresholds have been established, the use of—and setting KPIs for—APDEX-derived measures of visitor satisfaction is useful. (APDEX will be described shortly.)

Consider key markets: the nature of competitors and delivery conditions (quality of core Internet infrastructure, browser version, mobile device usage, and other factors) affects the detail of the monitoring undertaken and target KPI values. There can be little point in applying common KPIs derived from performance in advanced markets indiscriminately across all regions. It's better to understand local conditions and constraints and seek to be the best on the ground market by market. This is likely to provide a more pragmatic and lower-cost approach.

You should have in place a procedure for the regular, systematic review of all KPIs. Factors to be considered include the following:

- Historic performance relative to existing values
- Trending of all defined metrics and business processes
- Performance and trends in key competitors and bellwether sites (e.g., BBC, Facebook)

A fully evolved performance strategy gives attention to both objective (hard) and subjective (soft) aspects of performant delivery:

Hard metrics

- External: set target responses to the ISP cloud (best case) as well as to PC and mobile device end users across a range of connection bandwidths, to cached/uncached visitors, and between total (all object) page response and above-the-line (browser fill) response time. External KPIs/SLAs should exist for the following:
  - —Key use cases
  - —Key pages
  - —Third-party objects/affiliates

- Internal: Set and manage internal (infrastructure-based) KPI metrics. These metrics are outside the scope of this chapter.

Soft metrics

- Visual fidelity across key browser version/operating system/screen resolution combinations, and to key mobile devices/form factors.
- Fidelity of page links (internal and external).

Setting KPI Values

Consider the nature of the application and the goals for the business/brand. For example, do you wish to be in (and invest to achieve) the number-one benchmark position, top 10 percent, top quartile? A common approach is to aim for a position in the top 25 percent of all companies in your business sector (in key markets), with a response at least as good as a panel of specified direct competitors.

Reference second-order values (quoted retail industry average), as outlined in Table 7-3.

Table 7-3. *Reference second-order values*

eCommerce retail KPI	Upper quartile benchmark mean
Conversion rate (total site visitors)	>3.0%[1]
Shopping cart abandonment	<60%
Target page weight (PC browser)	<1 Mb
Add to cart (new visitors)	6.4%[2]
Add to cart (returning visitors)	10.8%[3]
Total page weight (mobile devices)	*[3]

Quoted average values are very dependent upon the nature and positioning of particular sites, the selection of data, and the dynamic nature of the metrics themselves. Users should discuss this with their web analytics and/or tag management system provider to derive target KPI data more specific to their precise offer. Consider the following:

- Understand the key users of the site in terms of browser versions, connectivity (fixed wire and mobile dongle), mobile device, location, and ISP.
- Distinguish between the performance of first time (uncached) users and frequent visitors. Note that unless specifically designed (by scripting duplicate consecutive

tests), active (synthetic) monitoring testing is usually uncached, thus providing worst-case responses.

- Understand key trends in interface (e.g., specific mobile phone or tablet) uptake among your key target demographics.
- Test performance to major browser (e.g., Internet Explorer, Chrome, Firefox) upgrades at beta stage if possible.
- In terms of capacity, undertake performance assurance of application KPIs:
  - —Across the range of demand of the normal (daily/weekly/monthly) business cycle.
  - —To anticipated peak demand—Christmas trading, sales, and so on (typically 1.5x projection from previous years).
  - —Prior to release of significant site/application upgrades or other related activities (e.g., hosting provider or primary ISP changes). Periodic nonfunctional performance testing should form part of continuous integration–based DevOps strategies.

External KPIs are typically as follows:

- Business relevant, that is, they affect revenue throughput directly or indirectly.
- Based on degradation over a period or with a defined pattern of occurrence rather than point or transient episodes.
- Based on the three cardinal metrics:
  - —Response time (seconds)
  - —Availability (% successful tests)
  - —Consistency (standard deviation, seconds)

What to measure? Typically:

- Use-case times
- Individual page times
- Response consistency
- Object/delivery component (e.g., DNS) times

External KPIs should specify the source of testing:

- Site visitor monitoring (perceived render times, browser version/device response times)
- Clean-room ISP cloud testing (absolute metrics, competitor benchmarks)

- Scheduled end-user testing (geographic/specific location testing, subjective metrics)
- Performance capacity testing (internal/cloud/end user)

There are three types of metrics:

- Absolute
  - Breaches indicate a *prima facie* issue requiring intervention and resolution.
  - Deviations from range of standard behavior across the business demand cycle.
- Subjective
  - Based on judgment and/or empirical behavior data (e.g., abandonment rates).
- Relative
  - To competitors (public sites).
  - To other internal applications.
  - To other objective criteria.

The preceding categories apply to monitoring metrics, that is to say, the application response across the normal cycle of business usage. Performance metrics apply to the application response across conditions of simulated demand (performance testing). SLAs imply involvement of another actor (typically external, but may be a defined internal IT/operations function).

The Application Performance Index (APDEX)

The APDEX, or Application Performance Index, is a means of comparing end-user satisfaction by measuring application response. It was initially derived in 2005 by Peter Sevcik of NetForecast Inc., and is currently supported and developed by corporate members of the Apdex Alliance (typically larger performance-related companies such as Akamai, Compuware, and Keynote Systems).

The principal goal of APDEX is to provide a clear and unambiguous method of comparing user experience across any application, user group, or enterprise. All members of the Alliance produce products that calculate and report the APDEX score. In theory, it is possible to calculate the index from any end-user response metrics. However, this does not obviate the requirement to critically examine the results delivered by your tooling to ensure that they accurately represent the end users' perception of the application response. As an example, user wait times are not equivalent to the total page response metrics produced by synthetic monitoring systems. These are essentially recording the total network traffic time, so unless the above-the-fold time is indicated

(as it is with some synthetic tooling, such as Google WebPagetest), a passive (RUM) tool providing a browser fill metric is required.

The APDEX has a potential range of zero (least overall satisfaction) to one (greatest overall satisfaction) experienced by users of the application. Users are segmented into one of three categories (Satisfied, Tolerating, or Frustrated) on the basis of their experience of application response. The index is calculated through a comparison of the number of responses in each category relative to the total sample.

Categories are related to a target value (*t* seconds) as follows:

Satisfied zone
: Zero to t

Tolerating zone
: t to 4t

Frustrating zone
: 4t and above

If you are interested in the theoretical basis of APDEX, references are provided at the end of this chapter.

Management Information

Before embarking on an information delivery program, you need to define and group specific indicator metrics, via a minimal number of relevant summary management reports. Popular approaches include the following:

- Balanced scorecards
- Traffic light (RAG)–based matrix
- Dashboard reports

Although some metrics, such as availability, are likely to be common to all business constituencies, the particular metrics selected and, in particular, the RAG thresholds used will differ depending upon the target audience.

For example, the Operations team will require early warning of potential issues. It will typically set far tighter warning thresholds based on root-cause indicators. Senior business (e.g., C-level) users are usually focused on issues that impact business performance (e.g., order rates, market share) and are focused on first-cause metrics that will typically have broader thresholds.

Having agreed upon a master set of KPIs and the type and mode of delivery (heads-up dashboard, daily email, SMS alert), you must now consider how best to process the raw performance data for maximum relevance.

Data Preparation

The importance of pattern interpretation (that is, the variation of site behavior across the business demand cycle) has been mentioned. However, in drawing conclusions from data, it is important to consider and minimize where possible the inherent limitations in the figures presented. These derive from three principal sources:

- Limitations inherent in the nature of the test design. As an example, some tooling vendors use artificially throttled bandwidth tests from core Internet cloud locations as a proxy for end-user performance. Others use outdated HTTP 1.0–based agents, which cannot monitor modern client-based processes.
- Limitations dependent upon the test source—for example, using tertiary retail ISP test locations rather than near-tier-one dedicated ISP or LINX-based tests.
- Limitations based on the nature of the results obtained. This is covered in the following section.

Statistical Considerations

Ongoing week-to-week or month-to-month performance monitoring (as opposed to specific deep-dive issue-resolution activities) are based on three parameters:

Availability:: The proportion of external scheduled tests that are successful or fail—reported as a percentage of total (active) tests. Response time:: The time taken for a process, page, or transaction to execute. Various subtleties should be considered (e.g., total load time versus above-the-fold or browser-fill times); these are covered elsewhere. Reported in seconds or milliseconds. Consistency:: The variation of the results from multiple repetitions of a given test. Often reported as standard deviation, although there are better metrics (as discussed momentarily).

Treatment of availability is acceptable, provided any inherent underlying confounding variables are noted/known. These would include such factors as low test volumes. As an example, a single failure from a run of 5 tests is reported as an availability of 80 percent. The same single failure from 100 tests results in a reported availability of 99 percent. When you are considering availability metrics, it is important to exercise particular care in distinguishing between reported (or what I term *nominal*) availability and actual functional availability.

Active (synthetic) monitoring tooling will typically record a page load as successful even if a proportion of the content has failed to download (or, indeed, if it is an error trap page of the "Our site is having problems; please try later" type delivered in place of the requested page). Before you regard reported availability as accurate, therefore, it is strongly recommended that you do the following:

- Smoke test the results. A recorded availability of less than, say, 85 percent would almost always be accompanied by other symptoms, even if they are calls to the customer support center. Is this a test artifact?
  - Does the site work when you visit it?
  - What do the site traffic and order volumes look like?
- Run a payload test. This is very useful for picking up the delivery of partial or error pages.

Measurement of response times and consistency can present more difficulties if you do not take care to understand (and, if necessary, correct for) any inherent distortion in the raw data.

Consider the charts in Figure 7-15 and Figure 7-16.

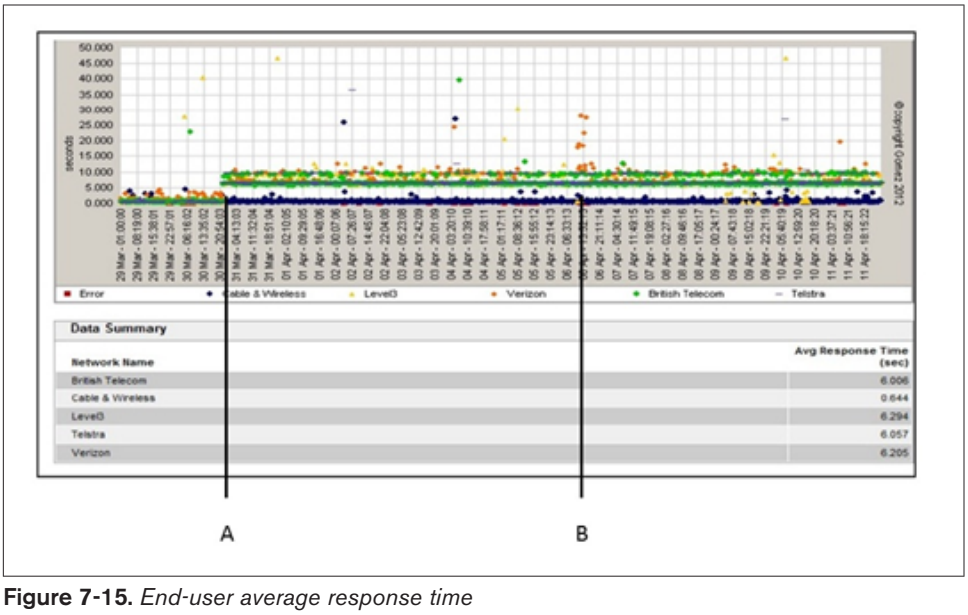


Figure 7-15. *End-user average response time*

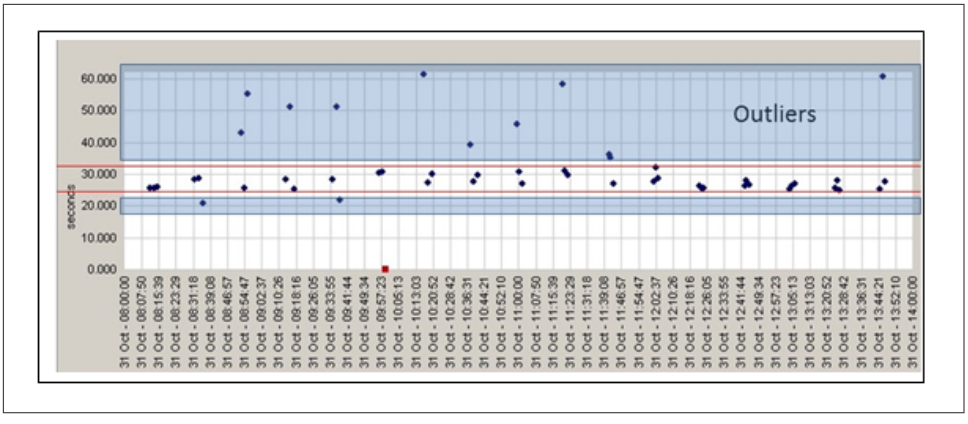


Figure 7-16. *End-user outlier response time*

Figure 7-15 illustrates the value of examining the raw individual data points before determining the best statistical treatment. Averages can hide a multitude of sins. In this case, a scattergram display was used. It represents each individual data point as a single point, color-coded by its origin (location of ISP node). The value of such an approach in understanding underlying application behavior is clear.

Point A shows a state change in response from the British Telecom test node location. As BT carries over 80 percent of UK traffic, and any performance changes (e.g., due to peerage issues between customers using BT and the data center home ISP) can have major commercial consequences in terms of user experience. After the state change at A, two separate performance bands are detected in the BT results. This is characteristic of load-balancing issues, but more relevant to our current discussion is the effect on the distribution of results producing a multimodal distribution.

At point B, you can see the effect of an issue localized to the Verizon (orange) node. In this case, the effect is transient, and would be well reflected in an increase in average response, flagging alerting, and more detailed investigation.

Figure 7-16 illustrates a different scenario. In this case, the scattergram shows a high dispersion of results that is not transient (as with the Verizon example), but systemic. The application throws off frequent, sporadic results that are much higher (or in a few cases, lower) than the majority of the results. In the chart, the core response times (average of ~26 seconds) are bounded by the red lines. Outliers (which in certain cases are three times as long) appear in the (blue) shaded portion of the chart outside this area.

This kind of dispersion in results is not uncommon, particularly in applications that are operating close to their infrastructure capacity, either in the general run of business, or at times of peak demand (e.g., pre-Christmas trading, or in intranet applications close to internal deadlines).

The issue that this presents in interpretation is that the simple and commonly used measures of central tendency—the arithmetic mean (for response) and the standard deviation (for consistency, or results variability around the mean)—make a core assumption of normality of distribution This is demonstrated in Figure 7-17.

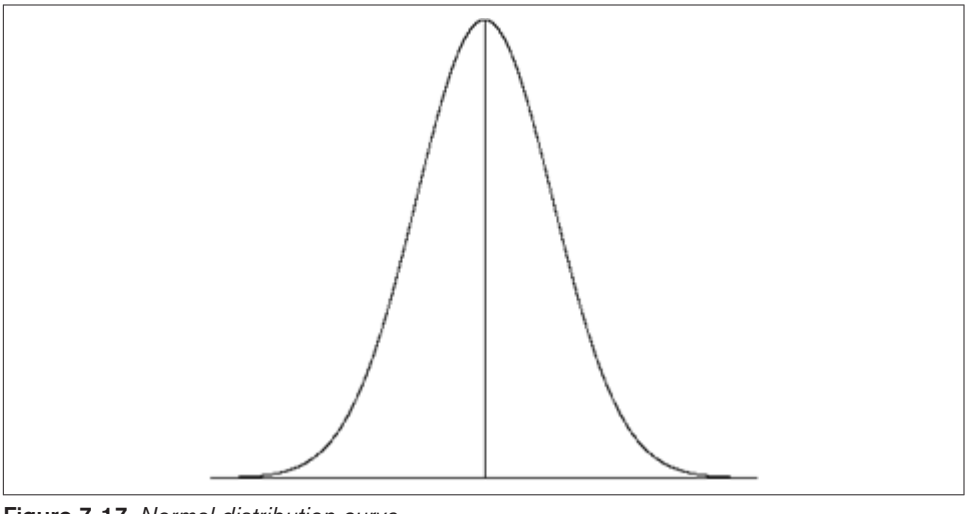


Figure 7-17. *Normal distribution curve*

As you can see, this distribution is symmetrical (i.e., the proportion of results of a given value are the same to the left and right of the center line). The presence of outliers (or other distortions such as distributions that exhibit kurtosis or skewedness) render this core assumption invalid and tend to likewise invalidate results that are based on it. In simple terms, if an application has responses to (say) 1 in 10 visitors in one month that are much higher than the majority, and the following month this does not occur (or the proportion of outliers changes significantly), the mean value will be markedly affected. This sensitivity to data quality makes the mean a poor choice when you are comparing a long series of performance results. Fortunately, a number of alternatives exist. These are more or less involved depending upon the standard characteristics of the data. While professional statisticians may have resource-to-data transformations (e.g., logarithmic), in the case of most correctly specified corporate systems, one of two approaches usually suffices, namely:

Monitor the median rather than the mean. The median is the value that cuts the data in half. It is, in statistician's parlance, a more robust indicator than the mean. In other words, it is less affected by small numbers of results that vary from a standard normal probability distribution.

In cases where preliminary examination of the data shows a large number of outliers and/or the results of such regular outliers are likely to significantly distort the mean, you should see a truncated distribution when reporting on central tendency (i.e., the

peak of the probability distribution curve—which is what the mean seeks to do). As mentioned, data transformation is one solution.

Provided that the residual data volume is sufficiently high, a simpler solution is to use a *trimmed mean*. The trimmed mean is calculated in the same way as a simple arithmetic mean (i.e., the sum of all values divided by their number), but a fixed proportion of data points are removed (the same amount for either end of the scale). As an example, a 20 percent trimmed mean would discard the highest and lowest 20 percent of values. The amount of trimming required will depend upon the particular results distribution obtained.

If the data is reasonably normally distributed, then the standard deviation can be used. Should many outliers exist, another truncated distribution (such as Caudy) may be preferred to model the data. This is similar to Figure 7-17, but it has fatter tails. You may be more familiar with Student's *t*, which has the same probability distribution (at 1 degree of freedom) as the Caudy distribution. The equivalent consistency measure for this model is the median absolute deviation (MAD). Due to differences in its calculation, the MAD is more resistant to the distorting effects of outliers than the standard deviation.

Although replacing the mean and standard deviation with the median (or trimmed mean) and MAD will assist in ongoing comparison and iterative goal setting for your site, it has the disadvantage that these figures are not quoted for other (e.g., competitive) sites. Although the assumption of normality cannot, of course, necessarily be made for these sites either, it may be helpful to note that, for normal distributions, multiplying the MAD by 1.48 will give the appropriate standard deviation.

An alternative, if justified, would be to monitor key competitors and apply the transformation to the results obtained. This is rarely an appropriate use of investment, but one other metric is worthy of mention, as it can provide a comparative figure independent of the mean. This is the *percentile rank*. The percentile rank represents the figure above (or below) which *x*% of results lie. This metric can sometimes provide a simple alternative comparator for performance over time. It is more commonly used in situations such as the comparison of year-on-year performance of students at public examinations than in performance monitoring. It is most seen in an IT context (often at the 95th percentile) in situations of utilization/capacity planning, such as burstable bandwidth contracts.

Correlation

Statistics is a fascinating field of study, but the further reaches of it are perhaps best left to researchers who have the time and justification for their efforts. The small number of robust metrics just described is sufficient for almost all day-to-day monitoring purposes.

One final metric, correlation, is worthy of mention in an introductory text. You may have discovered an issue that has several possible causes. Correlation enables statistical comparison of two data sets. It provides a measure of the extent of association between them. This does *not* necessarily imply causation, as other factors may be acting on the data measured, but measuring correlation can certainly serve to exclude unconnected metrics, and to some extent rank others in terms of likely importance.

Correlation between data may be simple (i.e., linear) or more complex, requiring various transformations of the raw data. Rapid comparison of data sets for linear relationships is straightforward with Student's *t* test. Details of how to apply the test and the tables required for derivation of the correlation coefficient can be found in any basic statistical text.

If you are using the *t* test, here are a few points to bear in mind:

- It is a measure of a linear (straight line) relationship between variables, but others (e.g., exponential) may exist.
- Its accuracy will be affected by the size of the sample—biological statistics sometimes works with as few as 20 pairs, but the more the better—and this is not usually a constraint in performance work.
- The correlation coefficient varies between zero (no relationship) and one (perfect relationship, either negative [–1] or positive [+1]).

Interpretation will depend upon the circumstances, but values greater than around 0.7 may be regarded as relatively strong.

One final point: if you are using Student's *t-tables*, you will find values for one- and two-tailed tests. Which to apply? This depends on what you are trying to determine. A one-tailed test indicates the strength of a single-direction (positive or negative) relationship between the pairs of figures—in other words, when testing the strength of a particular hypothesis. The two-tailed test indicates the likelihood of either a positive or negative correlation (i.e., when you have no idea whether any relationship exists).

In performance work you will almost always be using one-tailed tests. As an example, consider Table 7-4: what aspects of third-party affiliate content have the most impact on site performance?

Table 7-4. *Third-party response-time contribution*

Page response (total load time [sec])?	# Third-party hosts	# Third-party connections	# Third-party objects
Median	9.4	26.5	98
91.5	Correlation coefficient (Pearson r)		0.723

In this case, neither of the factors examined had an overwhelmingly strong claim to be the key factor, and (of course) they are all interrelated; the more third-party objects you have, the more connections there will be, and so forth. A clear understanding would require a larger sample volume and (probably) better test design than was applied here, but, inasmuch as it illustrates the point, examination of the correlation coefficients shows that, for this single test at least, the number of third-party hosts appears to have the least single influence on page load performance, while the number of third-party connections has the greatest.

Further investigation could consider whether some combination of individual factors provided a stronger result (e.g., one metric divided or multiplied by another), or indeed whether a weighted model (applying a constant value to one or several metrics) was more satisfactory. The next step would depend on your reasons for conducting the investigation.

Effective Reporting

Application monitoring is about gaining understanding. This short section addresses some aspects of the delivery of that understanding via appropriate management reports. Effective management reports should be as follows:

- Relevant
- Timely
- Readily comprehensible

Creation of reports is not a mundane activity. The challenge is not the creation of the reports themselves; this can usually be undertaken within the monitoring tool, or alternatively exported to one of the literally hundreds of report/dashboard creation applications available. Rather, the important aspect is to make the reports appropriate and actionable. It is sensible to begin by considering the audience(s) for the various reports, both technical (IT, operations) and business (C-level management, department heads, marketing, ecommerce, and others).

Reports should address the various user groups in their terms. Some they will tell you, and others you can to infer from the nature of their roles and (particularly) how they are remunerated. An example may be an emerging markets product or marketing manager. Data relating performance to key in-market competitors and monthly changes in net new customers will be more meaningful than more arcane throughput or network utilization metrics useful to an operations function.

When you understand what is required (in an ideal world), an audit of existing systems (web analytics, sales, order processing) will reveal the base data available to enrich basic performance data. Web analytics in particular is a useful source of business-relevant metrics (conversion, abandonment, bounce rate, stickiness), should these not be available within the monitoring tooling itself.

Report outputs should be as simple to comprehend and as short as possible. Balanced scorecard approaches, which score agreed-upon key metrics at regular intervals (daily, weekly, monthly) are often useful. They are particularly powerful if supported by a traffic-light-based RAG (red/amber/green) output.

Thresholds, also, should be appropriate to the audience. An Operations team will typically wish to have far tighter triggers in place—providing early warning of approaching issues—than business management. Senior management, in particular, typically is interested only in output impacts such as revenue or significant competitive deficits. Too many red lights on their reports are likely to result in the sort of profile-raising phone calls that most ops teams can do without!

Trend charts are useful in setting and managing iterative goals, and for future planning. The appropriate statistical treatments should be applied to any projections generated. A combination of regular, standard charts to support specific regular business meetings, together with real-time dashboards in key locations, is useful, both as a source of actionable data and to convey a perception of control across the business.

Finally, consider using subjective outputs. The comparative videos generated by Google WebPagetest are a good example, as are APDEX values for performance-related customer satisfaction; these can trigger beneficial actions (e.g., IT investment) as opposed to dry tables, which, however accurate, are less emotionally engaging.

Competitive Understanding

There are several aspects to understanding website performance relative to that of key competitors. These include the following:

- Perspective (clean room versus end user—specific category or all visitor)
- Device and delivery mode (PC browser, mobile phone, tablet; fixed-wire and wireless network)

- Geography
- Selection criteria (sector competitors, bellwether sites)

Internet cloud testing provides the best data for absolute competitive comparison in ideal conditions.

End-user data is noisier, but also more revenue-relevant. Passive (RUM) visitor data is continuous and has broad reach, but is often partial (e.g., excluding Safari and some older browser versions) and does not record failed visit attempts. Typical passive (RUM) data is based on page delivery rather than on individual objects.

The line chart in Figure 7-18 provides an average performance trace across a number of test locations. The example compares two key metrics: response (total page load) time and availability.

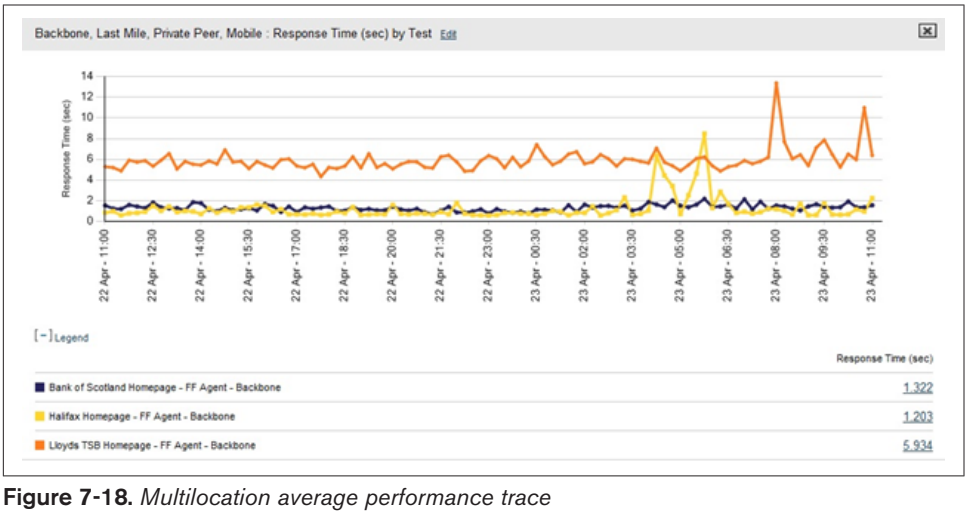


Figure 7-18. *Multilocation average performance trace*

Benchmarks (public/private)

Benchmarks provide a periodic (typically weekly or monthly) league table of the performance of the target site relative to a set of defined comparator sites (Figure 7-19).

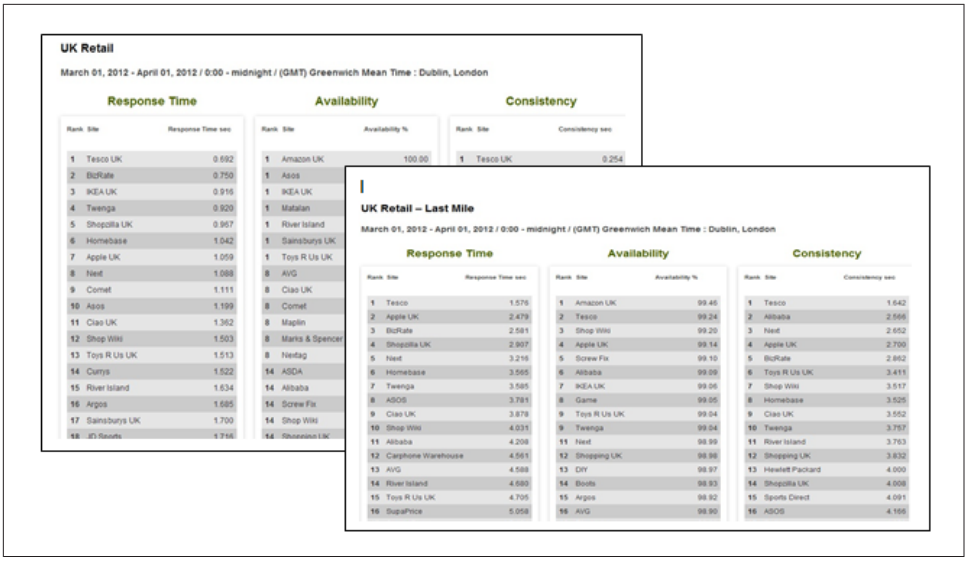


Figure 7-19. *UK retail website end-user-performance leaderboard*

Dashboard displays

Operational dashboard—RAG display.

Figure 7-20 is a heads-up display of key transaction performance across a range of end-user bandwidths, as used by a major UK retailer.

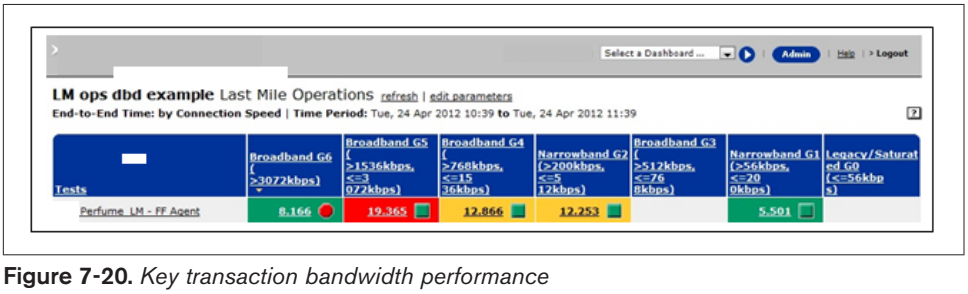


Figure 7-20. *Key transaction bandwidth performance*

Visitor performance map

Heads-up display of aggregate visitor satisfaction (performance versus user-defined thresholds) for satisfactory (green), tolerating (amber), and frustrating (red).

Global view (illustrated in Figure 7-21)—site performance by country. Such displays are particularly suited to the display of visitor-based passive (RUM) data. All major vendors now supply such a view as standard within their products.

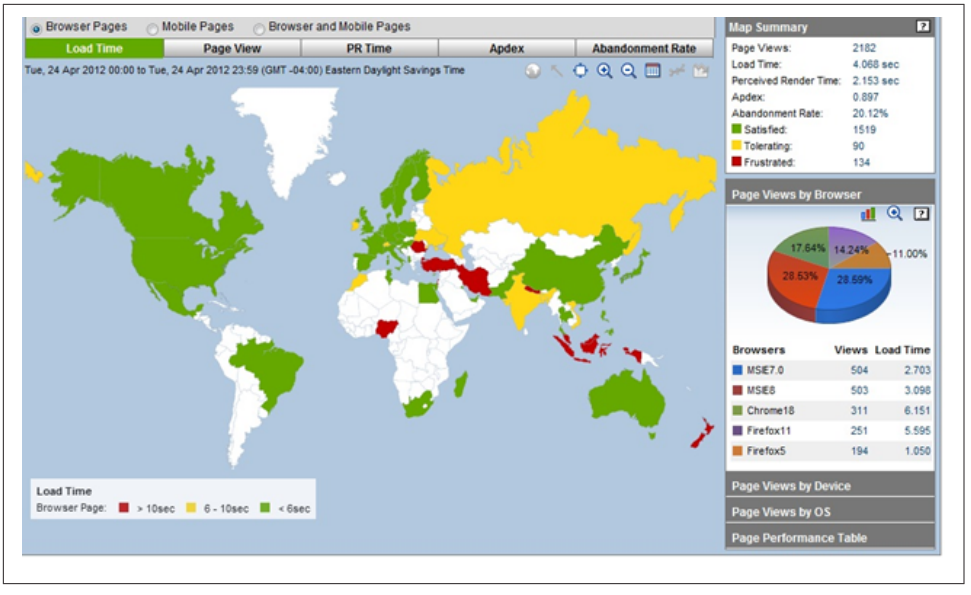


Figure 7-21. *Performance trend—24hr*

Figure 7-22 shows performance over a rolling 24-hour period, illustrating the relationship between site traffic and performance. The tabular display below it shows metrics based on pages, originating ISP, visitor browser/version, and geographic location. It is color-coded to show the trend versus the previous day or week.

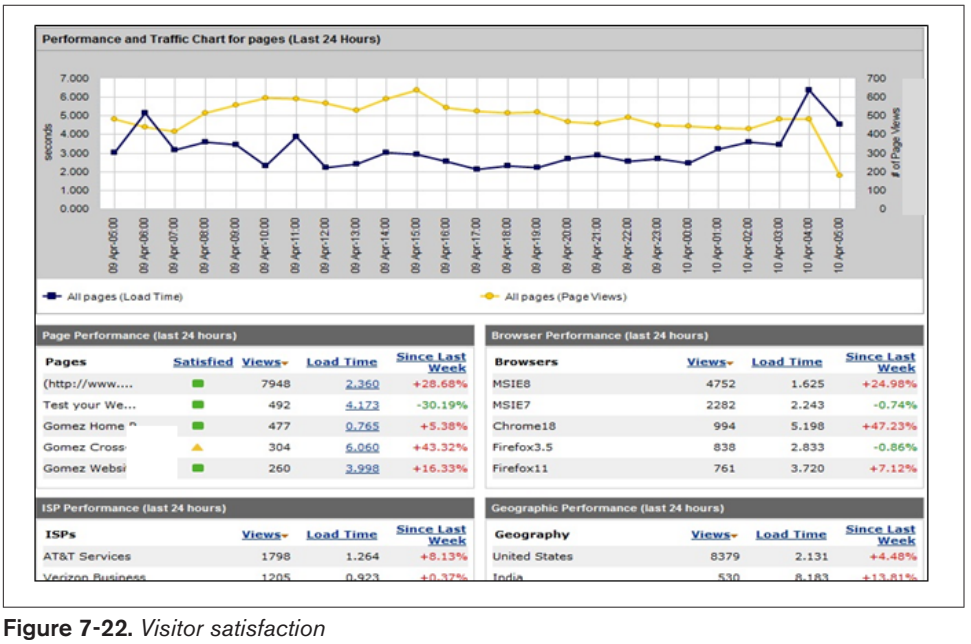


Figure 7-22. *Visitor satisfaction*

The stacked bar chart shown in Figure 7-23 displays the relationship between (proportion of) customer satisfaction (satisfactory/tolerating/frustrated—the APDEX adopted by many vendors in recent years) versus overall site traffic. Such a display is useful for highlighting traffic-related performance issues.

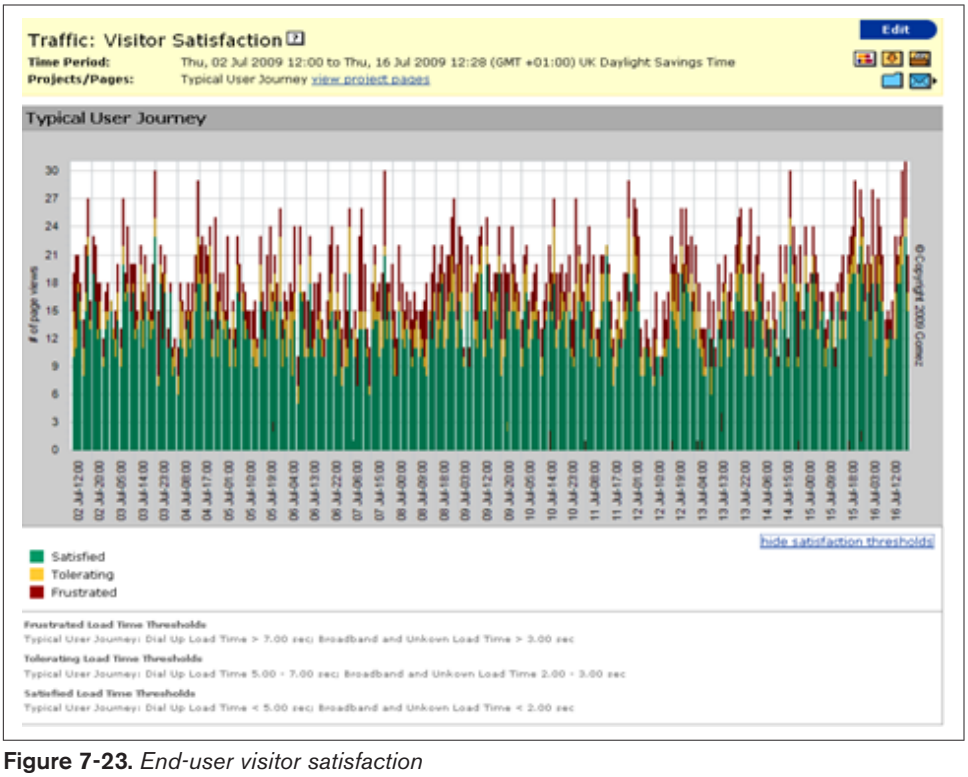


Figure 7-23. *End-user visitor satisfaction*

Alerting

The core issue with alerting is not whether a given tool will generate alerts, as anything sensible certainly will. Rather, the central problem is what could be termed the *actionability* of the alerts generated. Failure to flag issues related to poor performance is clearly deficient, but unfortunately over-alerting has the same effect, as such alerts will rapidly be ignored.

Effective alert definition hinges on the determination of normal performance. Simplistically, you can determine this by testing across a business cycle (ideally, a minimum of three to four weeks). That approach is fine provided that performance is reasonably stable. However, that is often not the case, particularly for applications experiencing large fluctuations in demand at different times of the day, week, or year.

In such cases (which are extremely common), the question becomes at which point of the demand cycle should you base your alert threshold? Too low, and your system is simply telling you that it's lunchtime (or the weekend, or whenever greatest demand occurs). Too high, and you will miss issues occurring during periods of lower demand. There are several approaches to this difficulty, of varying degrees of elegance:

- Select tooling incorporating a sophisticated baseline algorithm that's capable of applying alert thresholds dynamically based on time of day, week, month. Surprisingly, many major tools use extremely simplistic baseline models, but some (e.g., App Dynamics APM) certainly have an approach that assists. When you're selecting tooling, you'll find it worthwhile to investigate this area.
- Set up independent parallel (active monitoring) tests separated by maintenance windows, with different alert thresholds applied depending upon when they are run. This is a messy approach that comes with its own problems.
- Look for proxies other than pure performance as alert metrics. Using this approach, you set a catch-all performance threshold that is manifestly poor regardless of when it is generated. This is supplemented by alerting based upon other factors that flag delivery issues. Examples include the following:
  - Payload—that is, error pages or partial downloads will have lower byte counts. Redirect failures (e.g., to mobile devices) will have higher-than-expected page weights.
  - Number of objects.
  - Specific flag objects.
- Ensure confirmation before triggering an alert. Some tooling will automatically generate confirmatory repeat testing, whereas some will enable triggers to be based on a specified number or percentage of total node results.
- Gotchas—make sure to account for these. Good test design—for example, controlling the bandwidth of end-user testing to screen out results based on low-connectivity tests—will improve the reliability of both alerts and results generally.

Passive (RUM)-based alerting presents its own difficulties. Because it is based on visitor traffic, alert triggers based on a certain percentage of outliers may become distorted in very low-traffic conditions. For example, a single long delivery time in a 10-minute time slot where there are only 4 other normal visits would represent 20 percent of total traffic, whereas the same outlier recorded during a peak business period with 200 normal results is less than 1 percent of the total. Passive (RUM) tooling that enables alert thresholds to be modified based on traffic is advantageous. While it does not address the normal variation issue, replacing binary trigger thresholds with dynamic ones (i.e., an alert state exists when the page/transaction slows by more than *x*% compared to its average over the past period) can sometimes be useful. Some form of trend

state messaging (that is, condition worsening/improving) subsequent to initial alerting can serve to mitigate the amount of physical and emotional energy invoked by simple fire alarm alerting, particularly in the middle of the night.

A few final thoughts on alerts post-generation. The more evolved alert management systems will permit conditional escalation of alerts—that is, alert group A first, then inform group B if the condition persists/worsens. Systems allowing custom-coding around alerts (such as Neustar) are useful here, as are the specific third-party alert-handling systems available. If using tooling that permits only basic alerting, you might consider integration with external alerting, either of the standalone service type, or (in larger corporations) integrated with central infrastructure management software. Lastly, as far as delivery mode, email is the basis for many systems. It is tempting to regard SMS texting as beneficial, particularly in extreme cases. However, as anyone who has sent a celebratory text on New Year's Eve only to have it show up 12 hours later can attest, such store-and-forward systems can be false friends.

Gotchas!

Arguably, active (synthetic) external monitoring, to a greater extent than related disciplines such as application performance or passive (RUM) monitoring, is prone to errors of interpretation. These typically stem either from poor test design (not framing the question with sufficient rigor) or from failure to control the quality of the data obtained. However, as I have reiterated several times, clean-room testing, while necessary, is insufficient within a comprehensive performance management schema designed to maximize end-user satisfaction. Unfortunately, the more you try to replicate end-user conditions, the more dangers arise in capturing and reporting erroneous findings. The following list summarizes some of the areas for which it is important to check and control:

Target destination errors
: You know what you entered as a test target. However, redirects, whether intentional or accidental, can lead to your testing something other than the intended page. Typical examples are the delivery of a PC site to a mobile device, or the delivery of a partial or error page instead of the complete version. Validate tests (particularly if abnormally fast or slow) by examining the size of the page downloaded and/or the number of objects. If available, consider using screen capture functionality to confirm.

Failure to discriminate between "fatal" errors
: Fatal errors (i.e., those causing a page to fail, as opposed to individual content failures) are reflected as lack of availability in active testing. Always examine the point at which a page failed (i.e., how many objects successfully loaded). In addition to often providing insight into the nature of the root cause, such investigation

will often indicate whether the error is of one consistent type or a mixture (although the latter may still have a common cause).

Latency
: Ensure that aberrant results are not due to round-trip latency (e.g., from poor ISP node selection or inadvertent inclusion of end-user peers using distant proxy DNS servers).

Peerage
: Group and compare results by ISP. Although there will always be some variation, systemic problems will be obvious if results over an extended period are compared. Filter and compare end-user results by tertiary ISP (and/or set up confirmatory tests) if in doubt.

Banding
: Average traces (while they have their uses) can hide a multitude of detail. Always examine a scattergram trace of individual results to identify patterns of outliers or the delivery of banded clusters of results. Further examination at object level may provide further insight. Banding is commonly an indication of poor load-balancer configuration.

Browser version
: Passive (RUM) data can be useful in identifying performance differences between browsers or versions—provided the browser concerned is recorded by your passive (RUM) tooling. Beware of reliance on active testing from a single browser agent without cross-validation.

Device and/or PC client limitations
: Differences in processor speed, operating system, signal strength, and memory state (and more) can affect recorded performance.

Native versus third-party domains
: Third-party affiliates can have significant and unpredictable effects on performance. Always examine at individual object level, plot suspect objects individually, and/or run confirmatory tests with object filtering.

Connection speed
: Particularly when testing from end users (especially if using wireless carriers), you absolutely must filter out any very low connectivity tests (and/or clearly specify test conditions). Very low connectivity conditions are extremely distortive. They can lead you to draw incorrect inferences despite them being, in fact, a correct representation of performance in those conditions.

Summary

Now that we've covered in some detail the importance of understanding performance from the end-user's perspective, the next chapter looks at how to integrate external monitoring with static performance testing.

CHAPTER EIGHT

Integrating External Monitoring and Performance Testing

It's performance, Jim, but not as we know it.

—Anonymous

AFTER THE EXPLANATION OF EXTERNAL MONITORING IN CHAPTER 7, THIS CHAP- ter continues the discussion by focusing on how to integrate external monitoring with static performance testing. The development and increasing uptake of new frontend (i.e., client-side) technologies represent a challenge to traditional models of performance testing. It is becoming ever more important to ensure end-user performance, rather than continue to rely on infrastructure-based, thin-client testing models. Examples are many, but include the following:

Comet implementations

Comet interactions involve the use of persistent, server-initiated connections and data transfer. They have the effect of breaking the traditional GET-POST/discrete page-based model of web applications. Therefore, pages do not unload, marking completion of delivery, but rather continue indefinitely while data (e.g., live chat, sport results, new tickers) continues to be delivered. *Comet* is a generic term for such connections. Several key variants exist, all of which have a common underlying philosophy (designed to reduce delivery latency). One example is HTTP long polling (aka server push). This can be initiated either directly (typically using JavaScript tagging), or via proxy server (to facilitate cross-domain delivery). Among others, HTML5 has a defined method (specified within the WebSocket API) for creating persistent server push connections. Other, nonapproved (i.e., non-W3C-standard compliant) web socket transport layers may be used. These may or may not be possible to script for outside standard browsers.

Multimedia stream delivery

Again, a number of variants exist. As an example, the Real Time Message Protocol (RTMP), which is used for streaming in Adobe Flash, includes native, encapsulated, or secure (SSL-based) options.

These and other rich Internet application (RIA) technologies (e.g., Ajax, Flex) in addition to potentially changing the individual object/page delivery paradigm, serve to leverage increasing power on the client side—for both PC and (increasingly) mobile devices, especially through the availability of relatively high-system-capacity tablet devices. In many retail ecommerce applications, these are coming to represent 40 percent or more of digital channel revenue.

Therefore, the ability to test from, and to effectively script for and around, such content (e.g., bracketing particular calls to identify the source of application vulnerabilities) is increasingly relevant.

Other confounding variables in real-world delivery that are relevant to thick-client performance testing include the use of external performance interventions, particularly CDNs, that have the potential to distort test interpretation (unless excluded), and the introduction of responsive applications using HTML 5/CSS3 as replacements to earlier multidevice application approaches. The latter will deliver different content depending on the nature of the requesting end-user PC browser or device. Such responsive delivery also applies to the server-initiated interactions previously discussed. All in all, they represent an inherent lack of consistency that can undermine the basic steady-state comparison assumption that underpins interpretation of performance tests.

It should be noted that these, and other noncommunication developments such as Adobe expressive web CSS extensions (regions, shapes), as new technologies do not have full browser version support. You must take care to match end-user test traffic browsers when designing tests of applications incorporating all such innovations. For example, the memory-intensive nature of rich browser testing severely limits the number of such tests you can run from a single machine. Performance distortions due to such system constraints may be introduced even though the tests run with apparent success. A one-to-one model, such as is offered by some of the vendor solutions employing distributed users, may be advantageous, although the countervailing issue in such cases is the need to ensure similarity of system performance across the test peers. Regardless of these limitations, you can gain considerable insight into the performance of modern applications through rich client testing (at least as a component of the overall test set). The incorporation of external monitoring/end-user response into performance load testing can take one of two forms, which I have termed *integral* or *combined*. They are briefly discussed next.

Tooling Choices

There are a number of products that will generate all or a proportion of load traffic from real browsers. In theory, at least, this should provide benefits by enabling you to manage larger and/or geographically distributed rich browser traffic. Depending upon the particular product, such insights may include the ability to accurately associate external with internal event timing, drill down into the performance of individual client-site components (e.g., JavaScript, CSS, and third-party affiliates) under different load conditions, and, in certain cases, segment end-user performance by other end-user-relevant metrics (e.g., geographic location, tertiary ISP).

While no recommendation of particular products is implied by what follows, those of you for whom frontend optimization is important (and that should be everyone!) may benefit from some initial pointers into the marketplace. A number of experienced performance consultancies exist (at least in the UK and some European countries). These are often well placed to advise and support the tool evaluation process or, indeed, undertake such performance testing as required. It is to be anticipated that solutions offering visibility into and support for in-client application processing will proliferate, driven by the need to effectively test these aspects as part of a comprehensive performance assurance strategy. At the time of writing (early 2014), tooling options include the following:

Compuware Gomez
: The originator of end-user-based performance testing, the Gomez SaaS solution (now known as Compuware APMaaS) has evolved in the five or more years since its inception to include the ability to run hybrid tests combining cloud-based, thin-client HTTP traffic with real-user PC browser traffic derived from the vendors' global last-mile network of private end-user PCs. This option is limited to true HTTP/HTTPS-based applications. Mobile end-user traffic can be included, from a number of wireless carrier test nodes around the world. However, note that such mobile traffic, while it has the advantage of true wireless carrier connectivity, is not derived from actual mobile devices, but rather is simulated through the introduction of device headers based on a standard Mozilla or WebKit browser engine. The Gomez SaaS platform is designed to be self-service, which may be advantageous in some circumstances (although it may be unrealistic for high-volume and/or complex tests). One other advantage is conferred through the script recorder, which facilitates the filtering of inappropriate content during testing.

TestPlant eggPlant/Eggon TestPlant (formerly Facilita Forecast)
: This option offers the ability to integrate real-device traffic into load tests, generated via its performance load tooling or otherwise. The ability to link and analyze native mobile applications (using the eggOn VNC) and to associate performance

issues with device-related state (e.g., memory or battery availability) is often beneficial.

SOASTA CloudTest/Touchtest
: Initially designed as a functional test tool for mobile devices, Touchtest also captures some system metrics, and can be integrated with its CloudTest performance-load offering.

Neustar
: The rich PC client–based load-testing product offered by this vendor has been evolved from Browser Mob, an open source tool.

Keynote/Keynote systems
: Keynote, a long-established vendor in this space, has a number of potentially relevant offerings that you may wish to consider.

Your selection will depend on a number of factors, including the following:

- Volumes required
- Relevant geographic perspectives
- Ability to script the application transactions
- Ability to filter inappropriate components (e.g., CDN cached content)
- Understanding/control of the system characteristics (OS, memory, processor speed, battery state)
- Ease of use/skills required
- Cost

Active and Passive Integration with Static Performance Testing

Provided that the application under test can be reasonably effectively performance tested using traditional thin-client/HTTP parameterization/core infrastructure models—that is, that it is not necessary to replicate the client-side coding in order to create meaningful user transactions—then an alternative may be to run traditional thin-client performance load testing, but to supplement it with end-user response visibility derived from traffic generated during the test. However, simply running end-user monitoring in parallel with performance testing will not usually provide sufficient granularity to help you isolate and resolve an issue as a test output. In theory, it is possible to run testing on the production environment during trading periods. In theory, any negative end-user effects would be recorded by passive (RUM) tooling.

However, this approach is limited by the shortcomings of most passive (RUM) tooling in terms of browser support and object-level data capture. A further limitation is that the normal user traffic will provide a variable base load on the application. This makes the interpretation of results, and, in particular, the running of replica tests extremely difficult.

Provided that the load generated is small in comparison to the base load profile, you can, however, obtain useful data by running prescheduled synthetic end-user tests in parallel with the load test. Although timestamp correlation can present difficulty (the browser-based timings not being exactly synchronized with server timings), such an approach can be beneficial, and certainly has advantages over not having any end-user performance visibility.

Such testing should be run from as close to true end-user conditions as possible. Assuming that it is practicable to install the monitoring agent onto a range of suitable user machines (e.g., in the case of closed intranet systems or possibly applications, such as some major retailers and gaming companies, that have access to a customer test panel), then the private peer options offered by many monitoring vendors may be utilized to good effect.

The first caveat is that the bulk of the following discussions relates specifically to the performance testing of Internet-based applications. Obtaining end-user metrics from closed client-server applications is equally important but more challenging, and I will discuss this briefly at the end of this section.

As we have seen in the previous chapter, two types of external visibility are possible: that from active (scheduled, synthetic) and passive (visitor or RUM) tooling. Both have value as a component of effective performance testing, although care is required in test design and interpretation. The fundamental issue of synchronized timestamping has already been mentioned. To further clarify: end-user data (of whatever kind) will usually report browser-derived timings. These will differ from infrastructure-based timestamps. Usually, your understanding that there will not be an exact correlation is all that is required, although these differences sometimes present difficulties in accurate issue isolation, particularly in the case of sporadic/episodic errors. Luckily, most of the issues derived from performance testing are not of this nature, but rather reflect reaching of thresholds or uncovering other capacity gating factors.

Passive and Performance Testing

Passive real-user monitoring (RUM) data is of considerable value as part of an effective production-quality assurance program. However, the general value to performance testing may be limited, for a number of reasons. Consider the essence of passive RUM —the secret is in the name—it is real-user (visitor) monitoring. Although all traffic will, of course, be detected and analyzed, how much value is in knowing the end-user

metrics related to 1,000 identical thin-client users all delivered from a narrow IP range on the Internet backbone in one (or a few) locations?

If your performance test model includes full-browser-originated traffic and/or mobile device users, more value will likely be obtained. The use of such extensions to the performance test model is often highly beneficial, particularly given the extent of client-side application logic in modern applications.

The greatest value of passive (RUM) data in the practice of performance testing lies in the particular case where the load testing is undertaken against a live production system. The risk of introducing performance deficits during testing, with a knock-on revenue effect, deters most from this approach. If, however, system characteristics are well known from earlier development and preproduction testing, and the testing is effectively business as usual (BAU) rather than stress based (for example, replicating anticipated traffic from a sale or marketing initiative), the value in accessing performance impact data across all live users may be substantial.

You still need to keep in mind the limitations of the particular passive (RUM) tooling employed, but you cannot deny the value of being able to examine distributed effects to end users of increased system load. The core performance test principles apply: in particular, to understand *normal* by baselining across a complete business cycle beforehand. In this case, a detailed appreciation not only of generalized end-user performance, but also of granular metrics by user type, is valuable. Such benefit is well worth the effort involved, as it often serves to highlight areas that will repay tuning or otherwise inform the marketing approach or design of the retail offer.

Depending on the passive (RUM) tooling available, you may wish to enrich raw performance metrics with data from allied technologies such as web analytics. Table 8-1 provides a list of metrics that may prove valuable to the business both during and outside performance testing.

Table 8-1. *Suggested passive (RUM) metrics*

Page response (total load, perceived render [browser fill], DOM ready-time by)
Geography (region, country, area)
Browser type and version
Mobile device/operating system
Connectivity (ISP, hardwired/wireless, bandwidth)
Conversion rate
Abandonment (shopping cart, key transactions)
Bounce rate (key landing pages—e.g., SEO destinations)

Stickiness (total time on site)

Table 8-1 is not exhaustive, but regular recording and trending against other key metrics (site traffic, online revenue) provides the basis of a true business-facing monitoring and testing regime. It should also provide the basis of KPI definition and iterative goal setting as part of a continuous performance improvement strategy.

Once defined, changes to key metrics during performance testing are simply flagged. At Intechnica, we find that RAG-based standard reports provide a rapid and comprehensible way of interpreting test outputs.

RUM and APM

The advantages of application performance monitoring (APM) tooling are discussed elsewhere in the book. In essence, it provides visibility across the tiers of the delivery infrastructure, thus giving us insight into the cause of performance issues highlighted by load testing. Almost all prominent vendors now offer a passive (RUM) extension to their core product, thus giving more or less end-to-end visibility—that is, from browser or mobile device through to the originating application or database server.

Integration of Active Test Traffic with APM Tooling

You can gain an additional advantage in terms of root-cause investigation of issues arising during or following load testing by identifying and integrating the active (synthetic) test traffic with session-based activity within the delivery infrastructure. Some APM tooling will support this analysis, which is valuable for examining not only the end-user traffic, but also the core load requests. You typically achieve such integration by placing a flag in the header of the synthetic test pages, as you can see in Figure 8-1.

Figure 8-1. *Inserting a header flag to identify synthetic test traffic to an APM tool (in this example Gomez SaaS/Compuware dynaTrace)*

The more sophisticated and complex passive (RUM) products will capture specific individual session data. This takes away the requirement to infer causation by comparing

passive (RUM) and infrastructure metrics, by providing an absolute link between these elements.

Active External Monitoring and Performance Testing

So passive (RUM) monitoring has its place as part of an effective performance testing strategy, although that place may be somewhat niche. The role of active (synthetic) monitoring is somewhat different. As discussed previously, among the key advantages of active monitoring are the ability to record application availability and compare scheduled recurrent tests from known conditions/locations. Furthermore, testing from browser clients and mobile devices, simulated or real, provides detailed visibility into end-user performance of the system under test in the absence of real visitors (i.e., in the common situation of testing preproduction environments or production environments with little or no native traffic).

A distinction should be made between systems that are using end-user devices to generate the load itself (examples include Browser Mob and Gomez SaaS) and the more common situation where end-user monitoring is used to extend the understanding from application performance while under load generated from elsewhere (typically, within the delivery infrastructure or from external ISP cloud-based locations).

In the case of the former, where the base load is (or may be) generated from end-user locations, considerations are similar to the load and monitoring scenario. Some detailed provisions should be made. As an example, it is advisable to filter test scripts at object level to remove components that are subject to CDN acceleration. To quote an ex-colleague, there is little advantage in attempting to boil the ocean, and it also avoids irate calls to the operations center from CDN providers concerned about DDoS (distributed denial of service) attacks. CDN performance assurance is better undertaken on an ongoing, live-system basis.

Test Approach

As with testing in general, we must begin here by defining an effective test matrix. In the case of external monitoring, this essentially consists of defining a business-realistic series of external tests to supplement the core performance load test. The test set should map as closely as possible to the distribution of actual (or anticipated) application usage. Basic considerations should include the following:

- Geographic location
- Script pack mapping
- PC system characteristics
- Browser(s), versions

- Mobile devices
- Connectivity—wireless networks should always be used for mobile device testing; define bandwidth closely (or filter results) prior to interpretation of results

Depending upon the detailed nature of the application under test, it may be desirable to further subdivide the testing (e.g., setting up clusters of test peers by specific tertiary ISPs), although in practice it is usually sufficient to use results filtering for comparison, provided that sufficiently large test clusters are used.

Apart from economic considerations it is important to ensure that there is not a high disparity between the end-user traffic and the volumes of the performance test profile to avoid distortion of the results. In other words, avoid mixing very high levels of active (synthetic) end user monitoring traffic with low-volume performance tests.

Now that we have defined the monitoring scenario in terms of end-user devices and locations, standard monitoring principles apply. Use-case scripts should be created to mirror those used by the performance test tool.

Test Scheduling

End-user monitoring should commence considerably before the anticipated performance load test. In an ideal situation, external testing should be ongoing anyway. A period of two to three weeks is usually regarded as the minimum for effective baselining, although a week or so may suffice in the case of applications with very predictable usage patterns. Having established baseline values, end-user monitoring continues throughout the load test, and for an extended period afterward. This approach will detect any systemic deterioration (for example, due to memory leakage).

Figure 8-2 shows end-user performance pre-, per-, and post-performance test.

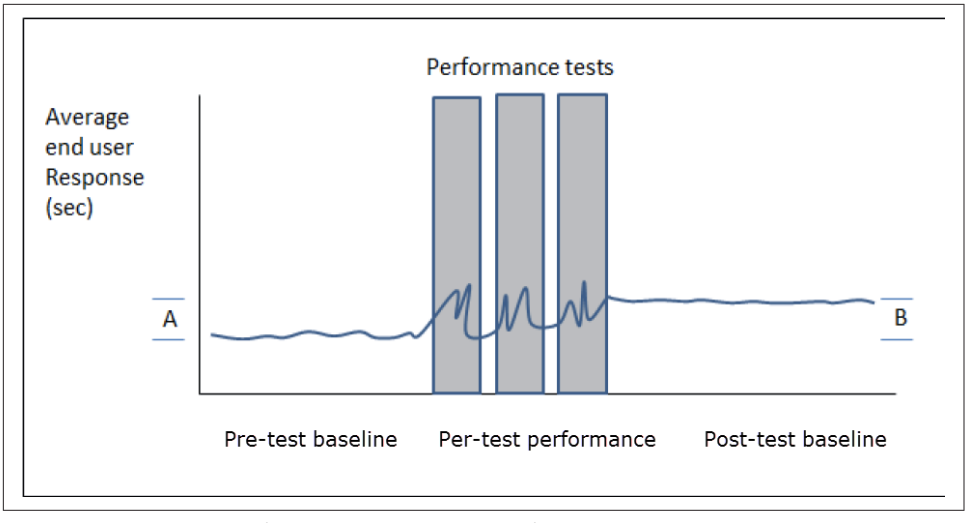


Figure 8-2. *End-user performance pre/per/post-performance testing*

A represents the impact on end-user response (in practice, by page, transaction, bandwidth, location, etc.) during the load tests. Response times should fall within agreed-upon KPI thresholds. *B* represents the difference (if any) between pre- and post-baseline values. If present, it should be investigated and the underlying issue resolved.

Figure 8-3 is an example taken from an end-user-based load test. The agreed-upon upper page response threshold for the test transaction was 40 seconds. Note the variation by geography and tertiary ISP (inset).

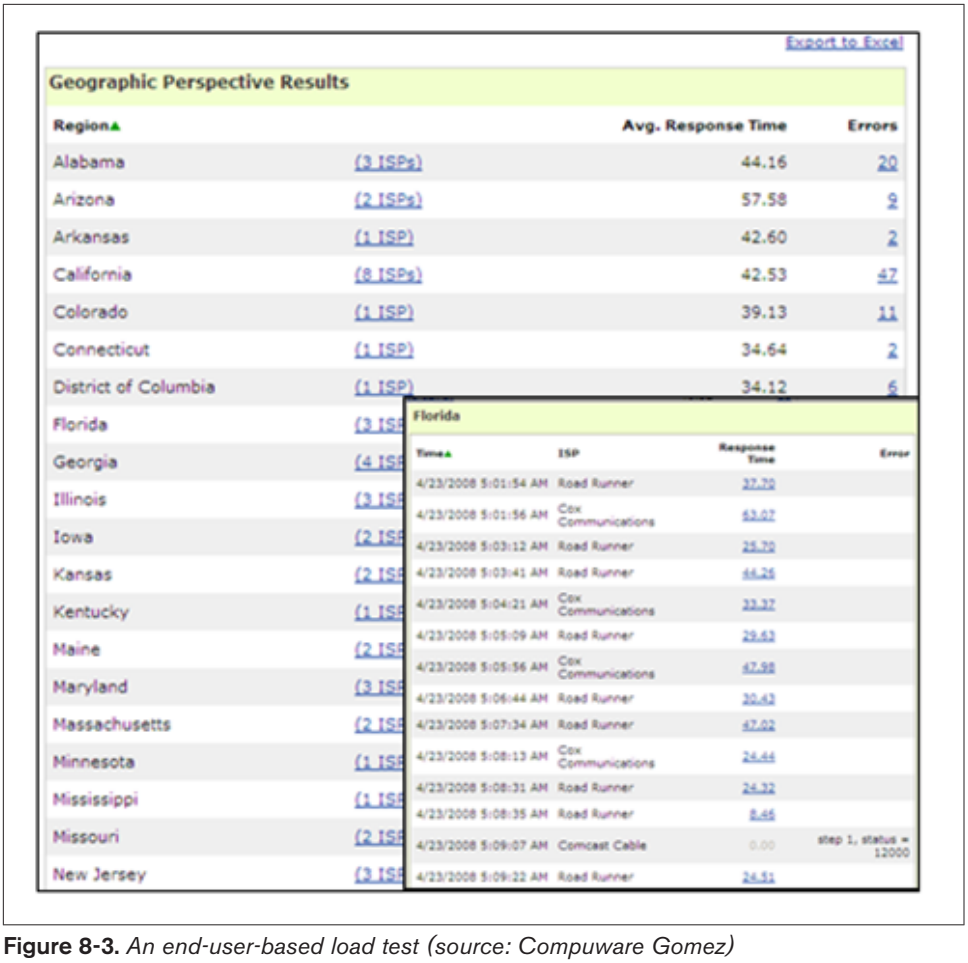


Figure 8-3. *An end-user-based load test (source: Compuware Gomez)*

Performance Testing of Multimedia Content

The progressive development of client-side system capacity as well as with the use of content caching and the evolution of adaptive streaming, has acted together with marketing's demand for compelling site content to drive the inclusion of multimedia content. Such content may be core to the site or provided by third parties.

Regardless of the origin of such content, it is important to understand any capacity limitations as part of performance testing. In terms of multimedia stream response, the following KPI metrics are typically considered:

- Availability
- Startup time

- Rebuffer time

The availability metric is essentially the same as that for individual pages: what proportion of requests for the content are successful? The importance of initial startup time (latency) and rebuffer time (essentially, quality of delivery) is dependent upon the nature of the content. Users of destination content (e.g., a recording of the Olympic 100m final) will typically be more tolerant of high startup time than for opportunist content (such as personalized advertising delivered to an ecommerce site).

The test approach is the same as that adopted for other active external testing used in concert with performance testing. The tooling choice (which, at the time of writing, is rather limited) should consider support for the following:

- The nature of the stream (e.g., *.swf*, *.mp3*)
- The specific codex used
- Support for responsive delivery (e.g., adaptive streaming in conditions of limited connectivity)

Multimedia performance testing is one use case where the end user (thick client/mobile device) has a significant benefit. To avoid "boiling the ocean"-type testing, where multimedia content is delivered via CDN, test managers may wish to test the origin only (blocking CDN hosts from their test scripts), and/or exclude multimedia content from stress testing, including it for the final BAU test phases.

End-User Understanding in Non-Internet Application Performance Tests

As we have seen, there is a large body of tooling available for monitoring the end-user performance of Internet-based applications, as part of performance load testing or otherwise. However, understanding and managing the end-user experience of non-Internet applications is equally important. Approaches fall into two categories: point-to-point testing and non-Internet APM:

Point-to-point testing
: Point-to-point testing (also known as *transaction trace analysis*) is based on test execution from an end-user machine, with subsequent analysis of the data generated. Such analysis tends to be network-centric, but it can provide useful insight into key gating factors inhibiting the effective delivery of end-user requests. Scripted transactions may be executed on a single or multiple recurrent basis. The latter is more relevant to data capture during performance load testing of the base infrastructure. Provided that your tooling is able to script against the target application, the key limitation of such testing (which, in many ways, is analogous to the use of active end-user testing of web applications, as discussed earlier), is that such test-

ing is undertaken from one or more predefined individual end users rather than on a distributed basis. This presents a number of potential problems, among them are the following:

Test point selection
: Begin by mapping (or gaining access to) a schematic of the network supporting the application and the distribution of end users across it. The test locations chosen should fairly represent performance across the entire user base. Any users or user groups regularly complaining of poor performance, particularly during periods of peak usage, should also be tested (and potentially flagged as a subgroup).

Pinch point saturation
: Take care in designing the end user-test traffic that it does not generate an unreasonably high amount of usage across points in the network with known constraints (e.g., limited WAN connections to distant offices).

Test commissioning and execution
: One of the more difficult aspects of seeking to use individual point-to-point test tools to provide end-user insight as part of a performance load test is the practical difficulty of coordination, potentially across a globally distributed user base. Sufficient time and resources should be allocated to this at the test planning stage.

Non-Internet APM tooling
: An alternative to scheduled, recurrent point-to-point analysis is to use one of the application performance management tools that provides insight into the application traffic across the supporting delivery infrastructure. Ideally, this should include visibility into end-user performance, both individually and aggregated by location or department. The choice of tool will critically depend upon the decodes available for the application in question. As an example, Compuware's long-standing data center RUM product provided agentless monitoring (via port spans or taps at the boundary of each infrastructure tier, together with a broad range of decodes—for examples, see Figures 8-4 and 8-5). Agentless monitoring is designed to deliver granular performance understanding without the performance overhead associated with agent-based approaches.

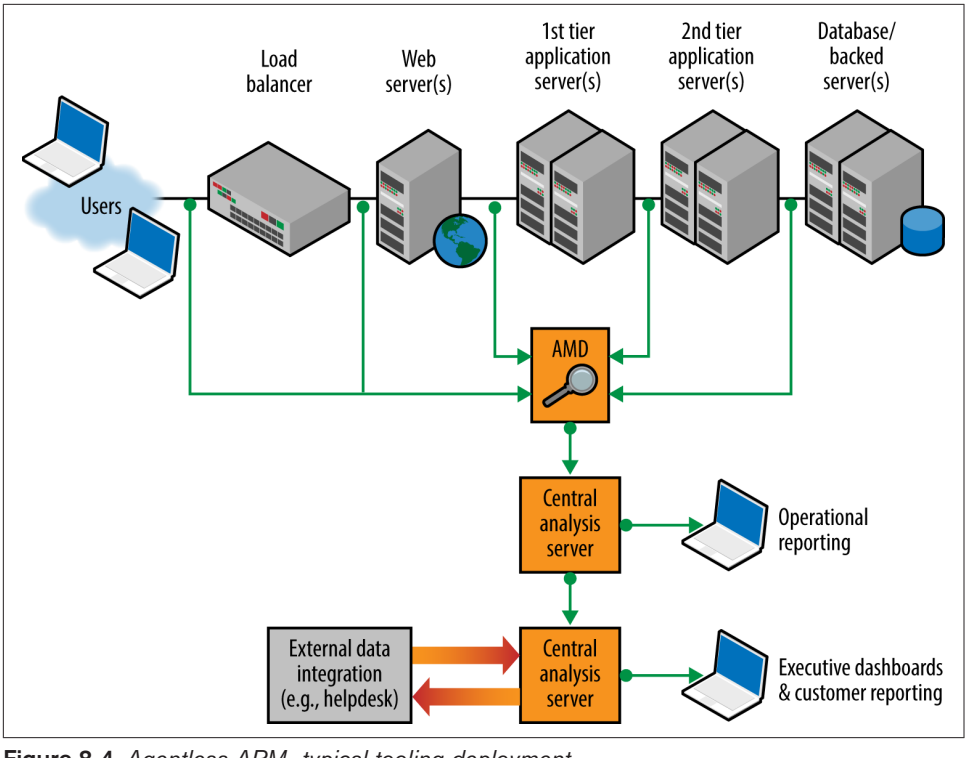


Figure 8-4. *Agentless APM—typical tooling deployment*

The following is a key checklist for performance testing of non-Internet applications:

- Map and understand system architecture, usage, and traffic patterns.
- Proof-of-concept APM instrumentation to ensure efficient decoding for specific application (e.g., SAP, Seibel).
- Instrument representative end-user machines.
- Baseline usage over a complete daily/weekly business cycle.
- Replicate test scripts between end-user and load scenarios.
- Ensure effective timestamping/synchronization.
- Seek to replicate errors identified.

Figure 8-5. *Specific decodes—vendor-specific example (source: Compuware DC RUM)*

Useful Source Materials

If you are interested in exploring end-user monitoring further, the following resources may prove helpful.

Web

- W3C standards
- APDEX

Books

- *High Performance Websites* by Steve Souders (O'Reilly)
- *Even Faster Web Sites* by Steve Souders (O'Reilly)
- *High Performance Browser Networking* by Ilya Grigorik (O'Reilly)
- *Complete Web Monitoring* by Alistair Croll and Sean Power (O'Reilly)
- *Statistics Without Tears* by Derek Rowntree (Penguin Books)

Summary

Now that we've explored end-user experience performance considerations, the next chapter looks at the impact of tech stacks on our approach to performance testing.

CHAPTER NINE

Application Technology and Its Impact on Performance Testing

IDIC: infinite diversity in infinite combinations.

—Mr. Spock, *Star Trek (1968)*

AS MENTIONED IN THE PREFACE, THE PURPOSE OF THIS BOOK IS TO PROVIDE A practical guide to application performance testing. The processes and practices described are intended to be generic; however, there are situations where the application tech stack may create additional challenges. It therefore seems appropriate to offer some guidance on how different technologies may alter your approach to performance testing. I'll try to cover all the important technologies that I've encountered over many years of performance testing projects, as well as some of the newer offerings that have emerged alongside NET.

Asynchronous Java and XML (AJAX)

This is not the Greek hero of antiquity, but rather a technology designed to make things better for the poor end users. The key word here is *asynchronous*, which refers to breaking the mold of traditional synchronous behavior where *I do something and then I have to wait for you to respond before I can continue.* Making use of AJAX in your application design removes this potential logjam and can lead to a great end-user experience. The crux of the problem is that automated test tools are by default designed to work in synchronous fashion. Traditional applications make a request and then wait for the server to respond.

In the AJAX world, a client can make a request, the server responds that it has received the request (so the client can get on with something else), and then at some random time later the *actual* response to the original request is returned. You see the problem? It becomes a challenge to match requests with the correct responses. From a programming standpoint, the performance tool must spawn a thread that waits for the

correct response while the rest of the script continues to execute. A deft scripter may be able to code around this, but it may require significant changes to a script with the accompanying headache of ongoing maintenance. The reality is that not all testers are bored developers, so it's better for the performance tool to handle this automatically. If AJAX is part of your application design, check that any performance testing tool you are evaluating can handle this kind of challenge.

TIP

When the first edition of this book came out, most performance tools found this sort of technology difficult to handle; however, asynchronous support in the current generation of tooling is generally effective and reliable. That said, it is still highly advisable to proof-of-concept a particular asynchronous requirement. I recently came across ASP .NET Signalr, which is not well supported by any of the mainstream tooling vendors.

Push Versus Pull

While we're on the subject of requests and responses, let's discuss the case where an application makes requests to the client from the server (push) rather than the client initiating a request (pull). I have seen an example of this recently in an application designed to route incoming calls to operatives in a call center. The user logs in to announce that he is ready to receive calls and then sits and waits for the calls to arrive. This scenario is also known as *publish and subscribe*. How would you go about scripting this situation using traditional automated performance testing tools? You can record the user logging in, ready to receive calls, but then there's a wait until the calls arrive. This will require some manual intervention in the script to create some sort of loop construct that waits for incoming calls. Sometimes you will have a mix of push and pull requests within the same script. If your performance testing tool doesn't handle this situation particularly well, then you must determine if the *push* content is important from a performance testing perspective. Can the server-initiated requests be safely ignored, or are they a critical part of the use-case flow? An example where you could probably ignore push content would be a ticker-tape stock-market feed to a client. It's always available, but it may not relate to any use case that's been identified and scripted.

Citrix

Citrix is a leader in the field of thin-client technology. In performance testing terms, it represents a technology layer that sits between the application client and the application user. In effect, Citrix becomes the presentation layer for any applications published to the desktop and has the effect of moving the client part of an application to the Citrix server farm. Testing considerations will revolve around ensuring that you

have enough capacity in the server farm to meet the needs of the number of virtual users that will be generated by the performance test. An important point that is often overlooked is to make sure that you have enough Citrix licenses for the load that you want to create. I have been involved in more than one Citrix performance testing engagement where this consideration (requirement!) had been overlooked, resulting in delay or cancellation of the project.

> **TIP**
>
> Citrix provides guidelines on the number of concurrent sessions per server that can be supported based on CPU type and memory specifications, among other factors.

Performance tests involving Citrix will often have validation of the server farm load balancing as a performance target. You must ensure that, when capturing your scripts and creating your load test scenarios, you are actually accessing the farm via the load balancer and not via individual servers. If there are problems, this should be pretty obvious: typically, there will be a few servers with high CPU and memory utilization while the other servers show little or no activity. What you want is a roughly even distribution of load across all the servers that make up the farm.

Figure 9-1 demonstrates Citrix load balancing under test from an actual engagement.

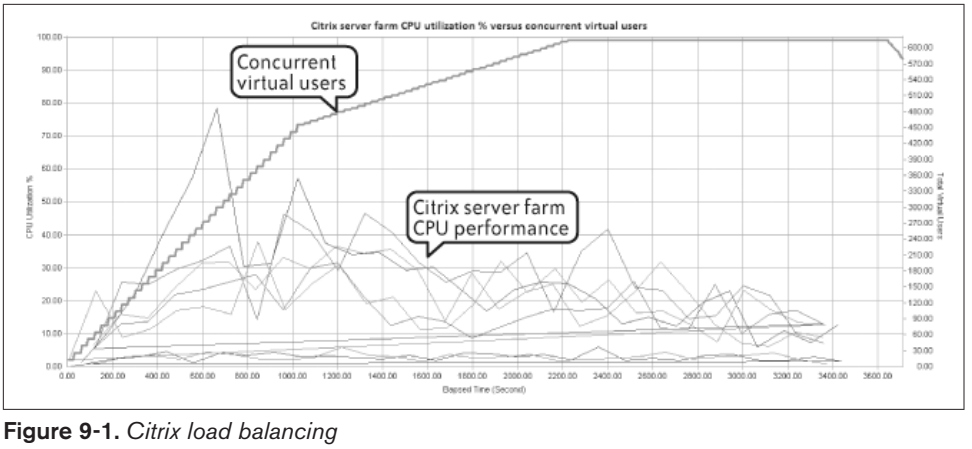


Figure 9-1. *Citrix load balancing*

Citrix Checklist

Here are some considerations to keep in mind before you use Citrix:

- Make sure that you have enough Citrix licenses for the number of virtual users required for performance testing.

- If you are performance testing a load-balanced Citrix server farm, make sure that you capture your use cases using an independent computing architecture (ICA) file; otherwise, on replay you may not correctly test the load balancing.
- Citrix connections rely on the appropriate version of the client ICA software being present. Make sure that the correct version is installed on the machine that will be recording your use cases *and* on the machines that function as load injectors. The Citrix client software is a free download from the Citrix website.
- It is good practice to increase the default amount of time that your performance tool will wait for a response from the Citrix server farm. I typically increase the default value by a factor of three or four in order to counter any delays introduced by slow Citrix responses during medium- to large-scale performance tests.
- It is also good practice to set the virtual memory page file size on each load injector machine to be three or four times total virtual memory. This is because the Citrix client was never intended to be used in the manner that it is used by automated performance test tools, with dozens of simultaneous sessions active from a single workstation or server. This means that Citrix is resource hungry, especially regarding memory; I have seen many cases where servers with many gigabytes of memory have struggled to reliably run 100 virtual users. You should anticipate requiring a greater number of higher-specification load injectors than would be the case for a simple web client application.

Citrix Scripting Advice

In my experience of scripting Citrix thin-client applications, I have learned some tips and tricks that definitely make Citrix scripting more straightforward:

- Where the application permits, access functionality using the keyboard rather than mouse or touchscreen. Hotkeys and Function keys are somewhat rarer in modern application design, but they tend to be far more reliable waypoints than mouse-driven or touch UI interaction, especially when it comes to script replay.
- Where a particular part of the UI is hard to reliably capture using image or object recognition, try tabbing from a location or control on the UI that you *can* capture and then pressing the Enter key or space bar to set focus. The tab order in many applications is somewhat random, so you may have to investigate a normalization of tab order for this to work.
- Try to minimize the number of times that you log in and log out of a Citrix session within your scripts. Citrix does not react well to frequent session connects and disconnects, so try to make this activity part of script `init` and `end` workflow rather than the `body` or `main`.

- Have someone on hand to kill off any orphan Citrix sessions that may persist after a performance test has completed. (This can normally be done only via the Citrix admin console.) This frequently happens when a Citrix VU session fails due to errors or time-out during test execution. If you leave these sessions active, your script will fail at session login unless you add code to select an already existing session. (Not recommended!)
- Finally, it is good practice to reboot your load injectors between Citrix volume tests. This removes any orphan socket connections that may persist even after a performance test has finished.

Virtual Desktop Infrastructure

While we are talking about thin clients, something relatively new to IT is the concept of a virtualized desktop infrastructure, or VDI. This differs from the Citrix model in that you maintain a library of virtual desktops that run on a centralized server farm rather than accessing applications or desktops via client-side software (you can even have Citrix within VDI). This means that end users can make do with specialist thin-client devices rather than PCs, simplifying configuration management even further.

In performance testing terms, this is yet another proprietary protocol layer between the client and the application. I am not aware of any current toolset that can deal with this technology, and I think it will be a difficult nut to crack, as the protocols sit lower in the OSI stack than ICA or RDP. Ultimately, you face the same challenges as you do with Citrix. And while it should be possible to embed the capture and load injection components within a VDI desktop, they will persist only while the VDI stream to the client device is active, so you may face licensing and connectivity issues when trying to automate script capture and playback.

I would be very interested to hear from anyone who has successfully performance tested such an environment.

HTTP Protocol

A number of unique considerations apply to applications that make use of the HTTP protocol. Rather than bundle them together under the term *Web*, the advent of .NET and SOAP/XML has broadened the scope of HTTP applications beyond that of the traditional browser-based client connecting to a web server.

Web Services

Microsoft's Web Services RFC has introduced a new challenge for automated performance testing tools. Normally (although not exclusively) making use of the HTTP pro-

tocol, web services do not necessarily require deployment as part of the web server tier, which creates a problem for those tools that make use of browser proxy server settings to record use cases. It's possible to make any application that uses web services proxy enabled, but you shouldn't assume that the customer or application developers will be prepared to make this change on demand.

The web service architecture is built around the Web Services Definition Language (WSDL) model, which exposes the web service to external consumers. This defines the web methods and the parameters you need to supply so that the web service can provide the information of interest. In order to test web services effectively, ideally you simply generate use-case scripts directly from the WSDL file; a number of tool vendors provide this capability. There are also several (free) test harnesses you can download that provide a mechanism to record web service activity, which allows you to get around the proxy server challenge.

See Appendix C for more details.

.NET Remoting

As part of. NET, Microsoft introduced the next generation of DCOM/COM+, which it termed *remoting*. Essentially this is a mechanism to build distributed applications using components that communicate over TCP or HTTP and thereby address some of the serious shortcomings of the older technology.

Like web services, remoting does not rely on web server deployment. It therefore presents the same challenges for automated performance testing tools with regard to proxy enablement.

In my experience it is normally possible to capture and script only those remoting applications that communicate over HTTP. I have had success with the TCP implementation in only a few cases (where the application use cases were extremely simple—little more than logging in to an application and then logging out). Microsoft initially recommended the TCP implementation as having superior performance over HTTP. Yet because this difference has largely disappeared in later versions of .NET, TCP implementations are uncommon.

Although it may be possible to record remoting activity at the Windows Socket (WINSOCK) level, the resulting scripts (because you are dealing with serialized objects) can be difficult to modify so that they replay correctly. This is doubly so if the application makes use of binary format data rather than SOAP/XML, which is in clear text. Such cases may require an unrealistic amount of work to create usable scripts; a better solution may be to look for alternatives, as discussed later in this chapter.

Browser Caching

It is extremely important that the default caching behavior of Internet browser applications be accurately reflected by the performance tool that you use. If caching emulation is inaccurate, then it is likely that the number of requests generated between browser client and web server will be considerably greater than would actually occur in a live environment. This will have the effect of generating a much higher throughput for a given number of virtual users. I can recall a recent example where a few hundred virtual users generated as much throughput as 6,000 real users. This may be satisfactory for a stress test when you are simply trying to find the upper limits of capacity, but it is hardly a realistic performance test.

Typical default caching behavior involves page elements such as images, stylesheets, and the like being stored locally on the client after the initial request, removing the need for the server to resend this information. Various mechanisms exist to update the cached elements, but the net result is to minimize the data presentation and network round-trips between client and web server, reducing vulnerability to congestion and latency effects and (we hope) improving the end-user experience.

Secure Sockets Layer

Many applications make use of the secure sockets layer (SSL) extension to the HTTP protocol, which adds an encryption layer to the HTTP stack. If your automated performance test tool can't handle SSL, then you won't be able to record any use cases unless you can do so from a non-SSL deployment of the application. There are two other aspects of SSL that you should be aware of:

Certificates
: One way of increasing security is to make use of client certificates, which can be installed into the application client before access is permitted to an application. This implements the full private/public key model. The certificates are provided in a number of formats (typically pfx or p12) and must be made available to your performance testing tool for successful capture and replay. If your application makes use of client certificates, check that the performance testing tool can deal with client certification; otherwise, you won't be able to record your application. Typically the certificates are imported into the testing tool prior to capture and then automatically used at appropriate points within the resulting script.

Increased overhead
: Making use of SSL increases the number of network round-trips and the load on the client and web server. This is because there is an additional authentication request sent as part of every client request as well as the need to encrypt and decrypt each network conversation. Always try to retain the SSL content within

your performance test scripts; otherwise, you may end up with misleading results due to reduced network data presentation and a lighter load on the web server(s).

Java

Over recent years it would be fair to say that IT has embraced Java as the technology of choice for distributed application server design. Products from IBM, such as WebSphere and Oracle WebLogic, dominate the market in terms of deployed solutions, and many other market leaders in packaged applications, such as SAP, provide the option of a Java-based mid-tier. In addition there are many other Java application server technologies for the software architect to choose from, including JBoss and JRun.

Java application clients can come in many forms, ranging from a pure Java fat client—which is probably the most challenging for a performance testing tool that records at the middleware level—to the typical browser client that uses the HTTP protocol to talk to the web server tier.

Because of the component-based nature of Java applications, it is vital to have access to the internal workings of the application server under load and stress. Several tools are available (see Appendix C) to provide this level of visibility, exposing components or methods that consume excessive amounts of memory and CPU as well as design inefficiencies that have a negative impact on response time and scalability.

By no means unique to Java, the following are some common problems that cannot easily be detected without detailed monitoring at the component and method level.

Memory leaks
: This is the classic situation where a component uses memory but fails to release it, gradually reducing the amount of memory available to the system. This is one reason why it's a good idea to run soak tests, because memory leaks may occur for a long time before affecting the application.

Excessive component instantiations
: There are often cases where excessive instances of a particular component are created by the application logic. Although this doesn't necessarily cause performance issues, it does lead to inefficient use of memory and excessive CPU utilization. The cause may be coding problems or improper configuration of the application server. Either way, you won't be aware of these issues unless you can inspect the application server at the component level.

Stalled threads
: This was mentioned briefly in Chapter 5. A stalled thread occurs when a component within the application server waits for a response from another internal component or (more commonly) an external call to the database or perhaps a third-party application. Often the only symptom is poor response time without any

indication of excessive memory or CPU utilization. Without visibility of the application server's internal workings, this sort of problem is difficult to isolate.

Slow SQL
: Application server components that make SQL calls can also be a source of performance issues. Calls to stored procedures will be at the mercy of code efficiency within the database. Note also that any SQL parsed on the fly by application server components may have performance issues that are not immediately obvious from casual database monitoring.

Oracle

There are two common types of Oracle application technology you will encounter. Each has its own unique challenges.

Oracle Two-Tier

Oracle two-tier applications were once the norm for a fat client that made use of the Microsoft Open Database Connectivity (ODBC) standard or, alternatively, the native Oracle Client Interface (OCI) or User Programming Interface (UPI) software libraries to connect directly to an Oracle database. The OCI interface is pretty much public domain, and a number of tool vendors can support this protocol without issue. However, the UPI protocol is an altogether different animal, and applications that make use of this technology can be extremely difficult to record. It is not uncommon to come across application clients that use a mix of both protocol stacks.

Oracle Forms Server

Current Oracle technology promotes the three-tier model, commonly making use of Oracle Forms Server (OFS). This involves a browser-based connection using the HTTP (with or without the SSL extension) to connect to a Java-based mid-tier that in turn connects to a backend (Oracle) database. This HTTP traffic starts life as normal HTML requests and then morphs into a series of HTTP POSTS, with the application traffic represented as binary data in a proprietary format.

ORACLE provides a browser plug-in called JInitiator that manages the connection to the Forms Server, which is a free download from the Oracle website.

Oracle Checklist

Here are some considerations to bear in mind before you use Oracle:

- If you are performance testing Oracle Forms, then make sure you have the correct version of JInitiator installed on the workstation where you will be recording your use cases *and* on the machines that will be acting as load injectors.

- Remember that your performance testing tool must specifically support the version of Oracle Forms that you will be using. It is not sufficient to use simple HTTP recording to create scripts that will replay correctly.
- If your application is the older Oracle two-tier model that uses the Oracle client communication, then the appropriate OCI/UPI library files must be present on the machine that will be used to capture your transactions *and* on the machines that will act as load injectors.

SAP

SAP has been a historical leader in customer relationship management (CRM) and other packaged solutions for the enterprise. SAP applications can have a traditional fat client or SAPGUI and can also be deployed using a variety of web clients.

One limitation that SAP places on performance testing tools is the relatively large resource footprint associated with the SAPGUI client. If you consider for a moment that the typical SAP user generally has only a single SAPGUI session running on her workstation, then this is entirely understandable. With an automated performance test tool we're creating literally dozens of SAPGUI sessions on a single machine, a situation that never occurs in the real world.

In order to maximize the number of virtual SAPGUI users, you should maximize the amount of RAM in each injector machine and then increase the virtual memory page file size to three to four times the amount of virtual memory. This will help, but even with 4 GB of RAM you may find it difficult to get more than 100 virtual SAPGUI users running on a single machine.

As with Citrix, you should anticipate requiring a greater number of machines to create a given SAPGUI virtual user load than for a typical web application. Happily, this is not so for the SAP web client, where you should find that injection requirements differ little from any other browser-based client application.

SAP Checklist

If you plan to use SAP, here are some pointers to keep in mind:

- Increase load injector page file size to three to four times virtual memory in order to maximize the number of virtual users per injector platform.
- Anticipate that a larger number of injector machines will be required to generate the load you are looking to simulate than would be the case for a typical browser-based application.

- If your performance testing tool makes use of the SAP Scripting API, then you will need server-side scripting to be enabled on the server and on the client where your transactions will be captured.
- To monitor SAP servers remotely, you can use either SNMP or WMI; otherwise, you will need the latest version of the SAP JCo package installed on the machine that will carry out the remote monitoring. You need to be a SAP customer or partner to have access to this software.
- Make sure that the Computing Center Management System (CCMS) is enabled on the SAP servers (and instances) that you wish to monitor; otherwise, you may fail to collect any KPI data.

Service-Oriented Architecture

Service-oriented architecture (SOA) is one of the current crop of technology buzzwords—along with Web 2.0, Information Technology Inventory Library (ITIL), information technology service management (ITSM), and others. But trying to establish clearly just what SOA means has been like trying to nail jelly to a wall. Nonetheless, the first formal standards are starting to appear, and SOA adoption has become more commonplace.

In essence, SOA redefines IT as delivering a series of services to a consumer (organization) based around business process management (BPM). A service may consist of one or more business processes that can be measured in various ways to determine the value that the service is providing to the business. You could think of a business process as something no more complicated than an end user bringing up a purchase order, but it is just as likely to be much more complex and involve data exchange and interaction between many different systems within an organization. The business processes themselves can also be measured, creating a multitiered view of how IT is delivering to the business.

Traditional applications make the encapsulation of end-user activity a pretty straightforward process, where each activity can be recorded as a series of requests and responses via a common middleware such as HTTP. With SOA, a business process can involve multiple use cases with interdependencies and a multitude of protocols and technologies that don't necessarily relate directly to end-user activity. How do you render this as a script that can be replayed?

This brings us to the concept of the service. As you are probably aware, Microsoft's web service RFC is a mechanism for requesting information either internally or from an external source via HTTP, the protocol of the Internet. It is usually simple to implement and understand, which is one of the advantages of using web services. A web service can be integrated as part of an application client and/or deployed from the

application mid-tier servers. If it's part of the client, then this has little effect on our performance testing strategy; but if it's part of the mid-tier, we'll likely be testing for concurrent service execution rather than for concurrent users. The use cases that we script will not represent users but rather an execution of one or more web service requests.

There is, of course, much more to the SOA performance testing model than web services. Other queue and message-based technologies, such as JMS and MQ, have been around for some time and, along with web services, are application technologies that just happen to be part of many SOA implementations. Each of these technologies can be tested in isolation, but the challenge is to performance test at the service and business process level, which is generally still too abstract for the current generation of automated performance testing tools.

Web 2.0

As with SOA you have probably heard the term *Web 2.0* being bandied about, but what does it mean? Well, it pretty much describes a collection of software development technologies (some new, some not so new) that Microsoft and Adobe, in particular, would like for us to use for development of the next generation of web applications:

- Asynchronous Java and XML (AJAX)
- Flex from Adobe
- Web services

Apart from Flex, I've already discussed how these technologies can affect your approach to performance testing. The challenge that Web 2.0 introduces is the use of a combination of different middleware technologies in a single application client. This is not a new concept, as a number of tool vendors have provided a multiprotocol capture facility for some time. Rather, it is the combination of .NET and Java technologies working at the component level that must be captured and correctly interpreted, taking into account any encryption or encoding that may be present.

Working at this level requires an intimate understanding of how the application works and helpful input from the developers. The output of your capture process is much more likely to be a component map or template, which provides the information to manually build a working script.

NOTE

If you're not familiar with Flex, it essentially provides an easier programming model for application developers to make use of the powerful animation and graphic capabilities that are part of Adobe Flash. Have a look at Adobe's website if you're interested in learning more.

Windows Communication Foundation and Windows Presentation Foundation

You may know Windows Communication Foundation (WCF) by its Microsoft code-name *Indigo*. WCF is a now established standard that unifies the following (Microsoft) technologies into a common framework to allow applications to communicate internally or across the wire. You might call this *Web 2.0 according to Microsoft*:

- Web services
- NET remoting
- Distributed transactions
- Message queues

I've already discussed .NET remoting and web services, and these technologies have their own challenges when it comes to performance testing. Distributed transactions refer to the component-level interaction that is an increasingly common part of application design, whereas message queues are rarely troublesome, as they tend to reside on the application mid-tier rather than the client.

Windows Presentation Foundation (WPF), on the other hand, is very much focused on the user interface making use of the new features of the Windows 7/8 operating systems and the latest version of the .NET framework (4.5 at time of writing). In performance testing terms, WPF will probably have little impact, as it underpins the underlying middleware changes introduced by Web 2.0. WCF has the most potential for challenges to scripting application use cases.

Oddball Application Technologies: Help, My Load Testing Tool Won't Record It!

There will always be applications that cannot be captured by automated performance testing tools. Frequently this is because the application vendor declines to release information about the inner workings of the application or does not provide an API that will allow an external tool to hook the application so that it can be recorded. There may also be challenges involving encryption and compression of data.

However, if you are quite certain that your performance testing tool should work with the target application technology, then read on.

Before Giving Up in Despair . . .

Before conceding defeat, try the following suggestions to eliminate possible performance tool or environment configuration issues.

On Windows operating systems, make sure you are hooking the correct executable (EXE) file

Sometimes the executable file you need to *hook* with your performance testing tool is not the one initiated by starting the target application. In these situations the application will work fine, but your performance testing tool will fail to record anything. Use the Windows Task Manager to check what executables are actually started by the application; you may find that a second executable is lurking in the background, handling all the client communication. If so, then hooking into this executable may achieve successful capture.

Proxy problems

If your performance testing tool uses proxy substitution to record HTTP traffic, then check to see that you are using the correct proxy settings. These are typically defined within your browser client and specify the IP address or hostname of the proxy server, together with the ports to be used for communication and any traffic exceptions that should bypass the proxy server. Most performance testing tools that use this approach will attempt to automatically choose the correct settings for you, but sometimes they get it wrong. You may have to manually change your browser proxy settings and then configure your performance tool appropriately. (Don't forget to record what the original settings were before you overwrite them!)

Can the .NET application be proxy-enabled?

Applications written for .NET do not need a web server to communicate, so they may not actually use the client browser settings. In most cases it is fairly simple to *proxy-enable* these types of applications, so if the customer or development section is willing, then this is a possible way forward.

Windows Firewall, IPsec, and other nasties

If you're convinced that everything is configured correctly but still capturing nothing, check to see if Windows Firewall is turned on. It will (by design) usually prevent performance testing tools from accessing the target application. If you can turn the firewall off, then your problems may well disappear.

Something else that you may run into is Internet Protocol Security (IPsec). From a technical perspective this is another form of traffic encryption (like SSL), but it works at layer 3 of the protocol stack rather than layer 4. In short, this means that an application need not be designed to make use of IPsec, which can simply be turned on between cooperating nodes (machines). However, by design IPsec is intended to *prevent* the replay of secure session traffic, hence the problem. Like Windows Firewall, IPsec can be enabled or disabled in the configuration section of Windows and in the configuration setup of your web or application server.

Antivirus software may also interfere with the capture process, and this may be more difficult to disable. It may be hardwired into the standard build of the application client machine.

Finally, check to see what other software is installed on the machine you're using to capture data. I have found, among other conflicts, that performance testing tools from different vendors installed on the same box can cause problems with the capture process. (Surely this is not by design!)

Alternatives to Capture at the Middleware Level

If all else fails, you still have a number of choices.

One, you can apologize to the customer or company, explaining that it is not possible to performance test the application, and quietly walk away.

Two, try recording use cases using a *functional* automated test tool that works at the presentation layer rather than the underlying middleware technology. These tools, as their name suggests, are designed for unit and functional testing of an application rather than for generating load, so the downside of this approach is that you can generally configure only a few virtual users per injection PC. Hence, you will need a large number of machines to create even a modest load. You will also find that combining functional and performance testing scripts in the same run is not enabled by the majority of performance tool vendors.

Another approach is to convince your customer to deploy the application over thin-client technology like Citrix. Several tool vendors provide support for the ICA protocol, and Citrix generally doesn't care what technology an application is written in. This strategy may involve the purchase of Citrix and an appropriate number of user licenses, so be prepared to provide a convincing business case for return on investment!

Manual Scripting

If no capture solution is feasible, then there is still the option of manually creating the scripts required to carry out performance testing. Some performance testing tools provide a link to popular integrated development environments (IDEs), such as Eclipse for Java and Microsoft's Visual Studio for .NET. This link allows the user to drop code directly from an application into the performance testing tool for the purpose of building scripts. Often templates are provided to make this task easier and to keep the resulting script code within the syntax required by the performance testing tool.

In most cases, manual scripting will greatly increase the time (and cost) of building scripts and will complicate the process of ongoing maintenance. However, it may be the only alternative to no performance testing at all.

Summary

This chapter has hopefully provided some useful advice for dealing with the challenges that application technology can bring to performance testing. I suspect that these challenges will always be with us, as new technologies appear and are in turn superseded by the next hot innovation. All we can do as performance testers is to try to keep ahead of the game through self-study and by encouraging tool vendors to keep their products current in terms of application tech-stack support.

CHAPTER TEN

Conclusion

This is the end, my only friend, the end.

—The Doors

SOMETHING I OMITTED FROM THE FIRST EDITION OF THIS BOOK WAS A CONCLUD-ing chapter. No one commented on its absence, but I thought in the revised edition it would be good to bring things to a close with a short look at future trends for performance testing. Certainly, understanding the end-user experience continues to be an ever-increasing focus of performance testing. The customer is king, and increasing conversions is the goal (at least for ecommerce).

More and more, the EUE is impacted by the plethora of mobile devices available to the consumer. Ironically, this makes IT's job increasingly difficult as they try to ensure a good EUE across an ever-expanding range of devices, operating systems, and browsers. I predict that the inclusion of on-device automation will become the norm for performance testing, together with the deployment of application performance monitoring software into testing environments (with some overlap into development). The greatly increased insight into and triage capability of software performance that APM provides, along with the relative ease of integration with performance test tooling, makes APM in some form a must-have for most companies.

With the move to continuous integration (CI) driving ever-increasing numbers of releases in shorter and shorter time scales, gaining early insight into performance problems is critical. Performance test tooling, in conjunction with APM, is well placed to provide this visibility so that problems can be dealt with early in the application life cycle, thereby improving the quality, and importantly, speeding up the delivery of software releases.

What will happen to performance tooling vendors is less certain. Enterprises are now far more reluctant to invest in expensive tooling when they can get much the same capability from SaaS and open source alternatives. This is a challenge facing software companies in general; however, for performance tooling shops there has been an acute

shift away from the established vendors for web testing requirements. Unless your performance testing need relates to some of the big customer relationship managers, enterprise resource planning packages (like BMC Remedy, SAP, or Oracle eBusiness Suite), or perhaps thin-client technology (like Citrix), then you really are spoiled for choice.

As a final thought, for those interested I extend an invitation to join my Performance Testers group on the popular business networking site LinkedIn. This group has been running for a number of years and has steadily grown to several thousand members. If you have a question on any aspect of performance testing, then I am sure that you will receive a helpful response. (You may even find gainful employment!) Best wishes and happy performance testing.

APPENDIX A

Use-Case Definition Example

THIS APPENDIX CONTAINS AN EXAMPLE USE-CASE DEFINITION TAKEN FROM A real performance testing project. Figure A-1 demonstrates the sort of detail you should provide for each use case to be included in your performance test. The example is based on a spreadsheet template that I have developed to capture use case information.

Ideally, you should create a similar template entry for each use case that will be part of your performance testing engagement.

Use Case 1	Bill Payment	PageViews	6

Step	Section	Action	Timing Name	Step Test Data (As Appropriate)				Step SLA	Post Step Think Time	Page View ?
				Source	File Name	Range	Volume Required	Response Time (secs)	Seconds	
1	Init	Enter online banking number, internet password and click "Continue"	Login Step 1	Data File	customers.csv		2500			1
2	Init	Enter three digits of personal access number (PAN) and click "Login"	Login Step 2	Hard coded						1
3	Body	Click "Payments & Transfer" from top tab menu	Click Payments	Client UI						1
4	Body	Click "Pay a bill" from secondary tab menu	Pay a bill	Client UI						1
5	Body	Select payee in "To" dialog	Select payee	Client UI						0
6	Body	Enter 5 Euro as transfer amount and press <ENTER>	Select To and From Accounts	Client UI						0
7	Body	Click on "Home"	Home	Client UI						1
8	End	Click on "Log out"	Log out	Client UI						1

Figure A-1. *Sample use-case definition*

Please see the following for an explanation of the template contents:

Use Case 1

Each use case should have a meaningful name such as "Bill Payment" in the example.

PageViews

This is a calculated value based on the number of web page view per use case step where this is relevant.

Step

The order that this step occurs in the use case.

Section

There are three options available:

Init

Whether this use case step is to occur only once at the start of a performance test.

Body

Whether this use case step is to be iterated during performance test execution.

End

Whether this use case step is to be executed only once at the end of a performance test.

Action

A description of the user action required for this use case step (e.g., step 1 requires the user to "Enter online banking number, an Internet password, and then click Continue").

Timing name

A label to identify the use case step during performance test execution and analysis (e.g., the timing name for step 1 is "Login Step 1").

Step test data (as appropriate)

It is a common for use case steps to require data entry. This section of the template attempts to capture information about any test data that may be required.

Source

The origin of the test data. This could be an external data file as indicated against step 1. Alternatively, it could simply be a hardcoded value or provided by the Client UI as indicated for the other use case steps.

Filename

The name of an external data file if this contains the test data required.

Range
: Any limits on the upper or lower value of the test data required (e.g., the requirement may be for a range of values between 1 and 100).

Volume required
: The number of unique test data items required (e.g., step 1 requires 2,500 unique sets of login credentials).

Step SLA
: Whether there is a service level associated with this use case step. Typically a maximum response time.

Post-step think time
: If any delay or pause should occur before execution of the next use case step.

Page view
: Whether this use case step results in one or more web page views. A useful thing to know when creating a load model if one of your performance targets is based on a certain number of page views per second.

APPENDIX B

Proof of Concept and Performance Test Quick Reference

THIS APPENDIX CONTAINS A CONVENIENT QUICK REFERENCE, DRAWN FROM EAR-lier chapters, of the steps required at each stage of planning and carrying out a proof of concept (POC), performance test execution, and performance test analysis.

The Proof of Concept

A proof of concept is an important prerequisite because it does the following (see Chapter 3 for details):

- Provides an opportunity for a technical evaluation of the performance testing tool against the target application
- Identifies scripting test data requirements
- Allows an assessment of scripting effort
- Demonstrates the capabilities of the performance testing solution against the target application

POC Checklist

You should anticipate no more than a couple of days for completion, assuming that the environment and application are available from day one.

Prerequisites

The following should be in place before you set up the POC environment:

- A written set of success or exit criteria that you and the customer have agreed on as determining the success or failure of the POC.
- Access to a standard build workstation or client platform that meets the minimum hardware and software specification for your performance testing tool or solution; this machine must have the application client *and* any supporting software installed.
- Permission to install any monitoring software that may be required into the POC test environment.
- Ideally, sole access to the application for the duration of the POC.
- Access to someone who is familiar with the application (i.e., a *power user*) and can answer your usability questions as they arise.
- Access to an expert who is familiar with the application (i.e., a developer) in case you need an explanation of how the application architecture works at a middleware level.
- A user account that will allow correct installation of the performance testing software onto the standard build workstation and access to the application client.
- At least two sets of login credentials (if relevant) for the target application.
- Two sample use cases to use as a basis for the POC: a simple *read-only* operation as well as a complex activity that updates the target data repository. These let you check that your script replay works correctly.

Process

Here are the steps involved in the POC process:

1. Record two instances of each sample use case and compare the differences between them using whatever method is most expedient. Identifying what has changed between recordings of the same activity will highlight any session data requirements that need to be addressed.
2. After identifying the input and session data requirements and any modifications needed to the scripts, ensure that each script will replay correctly in single-user

and multiuser mode. Make sure that any database updates occur as expected and that there are no errors in the replay logs for your scripts. Make certain that any modifications you have made to the scripts are free from memory leaks and other undesirable behavior.

Deliverables

Following are some guidelines for the POC deliverables:

- The output of a POC should be a go/no-go assessment of the technical suitability of your performance testing tool for successfully scripting and replaying application use cases.
- You should have identified the input and session data requirements for the sample use cases and gained insight into the likely data requirements for the performance testing project.
- You should identify any script modifications required to ensure accurate replay and assess the typical time required to script each use case.

Performance Test Execution Checklist

See Chapter 4 for an expanded version of this material.

Activity Duration Guidelines

The following are *suggested* duration guidelines for typical performance test activities:

Project scoping
: Time to complete nonfunctional requirements (NFR) capture and to produce a statement of work (SOW). Allow a day for each activity.

Scripting use cases
: For straightforward web applications, allow half a day per use case assuming an experienced technician. For anything else, increase this to a full day per use case.

Creating and validating performance test sessions or scenarios
: Typically allow one to two days' work.

Performance test execution
: Allow a minimum of five days.

Data collection (and software uninstall)
: Allow half a day.

Final analysis and reporting
: Allow at least one day.

Step 1: Pre-Engagement NFR Capture

You need this information to successfully create a project plan or statement of work:

- Deadlines available to complete performance testing, including the scheduled deployment date.
- A decision on whether to use internal or external resources to perform the tests.
- An agreed-upon test environment design. (An appropriate test environment will require longer to create than you estimate.)
- A code freeze that applies to the test environment within each testing cycle.
- A test environment that will not be affected by other user activity.
- All performance targets identified and agreed to by business stakeholders.
- The key application use cases identified, documented, and ready to script.
- A determination of which parts of use cases should be monitored separately (i.e., checkpointed).
- Identified *input*, *target*, and *session* data requirements for use cases that you select. Make sure you can create enough test data of the correct type within the timeframes of your testing project. Don't forget about data security and confidentiality.
- Performance test scenarios identified in terms of number, type, use-case content, and virtual user deployment. You should also have decided on the think time, pacing, and injection profile for each scripted use-case deployment.
- Identified and documented application, server, and network KPIs.
- Identified deliverables from the performance test in terms of a report on the test's outcome versus the agreed-upon performance targets.
- A defined procedure for submitting any performance defects discovered during testing cycles to development or the application vendor. If your plan is to carry out the performance testing in-house, then you will also need to address the following points related to the testing team:
  - Do you have a dedicated performance testing team? At a minimum you will need a project manager and enough testing personnel (rarely are more than two needed) to handle the scale of the project. See Figure B-1.
  - Does the team have the tools and resources it needs to performance test effectively?
  - Are all team members adequately trained in the testing tools to be used?

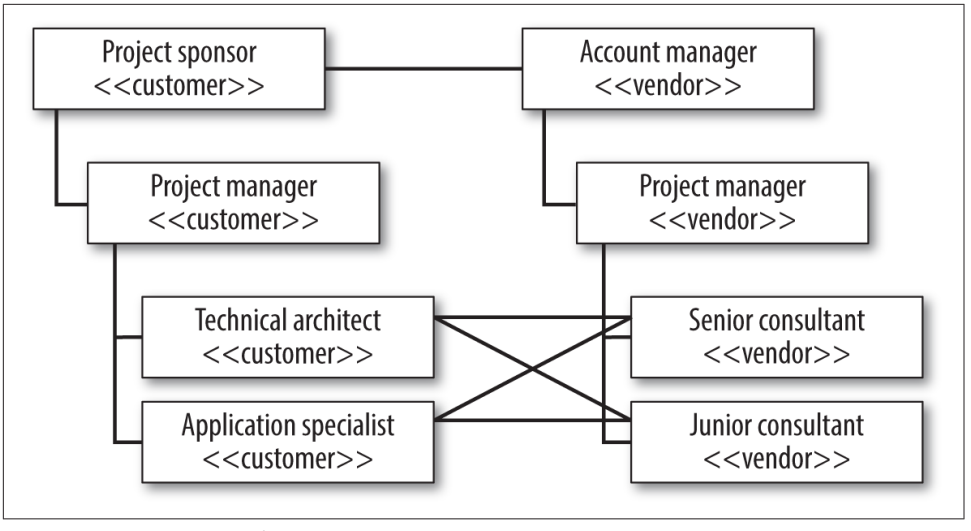


Figure B-1. *Example performance testing team structure*

Given this information, proceed as follows:

1. Develop a high-level plan that includes resources, timelines, and milestones based on these requirements.
2. Develop a detailed performance test plan that includes all dependencies and associated timelines, detailed scenarios and test cases, workloads, and environment information.
3. Include contingency for additional testing cycles and defect resolution if problems are found with the application during performance test execution.
4. Include a risk assessment of not meeting schedule or performance targets. With these actions under way, you can continue with each of the following steps. Not everything mentioned may be relevant to your particular testing requirements, but the order of events is important. (See Appendix E for an example MS Project–based performance testing project plan.)

Step 2: Test Environment Build

Strive to make your test environment a close approximation of the live environment. At a minimum it should reflect the server tier deployment of the live environment, and your target database should be populated realistically in terms of content *and* sizing. (This activity frequently takes much longer than expected.)

- Allow enough time to source equipment and to configure and build the environment.

- Take into account all deployment models (including LAN and WAN environments).
- Take external links into account, since they are a prime location for performance bottlenecks. Either use real links or create a realistic simulation of external communication.
- Provide enough load injection capacity for the scale of testing envisaged. Think about the locations where you will need to inject load from. If nonlocal load injector machines cannot be managed remotely, then local personnel must be on hand at each remote location. (*Don't forget to consider the cloud for load injection.*)
- Ensure that the application is correctly deployed into the test environment.
- Provide sufficient software licenses for the application and supporting software (e.g., Citrix or SAP).
- Ensure correct deployment and configuration of performance testing tools.
- Ensure correct deployment and configuration of KPI monitoring tools.

Step 3: Scripting

Proceed as follows for each use case to be scripted:

1. Identify the use-case *session* data requirements. You may have some insight into this requirement as part of a proof of concept.
2. Confirm and apply use-case *input* data requirements (see Appendix A).
3. Determine the use-case checkpoints that you will monitor separately for response time.
4. Identify and apply any changes required for the scripted use case to replay correctly.
5. Ensure that the script replays correctly from both a single-user and a multiuser perspective *before* including it in a performance test. Make sure you can verify what happens on replay.

Step 4: Performance Test Build

Making use of the application *load model* created as part of NFR capture, for each performance test consider the following points:

- Is it a pipe-clean, volume, stress, soak, or configuration test? A typical scenario is to have pipe-clean tests for each use case, first in isolation as a single user and then up to the target maximum currency or throughput. Run isolation tests to identify and deal with any problems that occur, followed by a load test combining

all use cases up to target concurrency. (You should then run stress and soak tests for the final testing cycles, followed perhaps by non-performance–related tests.)

- Decide on how you will represent *think time* and *pacing* for each use case based on the type of performance test scenario.
- For each use case, decide on how many load injector deployments you will make and how many *virtual users* should be assigned to each injection point of presence.
- Decide on the injection profile for each load injector deployment: Big Bang, ramp-up, ramp-up/ramp-down with step, or delayed start. Your performance test choice will likely involve a combination of Big Bang deployments to represent static load and one or more of the ramp variations to test for scalability (see Figure B-2).
- Will the tests execute for a certain length of time or be halted by running out of data, reaching a certain number of use-case iterations, or user intervention?
- Do you need to spoof IP addresses to correctly exercise application load balancing requirements? (If so, then you will need a list of valid IP addresses.)
- Do you need to simulate different network conditions like baud rates or cellular network quality? If so, then confirm the different scenarios required. Any response-time prediction or capacity modeling carried out prior to performance testing should have already given you valuable insight into how the application reacts to bandwidth restrictions.
- What runtime monitoring needs to be configured using the client, server, and network KPIs that have already been set up? The actual monitoring software should have been deployed as part of the test environment build phase, so you should already have a clear idea of exactly what you are going to monitor in the hosting infrastructure.
- If this is a web-based performance test, what level of browser caching simulation do you need to provide? New user, active user, returning user? This will very much depend on the capabilities of your performance testing solution. See Chapter 9 for a discussion of caching simulation.
- Consider any effects that the application technology will have on your performance test design. For example, SAP performance tests that make use of the SAP-GUI client will have a higher resource requirement than, say, a simple terminal emulator and will require more load injector machines to generate a given number of virtual users. Chapter 9 discusses additional considerations for SAP and other application technologies.

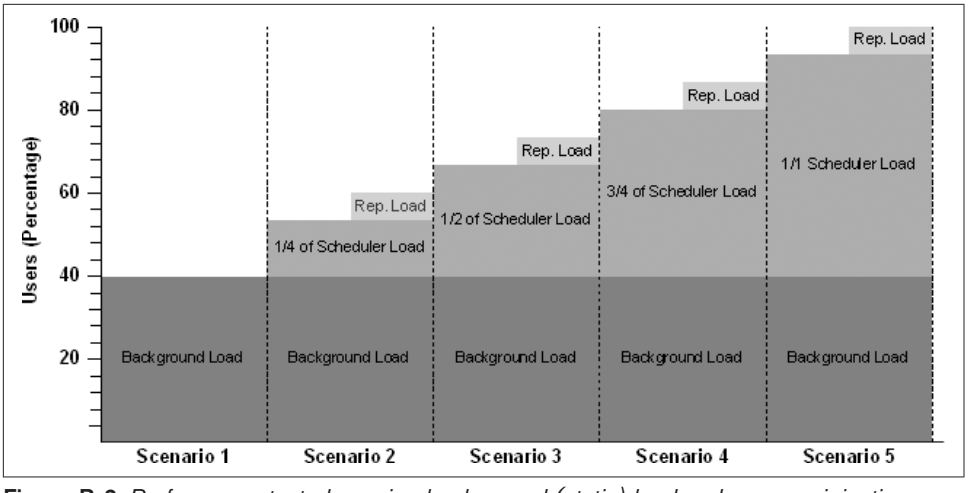


Figure B-2. *Performance test plan using background (static) load and ramp-up injection profiles*

Step 5: Performance Test Execution

Run and monitor your tests. Make sure that you carry out a *dress rehearsal* of each performance test as a final check that there are no problems accessing the application or with the test configuration. This phase should be the most straightforward part of any performance testing project. You've done the hard work: preparing the test environment, creating the scripts, addressing the data requirements, and building the performance tests. In an ideal world, performance test execution should be solely to validate your application performance targets. It should not become a bug-fixing exercise. The only unknown is how many test cycles will be required before you achieve your performance testing goals. I wish I could answer this question for you, but like many things in life this is in the lap of the gods. However, if you've followed the suggested performance testing checklists religiously to this point, you're in pretty good shape to be granted a miracle!

Here is a summary of tips related to performance text execution:

- Execute additional *dress-rehearsal* tests to verify you have sufficient load injection capacity for the target concurrency. Unless your concurrency targets are very modest, you should always check the upper limits of your load injectors.
- Execute pipe-clean tests to establish *ideal* response-time performance. (This is typically a single user per script for a set period of time or a certain number of iterations.)

- Execute volume tests, ideally resetting target database content between executions. This test normally includes all scripts prorated among the target number of virtual users.
- Execute isolation tests to explore any problems revealed by load testing, and then supply results to the developers or application vendor.
- Execute stress tests to generate data concerning future growth of transaction volume and application users.
- Execute soak tests (if time allows) to reveal any memory leaks and problems related to high-volume transaction executions.
- Execute any tests that are not performance related (e.g., different load-balancing configurations).

Step 6 (Post-Test Phase): Analyze Results, Report, Retest If Required

The following are pointers on what to do during the post-test phase:

- Make sure that you capture and back up *all* data created as part of the performance testing project.
- Compare test results to performance targets set as part of project requirements; this will determine the project's success or failure.
- Document the results of the project. Your report should include sections that address each of the performance targets.
- Use the final results as baseline data for end-user experience (EUE) monitoring.

Analysis Checklist

The focus here is on analysis rather than execution. See Chapter 5 for more details.

Pre-Test Analysis Tasks

- Make sure that you have configured the appropriate application, server, and network KPIs. Make sure there are no obstacles to installing and configuring agent software on the servers (unless you use remote monitoring).
- Set any available automatic thresholds for performance targets as part of your performance test configuration.

- If your performance testing tool provides the capability, configure autocorrelation between response time, concurrent virtual users, and server or network KPI metrics.
- If you are using third-party tools to provide some or all of your KPI monitoring, make sure that they are correctly configured *before* you run any tests—ideally, include them in your dress rehearsal.
- Be sure you can integrate third-party data with the output of your performance testing tool. Unless your tools do this automatically, this can be time-consuming.

Tasks During Test Execution

Here is a list of tasks to complete during test execution:

- Periodically examine the performance of your load injectors to ensure that they are not becoming stressed.
- Document every test that you execute. At a minimum, record the following:
  - The name of the performance test execution file and the date and time of execution
  - A brief description of what the test comprised
  - The name of the results file (if any) associated with the current test execution
  - Any input data files associated with the performance test and which use cases they relate to
  - A brief description of any problems that occurred during the test

If your performance testing tool allows you to annotate the performance test configuration with comments, use this facility to include whatever information will help you easily identify the test run. During execution, you should do the following:

- Watch for the sudden appearance of errors. This frequently indicates that some limit has been reached within the test environment; it can also mean that you've run out of data or that the operating system's default settings are interfering.
- Watch for a sudden drop in throughput. This is a classic sign of trouble, particularly with web applications where the virtual users wait for a response from the web server. If your application is using links to external systems, check to ensure that none of these links is the cause of the problem.
- Watch for an ongoing reduction in available server memory. Available memory should decrease as more and more virtual users become active, but if the decrease continues after all users are active, then you may have a memory leak.

Post-Test Tasks

Post-test tasks include the following:

- Collect all relevant data for each test execution. If you're relying on third-party tools to provide monitoring data, make sure that you preserve the files you need.
- Back up, onto a separate archive, all testing resources (scripts, input data files, and test results).
- Your report should map results to the performance targets that were set as part of the pre-test requirements capture phase.

APPENDIX C

Performance and Testing Tool Vendors

THIS APPENDIX PROVIDES A LIST OF THE LEADING AUTOMATED TOOL VENDORS for performance testing and related disciplines. I have also included entries for popular open source tools where appropriate. The tool vendor market remains dynamic, with a number of new entries since 2009. Many performance testing tool vendors now provide support for mobile testing either on-premise or as a SaaS offering, and the majority provide support for cloud-based load injection. The list is in no particular order and is not intended to be exhaustive. For each section, I have listed the tool vendor followed by the product name and (where appropriate) a link for more information.

Application Performance Management

Appdynamics
Appdynamics

Computer Associates: Introscope Compuware
dynaTrace

Correlsense
SharePath

New Relic
New Relic

End-User Experience and Website Monitoring

SOASTA
mPulse

Neustar
Website Monitoring Service

Google
WebPagetest

Functional Testing

Hewlett Packard
Quick Test Pro (QTP)

Selenium
Selenium

TestPlant
eggPlant Performance

Performance Testing

Borland
Silk Performer

Hewlett Packard
LoadRunner

IBM
Rational Performance Tester

Microsoft
Visual Studio Team System

Neotys
Neoload

Quotium
Qtest

TestPlant
eggPlant Performance

Open Source

Gatling
Gatling

(The) Grinder
Grinder

JMeter
JMeter

SaaS Performance Testing

Blazemeter
Blazemeter

Compuware
Gomez

Intechnica
TrafficSpike

KeyNote
KeyNote

NCC Group
SiteConfidence

SOASTA
Cloudtest

Requirements Management

Borland
Calibre-RM

IBM
Requisite-PRO

Open Source

TRUREQ
Trureq

APPENDIX D

Sample Monitoring Templates: Infrastructure Key Performance Indicator Metrics

THE FOLLOWING EXAMPLES DEMONSTRATE GROUPS OF COMMON SERVER KPI metrics that I use in performance testing projects. I have provided examples of generic and application-specific templates to demonstrate the top-down approach I use when troubleshooting performance issues revealed by performance testing.

Generic KPI Templates

Monitoring the use of this first set of metric templates provides a good indication of when a server is under stress. The metrics configured are focused on fundamental capacity indicators such as CPU loading and memory consumption. This is the basic level of server monitoring I use when performance testing.

Windows OS : Generic KPI Template

You will probably recognize these metrics, since they are taken from the Windows Performance Monitor (Perfmon) application. This is a nice mix of counters that monitor disk, memory, and CPU performance data, together with some high-level network information that measures the number of errors encountered and data throughput in

bytes per second. For many of these counters you will, of course, need to select the appropriate instances and sampling period based on your system under test (SUT) requirements. Make sure that the sampling period you select is not too frequent, as this will place additional load on the servers being monitored. The default of 15 seconds is usually sufficient to create enough data points without causing excessive load. The number of counters you are monitoring can also have an impact on server performance, so make sure that any templates you create have only the counters you really need. I recommend that, in addition to using this template, you monitor the top 10 processes in terms of CPU utilization and memory consumption. Identifying the CPU and memory hogs is often your best pointer to the next level of KPI monitoring, where you need to drill down into specific software applications and components that are part of the application deployment.

KPI metric	Notes
Total processor utilization %	
Processor queue length	
Context switches per second	
Memory available byes	
Memory page faults per second	*Use to assess soft page faults
Memory cache faults per second	*Use to assess soft page faults
Memory page readers per second	*Use to assess hard page faults
Free disk space %	
Page File Usage %	
Average disk queue length	
% Disk time	

Linux/Unix: Generic KPI Template

This next example is taken from the non-Windows OS world and demonstrates the same basic metrics about server performance. You generally have to use a number of different tools to get similar information to that provided by Windows Performance Monitor.

KPI metric	Source utility	Indicative parameter(s)
% Processor time	vmstat	cs,us,sys,id,wa
Processes on runq	vmstat	r
Blocked queue	vmstat	b

KPI metric	Source utility	Indicative parameter(s)
Memory available bytes	svmon	free
Memory page faults per second	sar	faults per second
Memory pages out per second	vmstat	po
Memory pages in per sec	vmstat	pi
Paging space	svmon	pg space
Device interrupts per sec	vmstat	in
% Disk time	iostat	%tm_act

Application-Specific KPI Templates

Drilling down from the generic KPI templates, the next level of analysis commonly focuses on metrics that are application specific. Use these templates to monitor software applications and discrete components that are part of your application deployment. These might include Microsoft's SQL Server database or one of the many Java-based application servers, such as JBOSS. Each application type will have its own recommended set of counters, so please refer to the corresponding vendor documentation to ensure that your KPI template contains the appropriate entries.

Windows OS: MS SQL Server KPI Template

The following example is also taken from Windows Performance Monitor and demonstrates suggested counters for MS SQL Server.

KPI metric	Notes
Access methods: Forward records/sec	
Access methods: Full table scans	*Missing indexes
Access methods: Index searches/sec	
Access methods: Page splits/sec	
Access methods: Table lock escalations/sec	
Buffer manager: Buffer cache hit ratio	
Buffer manager: Checkpoint pages/sec	
Buffer manager: Free list stalls/sec	
Buffer manager: Page life expectancy	
Buffer manager: Page lookups/sec	
Buffer manager: Page reads/sec	

KPI metric	Notes
Buffer manager: Page writes/sec	
General statistics: Logins/sec	
General statistics: Logouts/sec	
General Statistics: User connections	
Latches: Latch waits/sec	
Latches: Total latch wait time (ms)	
Locks: Lock wait time (ms)	*Lock contention
Locks: Lock waits/sec	*Lock contention
Locks: Number of deadlocks/sec	*Lock contention
Memory manager: Target server memory (KB)	*MS SQL memory
Memory manager: Total server memory (KB)	*MS SQL memory
SQL statistics: Batch requests/sec	
SQL statistics: Compilations/sec	
SQL statistics: Recompilations/sec	

APPENDIX E

Sample Project Plan

THE EXAMPLE PROJECT PLAN IN FIGURE E-1 IS BASED ON A TYPICAL PERFOR-mance project template in MS Project. I have updated the original content to reflect current best practice.

NOTE

The MS Project file is available from the book's companion website.

	Task Name	Duration	Start	Finish	Predecessors	Resource Names
1	**Performance Test Planning Example (Based on 5 Use Cases)**	**10.6 days?**	**Mon 03/03/14**	**Mon 17/03/14**		**Project manager ¦ Account manager**
2	**Scoping / Statement of Work (SOW)**	**2.25 days?**	**Mon 03/03/14**	**Wed 05/03/14**		**Senior Performance Consultant ¦ Performance Consultant**
3	Execute scoping exercise with customer	1 day?	Mon 03/03/14	Mon 03/03/14		
4	Create full SOW / project plan	1 day?	Tue 04/03/14	Tue 04/03/14	3	
5	Sign-off SOW	0.25 days?	Wed 05/03/14	Wed 05/03/14	4	
6	**Design**	**4 days?**	**Wed 05/03/14**	**Tue 11/03/14**	**2**	**Project Manager ¦ Performance Consultant**
7	Project kick-off meeting	0.25 days?	Wed 05/03/14	Wed 05/03/14		
8	Build and validate scripts	3 days?	Wed 05/03/14	Mon 10/03/14		
9	Build and validate performance test scenarios	1 day?	Mon 10/03/14	Tue 11/03/14	8	
10	**Iterative execution and analysis**	**2.6 days?**	**Tue 11/03/14**	**Thu 13/03/14**	**6**	**Performance Consultant**
11	**Pipeclean**	**0.35 days?**	**Tue 11/03/14**	**Tue 11/03/14**		
12	Execution - Real time feedback	0.25 days?	Tue 11/03/14	Tue 11/03/14		
13	Establish baseline / adjust load model	0.1 days?	Tue 11/03/14	Tue 11/03/14	12	
14	**Volume**	**0.5 days?**	**Tue 11/03/14**	**Wed 12/03/14**	**11**	
15	Execution - Real time feedback	0.25 days?	Tue 11/03/14	Tue 11/03/14		
16	Interim report	0.25 days?	Tue 11/03/14	Wed 12/03/14	15	
17	**Stress**	**0.5 days?**	**Wed 12/03/14**	**Wed 12/03/14**	**14**	
18	Execution - Real time feedback	0.25 days?	Wed 12/03/14	Wed 12/03/14		
19	Interim report	0.25 days?	Wed 12/03/14	Wed 12/03/14	18	
20	**Soak**	**0.75 days?**	**Wed 12/03/14**	**Thu 13/03/14**	**17**	
21	Execution - Real time feedback	0.5 days?	Wed 12/03/14	Thu 13/03/14		
22	Interim report	0.25 days?	Thu 13/03/14	Thu 13/03/14	21	
23	**Configuration**	**0.5 days?**	**Thu 13/03/14**	**Thu 13/03/14**	**20**	
24	Execution - Real time feedback	0.25 days?	Thu 13/03/14	Thu 13/03/14		
25	Interim report	0.25 days?	Thu 13/03/14	Thu 13/03/14	24	
26	**Closure**	**1.75 days?**	**Thu 13/03/14**	**Mon 17/03/14**	**10**	**Senior Performance Consultant ¦ Performance Consultant**
27	Data collection / uninstall	0.5 days?	Thu 13/03/14	Fri 14/03/14		Performance Consultant
28	Produce final performance closure report	1 day?	Fri 14/03/14	Mon 17/03/14	27	Senior Performance Consultant ¦ Performance Consultant
29	Present findings to client	0.25 days?	Mon 17/03/14	Mon 17/03/14	28	Senior Performance Consultant ¦ Project Manager ¦ Account Manager
30						

Figure E-1. *Example MS Project performance test plan*

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Colophon

The cover fonts are URW Typewriter and Guardian Sans. The text font is Meridien; the heading font is Akzidenz Grotesk; and the code font is Dalton Maag's Ubuntu Mono.

About the Author

Originally hailing from Auckland, New Zealand, **Ian Molyneaux** ended up in IT purely by chance after applying for an interesting looking job advertised as "junior computer operator" in the mid '70s. The rest is history: 36 years later Ian has held many roles in IT but confesses to being a techie at heart with a special interest in application performance. Ian's current role is Head of Performance for Intechnica, a UK-based digital performance consultancy.

On a personal level Ian enjoys crossfit training, music, and reading science fiction with a particular fondness for the works of Larry Niven and Jerry Pournelle. Ian presently resides in Buckinghamshire, UK, with wife Sarah and three cats, trying to get used to the idea of turning 56!