O'REILLY®

Web Performance Warrior

The Business of Speed

Andy Still

Web Performance Warrior

Delivering Performance to Your Development Process

Andy Still

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

Web Performance Warrior
by Andy Still

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.

Editor: Andy Oram
Production Editor: Kristen Brown
Copyeditor: Amanda Kersey
Interior Designer: David Futato
Cover Designer: Ellie Volckhausen
Illustrator: Rebecca Demarest

February 2015: First Edition

Revision History for the First Edition
2015-01-20: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491919613* for release details.

978-1-491-91961-3

[LSI]

For Morgan & Savannah, future performance warriors

Table of Contents

Foreword

In 2004 I was involved in a performance disaster on a site that I was responsible for. The system had happily handled the traffic peaks previously seen but on this day was the victim of an unexpectedly large influx of traffic related to a major event and failed in dramatic fashion.

I then spent the next year re-architecting the system to be able to cope with the same event in 2005. All the effort paid off, and it was a resounding success.

What I took from that experience was how difficult it was to find sources of information or help related to performance improvement.

In 2008, I cofounded Intechnica as a performance consultancy that aimed to help people in similar situations get the guidance they needed to solve performance issues or, ideally, to prevent issues and work with people to implement these processes.

Since then we have worked with a large number of companies of different sizes and industries, as well as built our own products in house, but the challenges we see people facing remain fairly consistent.

This book aims to share the insights we have gained from such real-world experience.

The content owes a lot to the work I have done with my cofounder Jeremy Gidlow; ops director, David Horton; and our head of performance, Ian Molyneaux. A lot of credit is due to them in contributing to the thinking in this area.

Credit is also due to our external monitoring consultant, Larry Haig, for his contribution to Chapter 6.

Additional credit is due to all our performance experts and engineers at Intechnica, both past and present, all of whom have moved the web performance industry forward by responding to and handling the challenges they face every day in improving client and internal systems.

Chapter 3 was augmented by discussion with all WOPR22 attendees: Fredrik Fristedt, Andy Hohenner, Paul Holland, Martin Hynie, Emil Johansson, Maria Kedemo, John Meza, Eric Proegler, Bob Sklar, Paul Stapleton, Neil Taitt, and Mais Tawfik Ashkar.

Preface

For modern-day applications, performance is a major concern. Numerous studies show that poorly performing applications or websites lose customers and that poor performance can have a detrimental effect on a company's public image. Yet all too often, corporate executives don't see performance as a priority—or just don't know what it takes to achieve acceptable performance.

Usually, someone dealing with the application in real working conditions realizes the importance of performance and wants to do something about it.

If you are this person, it is easy to feel like a voice calling in the wilderness, fighting a battle that no one else cares about. It is difficult to know where to start to solve the performance problem.

This book will try to set you on the right track.

This process I describe in this book will allow you to declare war on poor performance to become a *performance warrior*.

The performance warrior is not a particular team member; it could be anyone within a development team. It could be a developer, a development manager, a tester, a product owner, or even a CTO.

A performance warrior will face battles that are technical, political and economic.

This book will not train you to be a performance engineer: it will not tell you which tool to use to figure out why your website is running slow or tell you which open source tools or proprietary tools are best for a particular task.

However, it will give you a framework that will help guide you toward a development process that will optimize the performance of your website.

NOTE

It's Not Just About the Web

Web Performance Warrior is written with web development in mind; however, most of the advice will be equally valid to other types of development.

The Six Phases

I have split the journey into six phases. Each phase includes an action plan stating practical steps you can take to solve the problems addressed by that phase:

1. *Acceptance*: "Performance doesn't come for free."
2. *Promotion*: "Performance is a first-class citizen."
3. *Strategy*: "What do you mean by 'good performance'?"
4. *Engage*: "Test…test early…test often…"
5. *Intelligence*: "Collect data and reduce guesswork."
6. *Persistence*: "'Go live' is the start of performance optimization."

Phase 1: Acceptance *"Performance Doesn't Come For Free"*

The journey of a thousand miles starts with a single step. For a performance warrior, that first step is the realization that good performance won't just happen: it will require time, effort, and expertise.

Often this realization is reached in the heat of battle, as your systems are suffering under the weight of performance problems. Users are complaining, the business is losing money, servers are falling over, there are a lot of angry people about demanding that something be done about it. Panicked actions will take place: emergency changes, late nights, scattergun fixes, new kit. Eventually a resolution will be found, and things will settle down again.

When things calm down, most people will lose interest and go back to their day jobs. Those that retain interest are performance warriors.

In an ideal world, you could start your journey to being a performance warrior before this stage by eliminating performance problems before they start to impact the business.

Convincing Others

The next step after realizing that performance won't come for free is convincing the rest of your business.

Perhaps you are lucky and have an understanding company that will listen to your concerns and allocate time, money, and resources to

you to resolve these issues and a development team that is on board with the process and wants to work with you to make it happen. In this case, skip ahead to Chapter 2.

Still reading? Then you are working a typical organization that has only a limited interest in the performance of its web systems. It becomes the job of the performance warrior to convince colleagues it is something they need to be concerned about.

For many people across the company (both technical and non-technical, senior and junior) in all types of business (old and new, traditional and techy), this will be a difficult step to take. It involves an acceptance that performance won't just come along with good development but needs to be planned, tested, and budgeted for. This means that appropriate time, money, and effort will have to be provided to ensure that systems are performant.

You must be prepared to meet this resistance and understand why people feel this way.

Developer Objections

It may sound obvious that performance will not just happen on its own, but many developers need to be educated to understand this.

A lot of teams have never considered performance because they have never found it to be an issue. Anything written by a team of reasonably competent developers can probably be assumed to be reasonably performant. By this I mean that for a single user, on a test platform with a test-sized data set, it will perform to a reasonable level. We can hope that developers should have enough pride in what they are producing to ensure that the minimum standard has been met. (OK, I accept that this is not always the case.)

For many systems, the rigors of production are not massively greater than the test environment, so performance doesn't become a consideration. Or if it turns out to be a problem, it is addressed on the basis of specific issues that are treated as functional bugs.

Performance can sneak up on teams that have not had to deal with it before.

Developers often feel sensitive to the implications of putting more of a performance focus into the development process. It is important to appreciate why this may be the case:

Professional pride
: It is an implied criticism of the quality of work they are producing. While we mentioned the naiveté of business users in expecting performance to just come from nowhere, there is often a sense among developers that good work will automatically perform well, and they regard lapses in performance as a failure on their part.

Fear of change
: There is a natural resistance to change. The additional work that may be needed to bring the performance of systems to the next level may well take developers out of their comfort zone. This will then lead to a natural fear that they will not be able to manage the new technologies, working practices, etc.

Fear for their jobs
: The understandable fear with many developers, when admitting that the work they have done so far is not performant, is that it will be seen by the business as an admission that they are not up to the job and therefore should be replaced. Developers are afraid, in other words, that the problem will be seen not as a result of needing to put more time, skills, and money into performance, just as having the wrong people.

Handling Developer Objections

Developer concerns are best dealt with by adopting a three-pronged approach:

Reassurance
: Reassure developers that the time, training, and tooling needed to achieve these objectives will be provided.

Professional pride
: Make it a matter of professional pride that the system they are working on has got to be faster, better-scaling, lower memory use, etc., than its competitors. Make this a shared objective rather than a chore.

Incentivize the outcome
: Make hitting the targets rewardable in some way, for example, through an interdepartmental competition, company recognition, or material reward.

Business Objections

Objections you face from within the business are usually due to the increased budget or timescales that will be required to ensure better performance.

Arguments will usually revolve around the following core themes:

How hard can it be?
: There is no frame of reference for the business to be able to understand the unique challenges of performance in complex systems. It may be easy for a nontechnical person to understand the complexities of the system's functional requirements, but the complexities caused by doing these same activities at scale are not as apparent.

  Beyond that, business leaders often share the belief that if a developer has done his/her job well, then the system will be performant.

  There needs to be an acceptance that this is not the case and that this is not the fault of the developer. Getting a truly performant system requires dedicated time, effort, and money.

It worked before. Why doesn't it work now?
: This question is regularly seen in evolving systems. As levels of usage and data quantities grow, usually combined with additional functionality, performance will start to suffer.

  Performance challenges will become exponentially more complex as the footprint of a system grows (levels of usage, data quantities, additional functionality, interactions between systems, etc.). This is especially true of a system that is carrying technical debt (i.e., most systems).

  Often this can be illustrated to the business by producing visual representations of the growth of the system. However, it will then often lead to the next argument.

Why didn't you build it properly in the first place?
: Performance problems are an understandable consequence of system growth, yet the fault is often placed at the door of developers for not building a system that can scale.

  There are several counterarguments to that:

- The success criteria for the system and levels of usage, data, and scaling that would eventually be required were not defined or known at the start, so the developers couldn't have known what they were working toward.
- Time or money wasn't available to invest in building the system that would have been required to scale.
- The current complexity of the system was not anticipated when the system was first designed.
- It would actually have been irresponsible to build the system for this level of usage at the start of the process, when the evolution of the system and its usage were unknown. Attempts to create a scalable system may actually have resulted in more technical debt. Millions of hours of developer time is wasted every year in supporting systems that were over-engineered because of overly ambitious usage expectations that were set at the start of a project.

Although all these arguments may be valid, often the argument as to why this has happened is much simpler. Developers are only human, and high-volume systems create challenges that are complex. Therefore, despite their best efforts, developers make decisions that in hindsight turn out to be wrong or that don't anticipate how components integrate.

Handling Business Objections

There are several approaches to answering business objections:

Illustrate the causes of the problem
: Provide some data around the increased size, usage, data quantities, and complexity of the system that illustrate performance problems as a natural result of this growth.

Put the position in context of other costs
: Consider the amount of resources/budget that is applied to other types of testing, such as functional and security testing, and urge that performance to be considered at the same level. Functional correctness also doesn't come for free. Days of effort go into defining the functional behavior of systems in advance and validating them afterwards. Any development team that suggested developing a system with no upfront definition of what it would do and no testing (either formal or informal) of functional correctness would rightly be con-

demned as irresponsible. Emphasize that performance should be treated in the same way.

Put the problem in financial terms
: Illustrate how much performance issues are directly costing the business. This may be in terms of downtime (i.e., lost sales or productivity) or in additional costs (e.g., extra hardware).

Show the business benefit
: Explain how you could get a market advantage from being the fastest system or the system that is always up.

Illustrate why the process is needed
: Show some of the complexities of performance issues and why they are difficult to address as part of a standard development process; that is, illustrate why poor performance does not necessarily equal poor-quality development. For example, arguments such as:

- **Performance is not like functional issues**. Functional issues are black and white: something either does what it should do or it doesn't. If someone else has complained of a functional error, you can replicate it by manipulating the inputs and state of the test system; and once it is replicated, you can fix it. Performance issues are infinitely more complex, and the pass/fail criteria are much more gray.
- **Performance is harder to see**. Something can appear to work correctly and perform in an acceptable manner in some situations while failing in others.
- **Performance is dependent on factors beyond the developers control**. Factors such as levels of concurrency, quantity of data, and query specifics all have an influence.

Action Plan

Separate Performance Validation, Improvement, and Optimization from Standard Development

A simple step: if no one realizes that performance requires work, start pointing it out. When estimating or doing sprint planning, create distinct tasks for performance optimization and validation. Highlight the importance so that, if performance is not explicitly put into the development plan by the organization, it has to make a conscious choice not to do so.

Complete a Performance Maturity Assessment

This is an exercise in assessing how mature your performance process is. Evaluate your company's processes, and determine how well suited it is for ensuring that the application being built is suitably performant. Also evaluate it against industry best practice (or the best practices that you feel should be introduced; remember to be realistic).

Produce this as a document with a score to indicate the current state of performance within the company.

Define a Strategy and Roadmap to Good Performance

Create an explicit plan for how to get from where you are to where you need to be. This should be in achievable, incremental steps and have some ideas of the time, effort, and costs that will be involved. It is important that developers, testers, managers, and others have input into this process so that they buy in to the process.

Once the roadmap is created, regularly update and track progress against it. Every step along the roadmap should increase your performance maturity score.

Performance won't come for free. This is your chance to illustrate to your business what is needed.

Phase 2: Promotion
"Performance is a First-Class Citizen"

The next step on the journey to becoming a performance warrior is to get your management and colleagues to treat performance with appropriate seriousness. Performance can be controlled only if it truly is treated as a first-class citizen within your development process.

Is Performance Really a First-Class Citizen?

Performance can kill a web application. That is a simple fact. The impact of a performance issue often grows exponentially as usage increases, unlike that of a functional issue, which tends to be linear.

Performance issues will take your system out completely, leading to complete loss of income, negative PR, and long-term loss of business and reputation. Look back at news reports related to website failures in recent years: very few are related to functional issues; almost all relate to performance.

Performance issues can lead to a requirement for complete re-architecting. This can mean developing additional components, moving to a new platform, buying third-party tools and services, or even a complete rewrite of the system.

Performance is therefore important and should be treated as such.

This chapter will help you to elevate performance to a first-class citizen, focusing on the challenges faced with relation to people, process, and tooling.

People

As the previous chapter explained, many companies hold the view that performance issues should just be solved by developers and that performance issues are actually simply caused by poor-quality development. Managers and developers alike feel like they should be able to achieve good performance just through more time or more powerful hardware.

In reality, of course, that is true up to a point. If you are developing a website of average complexity with moderate usage and moderate data levels, you should be able to develop code that performs to an acceptable level. As soon as these factors start to ramp up , however, performance will suffer and will require special expertise to solve. This does not reflect on the competency of the developer; it means that specialized skill is required.

The analogy I would make to this would be to look at the security of a website. For a standard brochureware or low-risk site, a competent developer should be able to deliver a site with sufficient security in place. However, when moving up to a banking site, you would no longer expect the developer to implement security. Security specialists would be involved and would be looking beyond the code to the system as a whole. Security is so important to the system and so complex that only a specialist can fully understand what's required at that level. Managers accept this because security is regarded as a first-class citizen in the development world.

Performance is exactly the same: performance issues often require such a breadth of knowledge (APM tooling, load generation tools, network setup, system interaction, concurrency effects, threading, database optimization, garbage collection, etc.) that specialists are required to solve them. To address performance, either appropriately skilled individuals must be recruited or existing people skilled up. This is the role of the performance engineer.

Performance engineers are not better than developers (indeed they are often also developers); they just have different skills.

Process

Performance is often not considered in a typical development process at all, or is done as a validation step at the end. This is not treating performance as a first-class citizen.

In this sense, performance is again like security, as well as other nonfunctional requirements (NFRs). Let's look at how NFRs are integrated into the development process.

For security, an upfront risk assessment takes place to identify necessary security standards, and testing is done before major releases. Builds will not be released if the business is not satisfied that security standards have been met.

For user experience (UX) design, the company will typically allocated a design period up front, dedicate time to it within the development process, and allow additional testing and validation time afterward. Builds will not be released if the business is not happy with the UX.

In contrast, performance is often not considered at all. If it is, the developers do it in vague, subjective terms ("must be fast to load"), with no consideration of key issues such as platform size, data quantities and usage levels. It is then tested too late, if at all.

To be an effective performance warrior, you must start considering performance throughout the development lifecycle. This includes things such as doing performance risk assessments at the start of a project, setting performance targets, building performance testing and performance code reviews into the development process, and failing projects if performance acceptance targets are not met. Many of these are addressed in more detail in later chapters.

Tooling

To effectively handle performance challenges, you need the right tools for the job.

A wide range of tools that can be used, from tools that come built into the systems being used (for instance, Perfmon on Windows), to open source toolsets (for instance, JMeter), free web-based tools (such as WebPagetest), and tools that you can pay a little or a lot for.

Determining the right toolset is a difficult task and will vary greatly depending on:

- The kind of performance challenge you are facing (poor performance under load, poor performance not under load, poor database performance, networking congestion, etc.)
- The platform you are working on
- The type of system you develop (website, desktop, web service, mobile app, etc.)
- The budget you have to work with
- Skillsets you have in house
- Other tools already used in house or existing licences that can be leveraged

Choosing the right tools for your situation is very important. Poor tool choices can lead to wasted time and effort when trying to get to the bottom of a problem by misdiagnosing the root cause of an issue.

It is also essential that sufficient hardware and training is provided to get the full value out of the selected tools. Performance tooling is often complex, and users need to be given time and support to get the full value from it.

Action Plan

Make Performance Part of the Conversation

All too often, performance flies under the radar because it is never discussed. As a performance warrior, your first step is to change that, and a few simple steps can move the discussion forward:

- Start discussing performance at planning sessions, standups, retrospectives, and other get-togethers.
- Start asking the business users what they expect from performance.
- Start asking the development team how they plan on addressing potential performance bottlenecks.
- Start asking the testers how they plan on validating performance.

Often the answers to these questions will be unsatisfactory, but at least the conversation is started.

Set Performance Targets

It is essential that everyone within the team know what levels of performance the system is aiming for and what metrics they should be considering. This subject is addressed in more detail in the next chapter.

Treat Performance Issues with the Same Importance and Severity as Functional Issues

Performance issues should fail builds. Whether informal or formal performance targets have been set, there must be the organizational policies to declare a build not fit for release on the grounds of performance.

This then will require testing for performance, not just for functionality.

Assign Someone with Responsibility for Performance Within the Project

When performance is not being considered, a good way to move things forward is to assign someone within a team who is responsible for performance on that project/product. This doesn't necessarily mean that this person will be doing all performance target setting, testing, optimization, etc. She will just be responsible for making sure that it is done and that performance is satisfactory.

How to Integrate Performance Engineers into Development Projects

There are several structures you can choose from to implement a performance ethos into a development team:

1. Assign an existing team member.

   For smaller teams, this is often the only option: an existing team member is assigned this job alongside his usual role.

   Pros

   - That person has a good understanding of the project in a wider context.
   - Low cost.

Cons

- It will inevitably create conflicts with the time needed for that person's existing role.
- The person selected will not be a specialist in performance.

2. Place a dedicated person within the team.

   Embed someone within the team who is a dedicated performance engineer.

   Pros

   - Dedicated resource focused only on performance.
   - In-depth knowledge of the project, and thus well aligned with other team members.

   Cons

   - Can result in inconsistent performance practice across different projects.
   - An expensive proposition if you have a large number of teams.
   - May be underutilized during some parts of the development process.

3. Create a separate performance team.

   This team will be spread across all projects and provide expertise as needed.

   Pros

   - Pool of performance experts providing best practice to all projects.
   - Can share expertise across entire business.

   Cons

   - Not fully part of the core delivery team, which can lead to an us/them conflict.
   - Can lead to contention among projects.
   - Performance team members may not have the detailed knowledge of the product/project being completed.

4. Use an external agency.

   There are external agencies that provide this expertise as a service, either as an offsite or onsite resource.

Pros

- Flexible because company can increase or reduce coverage as needed.
- Reduced recruitment overhead and time.
- High level of expertise.

Cons

- Can be expensive.
- Time needed to integrate into team and company.

Give People What They Need To Get Expertise

Performance engineering is hard: it requires a breadth of understanding of a wide range of aspects of application development that can contribute to performance issues (clients, browsers, network, third-party systems, protocols, hardware, OS, code, databases, etc.). There are also many specialist tools that can be used to identify the cause of performance issues. All these require a specialist's skills.

Performance testing presents a new set of challenges (what to test, how to test, how the load model should be constructed, where to test from, how to analyse the results, etc). These skills don't lie beyond most people within development teams, but they do need time to learn and practice the skills needed and the budget to buy the tools.

Create a Culture of Performance

This sounds grandiose but doesn't need to be. It simply means evolving your company to see the performance of its systems as a key differentiator. Good performance should be something that everyone within the company is proud of, and you should always be striving toward better performance. Often this culture will start within one team and then be driven out to the wider business.

Some simple rules to follow when thinking about how to introduce a performance culture include:

- Be realistic: focus on evolution, not revolution. Change is hard for most people.
- Take small steps: set some achievable targets and then celebrate hitting them, after which you can set harder targets.

- Put things into a relevant context: present stats that convey performance in terms that will matter to people. Page load time will be of little interest to the business, but the relationship between page load time and sales will be.
- Get buy-in from above: performance can begin as a grassroots movement within an organization, and often does; but in order to truly get the results the site needs, it eventually needs buy-in from a senior level.
- Start sending out regular reports about performance improvements and the impact they are having on the business.

Phase 3: Strategy
"What Do You Mean by 'Good Performance'?"

Having got buy-in to introducing performance as a central part of your development process, the next question you have to answer as a performance warrior is, "What do you mean by 'good performance'?"

The answer to this question will vary for every product and project, but it is crucial for all stakeholders to agree on a definition. It is easy to get drawn in to vague concepts like "The site must be fast," but these are of limited value beyond high-level discussions.

Fundamentally, all stakeholders need to share an understanding of a *performance landscape*. This is a communal understanding of the key performance characteristics of your system, what measures of performance are important, and what targets you are working toward.

It is important to define your success criteria and the framework within which those criteria must be met. This includes ensuring that you have defined the platform that the system will be running on, the expected data quantities, the expected usage levels, and so on. Defining this landscape is essential to allow developers to make reasonable assessments of the levels of optimization that are appropriate to perform on the system.

All the time, effort, and investment that has been put into the first two phases can be undermined if this phase is handled badly. This is where you identify the value of performance improvements to the

business and how that value will be assessed. This is what you will be measured against.

Three Levels of the Performance Landscape

There are three levels at which to define what you mean by good performance.

- Performance vision
- Performance targets
- Performance acceptance criteria

Performance Vision

The starting point for performance across any system is creating a *performance vision*. This is a short document that defines at a very high level how performance should be considered within your system. The document is aimed at a wide audience, from management to developers to ops, and should be written to be timeless and talk mainly in concepts, not specifics. Short-term objectives can be defined elsewhere, but they will all be framed in terms of the overall performance vision.

This document is the place to define which elements of performance are important to your business. Therefore, all the statements should be backed by a valid business focus. This document is not the place where you define the specific targets that you will achieve, only the nature of those targets.

For example, this document would not define that the homepage must load in less than two seconds, only that homepage loading speed is an area of key importance to the business and one that is used as a means of differentiation over competitors.

As a performance warrior, this document is your r*ules of engagement*. That is, it doesn't define how or where the battle against poor performance will be fought, but it does define the terms under which you should enter into battle.

It is important to get as much business involvement as possible in the production of this document and from people covering a wide swath of the business: management, sales, customer service, ops, development, etc.

The following sidebar shows an example of a performance vision for a rock- and pop-music ticketing site.

Sample Performance Vision

Headlines

- The ability to remain up under load is a key differentiator between the company and competitors.
- Failure to remain up can result in tickets being transferred to competitors and PR issues.
- The industry is largely judged by its ability to cope under load.
- Peaks (1,000 times normal load) are rare but not unknown.

Details

The primary aim of the system is to deliver throughput of sales at all times. It is acceptable for customers to be turned away from the site or have a reduced service as long as there is a flow of completed transactions.

There is an acceptance that there will be extremely high peaks of traffic and that there is no intention of being able to scale out to meet the load of those peaks, but the system must remain up and serving responses and completing transactions throughout those peaks. The more transactions that can be processed, the better, but the cost of being able to process additional transactions must be financially feasible.

It is essential to maintain data integrity during high load. Repeated bookings and duplicate sales of tickets are the worst-case scenario.

Most peaks can be predicted, and it is acceptable for manual processes to be in place to accommodate them.

During peak events, there are two types of traffic: visitors there for the specific event and visitors there shopping for other events who are caught up in traffic. Visitors there for the specific event will be more tolerant of poor performance than the normal visitors.

There is an expectation that the system will perform adequately under normal load, but this is not seen as the key area of focus or an area of differentiation between the business and competitors.

KPIs / Success Criteria

The following KPIs will be tracked to measure the success of the performance of this system:

- The level of traffic that can be handled by the system under high load. This is defined as the number of people to whom we can return an information page explaining that the site is busy.
- The level of transactions per minute that can be possessed successfully by the system while the site is under peak load.
- The impact of peak events on "normal traffic," i.e., users who are on the site to buy tickets for other events.

Performance Targets

Having defined your performance vision, the next step is to start defining some measurable key performance indicators (KPIs) for your system. These will be used as the basis for your *performance targets*. Unlike the performance vision, which is designed to be a static document, these will evolve over time. Performance targets are sometimes referred to as your *performance budget*.

If the performance vision is your rules of engagement, the performance targets are your strategic objectives. These are the standards you are trying to achieve. Your KPIs should include numeric values (or other measurable targets) against which you can assess your progress.

It is essential that performance targets be:

- Realistic and achievable
- Business focused
- Measurable
- In line with the performance vision

Your performance targets fulfill two important roles:

- They create a focal point around which the development team can all work together toward a shared objective.
- They create a measurable degree of success that can be used by the rest of the business to determine the value of the performance-improvement process.

Once the performance targets are defined, it is essential to assess progress toward them regularly and report the progress throughout the business.

Example performance targets could be:

- Homepage loads in under 5 seconds.
- Above-the-fold homepage content visible to users in less than 1 second.
- Site capable of handling 10,000 orders in 1 hour.
- Site capable of handling peak load on 8 rather than the current 12 servers.
- Homepage load time faster than named competitors.
- Average search processing time less than 2 seconds.
- No SQL query execution should take more than 500 milliseconds.

Performance Acceptance Criteria

Having defined your rules of engagement (performance vision) and your strategic objectives (KPIs), you now need to define tactical objectives. These are short-term, specific objectives that will move you toward one or more of your strategic objectives. They will take the form of performance acceptance criteria that are defined for each feature, user story, or specification.

Any piece of work that is accepted into development, whether in an agile world (e.g., a feature or user story) or traditional waterfall development (e.g., a specification), should have a pre-defined set of performance-related criteria that must be met for the development to be accepted. Anything not meeting these criteria should be deemed as not fit for purpose.

All performance acceptance criteria should be framed in the context of moving towards, or at least maintaining the current state, of one of the KPIs.

Two types of acceptance criteria are associated with tasks:

- Those where the task is specifically focused on performance improvements and should see an improvement of performance against a KPI.

- Those where the task is additional functionality and the main objective will be to ensure that performance remains static or has an acceptable level of degradation against a KPI.

In both cases, is important to keep the criteria realistic.

Tips for Setting Performance Targets

Solve Business Problems, Not Technical Challenges

An often-heard complaint from budding performance warriors who are trying to get buy-in from their business and are struggling is, "Why don't they realize that they want as fast a website as possible?"

The simple answer to this is, "Because they don't!" The business in question does not want a fast website. The business wants to make as much money as possible. Only if having a fast website is a vehicle for them doing that do they want a fast website.

This is an essential point to grasp: make sure you are solving business problems, not technical challenges.

As a techie, it's easy to get excited by the challenge of setting arbitrary targets and putting time and effort into continually bettering them, when more business benefit could be gained from solving other performance problems or investing the money in non-performance enhancements.

There is a lot of professional pride to be had in having a faster page load time than your nearest competitor (or the company you used to work for), but slow page load time may not be your company's pain point.

So take a step back and understand the performance problems and how they are impacting the business. Start your performance optimization with a business impact, put it into financial terms, and provide the means to justify the value of the performance optimization to the business.

Think Beyond Page Load Time

The headline figure for a lot of discussions around performance is page load time, particularly page load time when under typical, non-peak traffic levels. However, it is important that you actually focus

on addressing the performance issues that are affecting your business.

The issues you are seeing may be slow page load when the system is not under load. It's equally possible, however, that you are seeing slowdowns under adverse conditions, intermittent slowdowns under normal load, excessive resource usage on the server necessitating an excessively large platform, or many other potential problems.

All of these examples can be boiled down to a direct financial impact on the business.

As an example, one company I worked with determined that its intermittent slowdowns cost 350 sales on average, which would work out to £3.36 million per year. This gives you instant business buy-in to solve the problem and a direct goal for developers to work on. Furthermore, it provides an observable KPI to track achievement and know when you are done, after which you can move on to the next performance problem.

Another company I worked with had a system that performed perfectly adequately but was very memory hungry. Its business objective was to release some of the memory being used by the servers to be used on alternative projects (i.e., reduce the hosting cost for the application). Again, this was a direct business case, a problem developers can get their teeth into and a observable KPI.

Look at the actual problems your business is having and understand the impact, then set your KPIs based on this.

Beware Over-optimization

When setting your targets, always remember: be realistic. Many a performance warrior has fallen into the trap of setting their targets too high, being over-ambitious, and trying to build an ultra-performant website.

But shouldn't you always build the most ultra-performant system you can?

No. You should always build an *appropriately* performant system. Over-optimizing a system can be just as negative as under-optimizing. Building a ultra-performant, scalable web application takes many factors, such as:

Time
: Building highly performant systems just takes longer.

Complexity
: Highly optimized systems tend to have a lot more moving parts. Elements such as caching layers, NoSQL databases, sharded databases, cloned data stores, message queues, remote components, multiple languages, technologies and platforms may be introduced to ensure that your system can scale and remain performant. All these things require management, testing, development expertise, and hardware.

Expertise
: Building an ultra-performant website is hard. It takes clever people to devise intelligent solutions, often operating at the limits of the technologies being used. These kind of solutions can lead to areas of your system being unmaintainable by the rest of the team. Some of the worst maintenance situations I have seen have been caused by the introduction of some unnecessarily complicated piece of coding designed to solve a potential performance problem that never materialized.

Investment
: These systems require financial investment to build and support, in terms of hardware, software, and development/testing time and effort. The people required to build them are usually highly paid.

Compromises
: Solving performance issues is done often at the expense of good practice or functionality elsewhere. This may be as simple as compromising on the timeliness of data by introducing caching, but often maybe accepting architectural compromises or even compromises in good coding practice to achieve performance.

Action Plan

Create Your Performance Vision

The first step is to define the performance vision for your company/system.

This typically starts with one or a number of workshops with relevant parties gathering data about how performance impacts their worklife in a positive and negative manner. A good starting point for these workshops is with the question, "What do we mean by 'good performance'?". Remember always to try and focus the discussions around what the business impact of the subjects being discussed are.

From the workshops, create a one-page performance vision for general agreement.

Set Your Performance Targets

From the performance vision, extract some key KPIs that you are going to focus on, and set realistic, achievable targets for them.

Ensure that the first improvements that you target are a nice mix of quick wins from which the business benefits. This will enable the rest of the business to see progress and value as soon as possible.

Create Regular Reports on KPIs

Institute a regular proactive reporting mechanism on your progress against KPIs, especially one that highlights success. This could be a regular email that you send out across the business, an item displayed on notice boards, inclusion in regular company newsletters, or mentions in company update meetings.

In whatever manner you deliver the reports, you should get the message of progress to the wider business. This is essential to get the culture of performance accepted throughout the business, which will avert pushback when requested functionality is delayed because of performance-related issues.

Revise Your User Story Gathering/Specification Process to Include Performance Acceptance Criteria

The earlier you start thinking about performance, the better, so it is important to start building this into the process as early as possible. Performance acceptance criteria should be included within the other standard NFRs that are determined before development starts.

Re-evaluate Your "Definition of Done" to Include Performance Acceptance Criteria

Many agile development teams now have a published "definition of done." This details all the actions that have to be completed before a piece of work can be declared done. This ensures that when work is declared as done, it is release ready, not just development complete.

Any team that has a definition of done should expand it to require that sufficient testing has been completed to ensure that the stated performance acceptance criteria of the application have been met.

This could be that as part of the sprint, you have created automated performance tests that will validate that your performance acceptance criteria have been met and fail builds until those have been passed. On a simpler level, it could be that a set of performance tests have been executed and the results manually analyzed or that performance code reviews have been carried out by other developers or performance engineers.

Phase 4 : Engage
"Test...Test Early. . . Test Often..."

You have now defined good performance. You can't just sit back and hope that the development you have done is good enough to hit those targets. A good performance warrior will now start testing to determine that they are being met ahead of pushing to production.

This opens up a whole new set of challenges that could fill a whole book (that book is *The Art of Performance Testing* by Ian Molyneaux (O'Reilly), an essential read for any performance warrior), so I'll just present a quick overview of some of the issues to be addressed.

Challenges of Performance Testing

Tooling

Performance testing tools range from open source to very, very expensive, each of them having their pros and cons. Some rely heavily on scripts and are aimed more at competent developers, whereas others are more drag-and-drop, aimed at less technical people. Some target particular technologies, such as Citrix-based systems.

For general web-based systems, JMeter is a good starting point. It is open source, has a reasonable learning curve, and has built up a good community to go to for support.

Environments

The environment that you test on can make a big difference. Obviously, if your live system runs on 10 quad core servers, each with 64

GB of RAM, testing on a single dual core server with 32 GB RAM (or just on your laptop) will not get the same results. It doesn't invalidate the testing, but you need to scale down your expectations.

Other aspects of your environment beyond hardware also have to be considered. Think also about infrastructure (are you going through the same load balancer, firewalls, switches, bandwidth, etc.?), data quantities (are you testing with 10 products when there are 100,000 in production?), contention (are their other systems that share elements of the system under tests?), and so on.

Often, creating a reasonable performance-testing environment is difficult for logistical or economic reasons, so you may need to think out of the box. Cloud environments are a good option to quickly spin up and down large platforms. Disaster recovery (DR) environments are another option if they can be temporarily used for performance testing.

Some companies actually use their own production environments (after all, what could be more production like?) during periods of low or no usage. Many subtleties and risks have to be considered before doing this, particularly, how you minimize the impact on real users during that period and how you isolate any data created by performance testing from production data.

User Journeys

Performance testing is usually based around replicating user journeys through your system to simulate multiple actions by multiple users simultaneously. It takes effort to simulate these actions in a manner that reflects what will happen on the production system.

Getting these as representative as possible is a complex task, especially for a greenfield project where much is based on conjecture and guesswork. For existing systems, server logs and analytics (e.g., Google Analytics) provide an excellent starting point.

It is important that a wide enough range of representative user journeys is created with sufficient randomization of data to ensure that the system under test is being effectively exercised while not invalidating the test by making it not repeatable.

Load Model

One of the most complex elements of performance testing is generating the pattern and levels of users to execute the user journeys. This is the *load model*.

On existing systems, server logs and analytics packages can be a good starting point.

It is essential that this is realistic and you create a valid number of users, acting in a realistic manner with realistic think times between each step of their journey. If your performance targets are focused around hitting a target level of transactions per second, then it is possible to reverse engineer the load model to determine how many users will be required to hit that level.

NOTE

A reasonable test execution plan will include multiple load models to mimic different patterns of usage such as normal load and peak load.

Types of Tests

There are different type of tests, and it is essential when executing a test that you are aware of the type of test that you are running and have defined what you hope to determine from that test.

Some examples of the types of performance test that can be executed are:

Load test
: A test to determine how a system performs when a specified level of traffic is executed against it.

Soak test
: A long-running test to determine the ongoing performance of a system over a longer period of time, illustrating issues, such as memory leaks, that become apparent only over a long period of time.

Capacity test
: A test that escalates levels of load being applied until a system reaches a breaking point.

Iterative Improvements to Performance Testing

All these elements create hurdles that must be overcome, but they are all surmountable. The important message here is that some testing is better than none. Once you are doing some testing, work on a program of continuous improvement and refinement until you are seeing more and more value from the testing you are doing.

During the early stages, you will likely see false positives (i.e., tests that indicate potential performance problems that don't materialize on production). You also may see performance issues on production that were not caught in testing. This does not invalidate the testing process; it just reveals that you need to understand why the test failure have occurred and evolve your testing process to mitigate that in future.

At each step of improving your testing process, the important questions to ask yourself are:

- What aspects of this test may mean that the results will not be representative of live use? How may these be mitigated to make the result more reliable?
- How are these tests working toward validation of my performance KPIs?

Test Early

The traditional role of performance in the development cycle was to get a release signed off through functional testing and complete a performance testing process prior to going live. This approach leads to a some fundamental problems:

- The project is generally regarded as finished by this point. Testing at the end of the project inevitably gets squeezed as previous steps overrun their deadlines. This results in:
  - Lack of time to realistically run the tests.
  - Resistance against dealing with any issues that come out of the testing because delays threaten deadlines.
- Performance issues often are caused by underlying architectural issues and are therefore much harder to fix than functional issues.

These two factors combine to create the perfect storm. You are finding bugs that are the hardest to fix, at the time when they are hardest to fix, at a point when people are not inclined to want to spend additional time doing major changes, all to be completed in a period of time that is constantly squeezed by the earlier phases.

Having said all that, why do many companies still insist on testing only at the end of projects? Arguments are usually based around one or more of the following reasons:

- Performance testing is hard. It takes additional time and effort to generate scripts. There is no point in doing this until there is a stable application to test against.
- There is only limited access to an environment for testing.
- There is no point in performance testing earlier; the app is only a development version and hasn't been performance optimized yet.

On the surface, all of these arguments have validity, but the same claims could be made for any kind of testing. Nevertheless, the agile movement has consistently shown that earlier functional testing results in faster, more reliable development.

The traditional approach of "finish development, go through a load testing process, and approve/reject for go-live" really doesn't work in a modern development environment. The feedback loop is just too slow.

Alternative Methods of Early Performance Testing

As performance warriors, you need to be looking at methods to execute performance testing earlier. However, the arguments against executing complete performance tests earlier in the process have a degree of validity, so it is worth considering other methods of validating performance at that stage.

Examples include:

- Waterfall chart analysis of pages being returned that can be automated using a tool such as WebPagetest or manually using Firebug or Chrome Developer Tools. See Chapter 5 for more information on these tools.
- Adding timers around your unit tests. Monitor for trends toward poor performance and for failures to hit a performance

threshold. This can also be used for integration testing if you are running tools such as Cucumber.
- Parallelization of unit tests to get some idea of concurrency impacts.
- Using performance monitors or application performance management (APM) in continuous integration (CI) environments to get an understanding of what is happening within your application. See Chapter 5 for more information on these tools.

Many problems occur only under heavy loads, and these kinds of testing won't encounter such conditions, but they may provide some early indications of such problems.

These low-level methods should be augmented with performance testing with higher volumes of traffic as early in the process as possible.

As well as testing earlier, it is also important that the performance engineering team has a lot of integration with the development team, for both practical and political reasons. As discussed in "Assign Someone with Responsibility for Performance Within the Project" on page 13, there are several ways of integrating a performance engineer into the team.

When issues are identified, performance engineers and developers must cooperate and share their knowledge to resolve the problem. Performance engineers should not just identify problems; they must be part of the solution.

Performance Engineers Still Need Space for Analysis

One of the downsides to pushing performance testing at early stages is that it often results in additional testing without appropriate space for analysis. Analysis of performance testing is important to expose testing's insights; performance is not based on black-and-white results.

An oft-cited but still forgotten principle is that data is not information. Human intelligence is required to convert data into information. An important role of the performance engineer in improving any process is to ensure that the extra step of creating information is taken. Data is too often the focus of attention because it can be provided more regularly. As a performance warrior, you must ensure

sufficient quality, not just quantity, of performance testing during the development process.

Data versus Information

Data
: This second test run had average response times 12% higher than the previous run.

Information
: The second test run had a higher average response time primarily due to the "SearchProducts" transaction spiking 10 minutes into the test at the point where concurrent executions for this transaction exceeded 100. In the previous test run, this spike didn't occur.

Considering running two levels of analysis on performance test results:

1. An automated level of analysis that compares the data against a set of KPIs at a high level and gives a RAG score. (Red flags a serious problem, amber indicates something to watch in further testing, and green means the system works as desired.) This is useful for running regular performance tests as part of a CI process.
2. A more formal level of analysis done by a human who then recommends actions and more in-depth assessments of the actual results.

Test Often

It is an often-heard motto within the agile and continuous delivery world: if something is hard to do, do it early and do it often!

The same is true of performance testing. The more often you can test and the less of an special event a performance test becomes, the more likely you are to uncover performance issues in a timely manner.

Cloud and other virtualized environments, as well as automation tools for creating environments (e.g., Chef, Puppet, and CloudFormation), have been game changers to allow earlier and more regular

performance testing. Environments can be reliably created on demand. To make testing happen earlier, we must take advantage of these technologies. Obviously you must consider the cost and licencing implications of using on-demand environments.

We can also now automate environment setup, test execution, and the capture of metrics during the test to speed up the analysis process. APM tooling helps in this respect, giving easy access to data about a test run. It also allows the creation of alerts based on target KPIs during a test run.

Adding Performance to a Continuous Integration Process

Once performance testing gets added to the tasks of the test group, the obvious next step for anyone running a CI process is to integrate performance testing into it. Then, with every check-in, you will get a degree of assurance that the performance of your system has not been compromised.

However, there are a number of challenges around this:

Complexity
: Full-scale performance tests need a platform large enough to execute performance tests from and a platform with a realistic scale to execute tests against. This causes issues when running multiple tests simultaneously, as may be the case when running a CI process across multiple projects.

  Automation can solve these problems by creating and destroying environments on demand. The repeatability of datasets also needs to be considered as part of the task. Again, automation can be used to get around this problem.

Cost
: Spinning up environments on demand and destroying them may incur additional costs. Automating this on every check-in can lead to levels of cost that are very difficult to estimate.

  Many performance test tools have quite limited licensing terms, based on the number of test controllers allowed, so spinning up multiple controllers on demand will require the purchase of additional licenses. The development team needs to consider

these costs, along with the most cost-effective way to execute performance tests in CI.

One solution to this is to use open source tools for your CI performance tests and paid tools for your regular performance tests. The downside is that this requires the maintenance of multiple test script packs, but it does, however, enable you to create simplified, focused testing that is CI specific.

Time

CI is all about getting a very short feedback loop back to the developer. Ideally this is so short that the developer does not feel it is necessary to start working on anything else while waiting for feedback. However, performance tests usually take longer than functional tests (30 minutes is a typical time span); this is increased if they first involve spinning up environments.

A solution for this is to run simplified performance tests with every check-in. Examples include unit tests timings, micro-benchmarks of small elements of functionality, and WebPagetest integrations to validate key page metrics. You can then run the full performance test as part of your nightly build, allowing the results to be analysed in more detail by performance engineers in the morning.

Pass/fail orientation

CI typically relies on a very black-and-white view of the world, which is fine for build and functional errors. Either something builds or it doesn't; either it is functionally correct or it isn't.

Performance testing is a much more gray area. Good versus bad performance is often a matter of interpretation. For CI, performance testing needs to produce more of a RAG result, with a more common conclusion being that the matter is worthy of some human investigation, rather an actual failure.

Trends

Adopting a spectrum of failure over a pass/fail solution requires people to investigate data trends over time.

The performance engineer needs access to historical data, ideally graphical, to determine the impact of previous check-ins to enable the engineer to be able to get the the root cause of the degradation.

Functional testing rarely needs a look back at history. If a functional test previously passed and now fails, the last change can be reasonably blamed. The person who broke the build gets alerted and is required to work on the code until the build is fixed.

A performance issue is not that black and white. If you consider a page load time of 10 seconds or more to be a failure, and the test fails, the previous check-in may merely have taken page load time from 9.9 to 10.1 seconds. Even though this check-in triggered the failure, a look back at previous check-ins may turn up a change that took the page load time from 4.0 to 9.9 seconds. Clearly, this is the change that needs scrutiny. Another alternative is to look at percentage increments rather than hard values, but this has its own set of problems: a system could continuously degrade in performance by a level just below the percentage threshold with every check-in and never fail the CI tests.

So performance testing departs from the simple "You broke the build, you fix it" model driving many CI processes.

Action Plan

Start Performance Testing

If you're not currently doing any performance testing, the first step is to start. Choose a free toolset or a toolset that your organization already has access to and start executing some tests. In the short term, this process will probably ask more questions than it answers, but it will all be steps in the right direction.

Standardize Your Approach to Performance Testing

Next evolve your performance-testing trials into a standard approach that can be used on most of your projects. This standard should include a definition of the types of tests you will run and when, a standard toolset, a policy about which environments to use for which types of tests and how they are created, and finally, an understanding of how results will be analyzed and presented to developers and managers. If your development is usually an evolution of a base product, look at defining a standard set of user journeys and load models that you will use for testing.

This standard will not be set in stone and should constantly change based on specific project needs. But it should be a good starting point for performance testing on all projects.

Consider Performance Testing at Project Inception

In addition to defining the performance acceptance criteria described in "Performance Acceptance Criteria" on page 21, the project's specification stage must also consider how and when to do performance testing. This will enable you to drive testing as early as possible within the development process. At all points, ask the following questions while thinking of ways you can do elements of the testing earlier without investing more time and effort than would be gained by the early detection of performance issues:

- Will you need a dedicated performance test environment? If so, when must this be available?
- Can you do any lower-level testing ahead of the environment being available?

Look at the performance acceptance criteria and performance targets and determine how you will be able to test them. What levels of usage will you be testing, and what user journeys will you need to execute to validate performance? How soon can scripting those user journeys start? What data will you need to get back to determine success? Will your standard toolset be sufficient for this project?

Integrate with Your CI Process

If you are running a CI process, you should try to integrate an element of performance testing within it. As described earlier, there are a lot of issues involved in doing this, and it takes some thought and effort to get working effectively.

Start with small steps and build on the process. Do not fail builds until there is a degree of trust that the output from the tests is accurate and reliable. Always remember that the human element will be needed to assess results in the gray area between pass and fail.

Phase 5 : Intelligence *"Collect Data and Reduce Guesswork"*

Testing will show you the external impact of your system under load, but a real performance warrior needs to know more. You need to know what is going on under the surface, like a spy in the enemy camp. The more intelligence you can gather about the system you are working on, the better.

Performance issues are tough: they are hard to find, hard to replicate, hard to trace to a root cause, hard to fix, and often hard to validate as having been fixed. The more data can be uncovered, the easier this process becomes. Without it you are making guesses based on external symptoms.

Intelligence gathering also opens up a whole new theater of operations – you can now get some real-life data about what is actually happening in production. Production is a very rich source of data, and the data you can harvest from it is fundamentally different in that it is based on exactly what your actual users are doing, not what you expected them to do. However, you are also much more limited in the levels of data that you can capture on production without the data-capture process being too intrusive.

Chapter 6 discusses in more detail the types of data you can gather from production and how you should use that data.

During development and testing, there is much more scope for intrusive technologies that aim to collect data about the execution of programs at a much more granular level.

Types of Instrumentation

Depending on how much you're willing to spend and how much time you can put into deciphering performance, a number of instrumentation tools are available. They differ in where they run and how they capture data.

Browser Tools

Client-side tools such as Chrome Developer Tools, Firebug, and YSlow reveal performance from the client side. These tools drill down into the way the page is constructed, allowing you to see data such as:

- The composite requests that make up the page.
- An analysis of the timing of each element.
- An assessment of how the page rates against best practice.
- Timings for all server interactions.

Web-based tools such as WebPagetest will perform a similar job on remote pages. WebPagetest is a powerful tool that also offers (among many other features) the capability to:

- Test from multiple locations.
- Test in multiple different browsers.
- Test on multiple connection speeds.
- View output as a filmstrip or video: it is possible to compare multiple pages and view the filmstrips or video side by side.
- Analyze the performance quality of the page.

Typically the output from these tools is in the form of a waterfall chart. Waterfall charts illustrate the loading pattern of a page and are a good way of visualising exactly what is happening while the page executes. You can easily see which requests are slow and which requests are blocking other requests. A good introduction to understanding waterfall charts can be found at a posting from Radware by Tammy Everts. Figure 5-1 shows a sample chart.

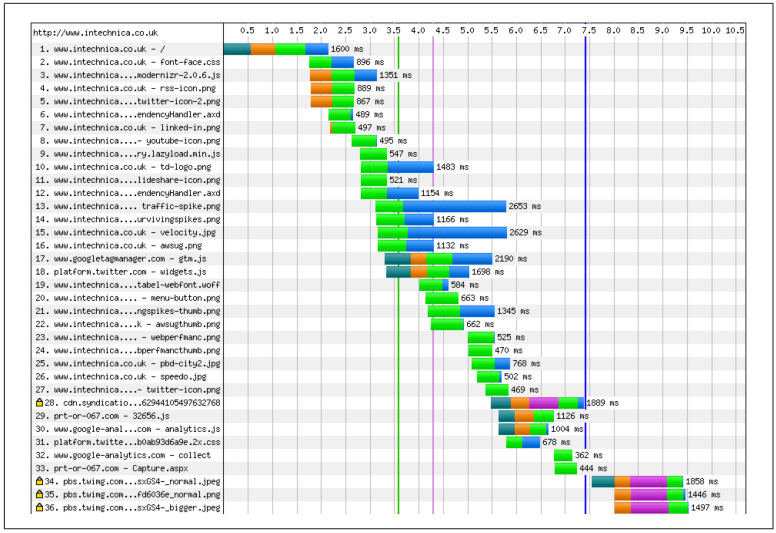

Figure 5-1. Example waterfall chart, in this case taken from WebPageTest

All of these tools are designed for improving client-side performance.

Server Tools

All web servers produce logfiles showing what page has been sent and other high-level data about each request. Many visualization tools allow you to analyze these logfiles. This kind of analysis will indicate whether you're getting the pattern of page requests you expect, which will help you define user stories.

At a lower level come built-in metrics gatherers for server performance. Examples of these are Perfmon on Windows and Sar on Linux. These will track low-level metrics such as CPU usage, memory usage, and disk I/O, as well as higher-level metrics like HTTP request queue length and SQL connection pool size.

Similar tools are available for most database platforms, such as SQL Profiler for SQL Server and ASH reports for Oracle.

These tools are invaluable for giving insight into what is happening on your server while it is under load. Again, there are many tools available for analyzing trace files from these tools.

These tools should be used with caution, however, as they add overhead to the server if you try to gather a lot of data with them.

Tools such as Nagios and Cactii can also capture this kind of data.

Code Profilers

For a developer, code profilers are a good starting point to gather data on what is happening while a program is executing. These run on an individual developer's machine against the code that is currently in development and reveal factors that can affect performance, including how often each function runs and the speed at which it runs.

Code profilers are good for letting developers know where the potential pain points are when not under load. However, developers have to make the time and effort to do code profiling.

Application Performance Management (APM)

In recent years there has been a growth in tools aimed specifically at tracking the underlying performance metrics for a system. These tools are broadly grouped under the heading APM.

There are a variety of APM toolsets, but they broadly aim to gather data on the internal performance of an application and correlate that with server performance. They generally collect data from all executions of a program into a central database and generate reports of performance across them.

Typically, APM tools show execution time down to the method level within the application and query execution time for database queries. This allows you to easily drill down to the pain points within specific requests.

APM is the jewel in the crown of toolsets for a performance engineer looking to get insight into what is happening within an application. Modern APM tools often come with a client-side element that integrates client-side activities with server-side activities to give a complete execution path for a specific request.

The real value in APM tooling lies in the ability it gives you to remove guesswork from root-cause analysis for performance problems. It shows you exactly what is going on under the hood. As a performance engineer, you can extrapolate the exact method call

that is taking the time within a slow-running page. You can also see a list of all slow-running pages or database queries across all requests that have been analyzed. Many tools also let you proactively set up alerting on performance thresholds. Alerting can relate to hard values or spikes based on previous values.

There is overhead associated with running these kinds of tools, so you must be careful to get the level of instrumentation right. Production runs should use a much lower level of instrumentation. The tools allow you easily to increase instrumentation in the event of performance issues during production that you want to drill into in more detail.

On test systems, it is viable to operate at a much higher level of instrumentation, but retain less data. This will allow you to drill down to a reasonable amount of detail into what has happened after a test has run.

Action Plan

Start Looking Under the Hood During Development

Start by using the simpler tools that are easier to integrate (e.g., client tools and code profilers) within your development process to actively assess the underlying performance quality of what you are developing. This can be built into the development process or form part of a peer/code review process.

Include Additional Data Gathering as Part of Performance Testing

As part of your performance-testing process, determine which server-side stats are relevant to you. At the very least, this should include CPU usage and memory usage, although many other pieces of data are also relevant.

Before starting any tests, it may be necessary to trigger capturing of these stats, and after completion, they will need to be downloaded, analyzed, and correlated to the results of the test.

Install an APM Solution

APM tooling is an essential piece of the toolkit for a performance warrior, both during testing and in production. It provides answers for a host of questions that need answering when creating performant systems and doing root-cause analysis on performance issues.

However, the road to successful APM integration is not an easy one. The toolsets are complex and require expertise to get full value from them. A common mistake (and one perpetrated by the vendors) is to think that you can just install APM and it will work. It won't. Time and effort need to be put into planning the data that needs to be tracked. You also need training, time, and space to learn the system before performance engineers and PerfOps engineers can realize the tool's potential.

Phase 6: Persistence
"Go Live Is the Start of Optimization"

There has traditionally been a division between the worlds of development and operations. All too often, code is thrown over the wall to production, and performance is considered only when people start complaining. The DevOps movement is gaining traction to address this issue, and performance is an essential part of its mission.

There is no better performance test than real-life usage. As a performance warrior, you need to accept that pushing your code live is when you will really be able to start optimizing performance. No tests will ever accurately simulate the behavior of live systems.

By proactively monitoring, instrumenting, and analyzing what's happening in production, you can catch performance issues before they affect users and feed them back through to development. This will avoid end-user complaints being the point of discovery for performance problems.

Becoming a PerfOps Engineer

Unlike functional correctness, which is typically static (if you don't change it, it shouldn't break), performance tends toward failure if it is not maintained.

Increased data, increased usage, increased complexity, and aging hardware can all degrade performance. The majority of systems will

face one or more of these issues, so performance issues are likely if left unchecked.

To win this battle, you need to ensure that you are capturing enough data from your production system to alert you to performance issues while they happen, identify potential future performance issues, and find both root causes and potential solutions. You can then work with the developers and ops team to implement that solution.

This is the job of the *PerfOps engineer*.

Just as DevOps looks to bridge the gap between development and operations, PerfOps looks to bridge the gap between development, performance, and operations.

PerfOps engineers need a good understanding of the entire application stack, from client to network to server to application to database to other components, and how they all hook together. This is how to determine where the root cause of performance issues lies and where future issues may arise.

The PerfOps Engineer's Toolbox

Given that performance is such a complex and subtle phenomenon, you have to be able to handle input from many types of tools, some general-purpose and some more dedicated to performance.

Server/network monitoring

The Ops team will more than likely have a good monitoring system already in place for identifying issues on the server/network infrastructure that you are using. Typically this will involve toolsets such as Nagios or Cactii, or proprietary systems such as HP System Center.

These systems will probably focus on things such as uptime, hardware failure, and resource utilization, which are slightly different from what you need for proactive performance monitoring. However, the source data that you will want to look at will often be the same, and the underlying systems are capable of handling other data sources that you will need.

Real-user monitoring (RUM)

RUM captures data on the actual experience that users are getting on your system. With this technology, you can get a real understanding of exactly what every (or a subset) of users actually experienced when using your system.

Typically, for web-based systems, this works by injecting a piece of JavaScript into the page that gathers data from the browser and transmits it back to the collection service, which aggregates and reports on it. There are now also RUM systems that integrate into non-web systems, such as native apps on mobile devices that give the same type of feedback.

Some of the newer RUM tools will integrate with APM solutions to get a full trace of all activity on the client and the server. This allows you to identify specific issues and trace the root cause of the issue, whether on the client, on the server, or a combination.

RUM tools are especially useful for drilling down into performance issues for subsets of users that may not be covered by testing or issues that may be out of the scope of testing. For example:

Geographic issues
: If users from certain areas see more performance issues than other users, you can put solutions in place to deal with this. Perhaps you need to target a reduced page size for that region or introduce a CDN. If you already use a CDN, then perhaps it is not optimally configured to handle traffic from that region, or an additional CDN is needed for that region.

Browser/OS/device issues
: RUM will turn up whether certain browsers have performance issues (or indeed, functional issues), or whether the problems stem from certain devices or operating systems. Most likely, it will be combinations of these that lead to problems for particular individuals (e.g., Chrome on Mac OS X or IE6 on Windows XP).

It is important to realize that RUM is run by real users on real machines. It is not a clean room system (i.e., there are other external activities happening on the systems that are out of your control, meaning results could be inconsistent). Poor performance on an individual occasion could be caused by the user running a lot of other programs or downloading BitTorrent files in the background.

RUM is also affected by "last mile" issues, the variance in speed and quality in the connection from the Internet backbone to the user's residence. RUM depends on having a large sample size so you can ignore the outliers.

The other weakness of RUM is that performance problems will become known only when they are already experienced by users. It doesn't enable you to capture and resolve issues before the users are affected.

Synthetic monitoring

Synthetic monitoring involves executing a series of transactions against your production system and tracking the responses. Transactions can be multiple steps and involve dynamically varied data. Synthetic monitoring can evaluate responses to determine next steps. Most solutions offer full scripting languages to enable you to build complex user journeys.

As with RUM, synthetic monitors will integrate with APM to enable you to see the full journey from client to server.

Synthetic monitors can be set up to mimic specific geographic connections as well as browser/OS/device combinations, and you can often specify the type of connection to use (e.g., Chrome on a Galaxy S3 connecting over a 3G connection).

Synthetic testing can also be "clean room" testing, usually executed from close to the Internet backbone in order to remove "last mile" problems.

Unlike RUM, it allows you to proactively spot issues before users have necessarily seen them. However, it is limited to testing what you have previously determined is important to test. It will not detect issues outside your tests or issues that users will encounter when performing actions or running device/browser combinations you did not anticipate.

An ideal monitoring solution combines synthetic monitoring and RUM.

APM tooling

APM tooling, as described in "Application Performance Management (APM)" on page 42, is the central point of data gathering for many of the tools described here. While it does not 100% replace

other tooling, it does work well in aggregating high-level results and correlating results from different sources.

The PerfOps Center

In the same way as your company may have a dedicated networks operations center (NOC), it is a good idea to create a PerfOps center.

This doesn't have to be a physical location but should be a central gathering point for all performance-related data in a format understandable by your staff, and with capability of drilling down to more detail if needed. This will gather data from other monitoring, RUM, and APM tools into one central point. A good PerfOps Center can do predictive and trend-based analysis of performance-related data.

Closing the PerfOps Loop to Development

It is essential that, having gathered all the data and proactive monitoring, you feed useful information back through to development and work with the development team on solution to the problems identified. The developers must be warned of performance issues, whether actual or potential. This information describes the performance problem and the source of the data that has been used to identify the problem.

Action Plan

Put Proactive Monitoring in Place

Create a monitoring strategy that gathers sufficient data to be able to become aware of performance issues as early as possible and be alerted when they are happening. Being alerted to a performance issue by an end user should be seen as a failure. In addition to the symptoms of the problem that is happening, you should have sufficient data captured to be able to do some root-cause analysis of the underlying cause of the problem.

Carry Out Proactive Performance Analysis

Regularly revisit the data that you are getting out of your systems to look for performance issues that have gone unidentified and trends towards future performance issues. Evaluate your performance

against the defined KPIs. Again, these should be identified and you should include root-cause analysis.

Close the Gap Between Production and Development

It is essential to provide a pipeline through to development for issues identified by the PerfOps engineer. The PerfOps engineer must also be involved in developing the solution, especially when replicating and validating the fix. Pairing programmers and PerfOps engineers for the duration of completing the fix is a good strategy.

Create a Dedicated PerfOps Center

Investigate the creation of a dedicated PerfOps center as a central point for all performance-related data within the company. The center can be used for analysis of performance test data on test and preproduction platforms as well. This builds upon the earlier theme of treating performance as a first-class citizen, as well as creating a focal point and standardized view of performance that can be accessed by more than just PerfOps engineers.

About the Author

Andy Still has worked in the web industry since 1998, leading development on some of the highest traffic sites in the UK. After 10 years in the development space, Andy cofounded Intechnica, a vendor-independent IT performance consultancy that focuses on helping companies improve performance on their IT systems, particularly websites. Andy focuses on improving the integration of performance into every stage of the development cycle, with a particular interest in the integration of performance into the CI process.

O'REILLY®

Wait. There's More.

4 Easy Ways to Stay Ahead of the Game

The world of web operations and performance is rapidly changing. Find what you need to keep current at **oreilly.com/velocity**:

More Reports Like This One

Get industry intelligence in timely, focused reports written to keep you apprised of the current and trending state of web operations and performance, best practices, and new technologies.

Videos and Webcasts

Hear directly from some of the best minds in the field through free live or pre-recorded events. Watch what you like, when you like, where you like.

Weekly Newsletter

News happens fast. Get it delivered straight to your inbox so you don't miss a thing.

Velocity Conference

It's the must-attend event for web operations and performance professionals, happening four times a year in California, New York, Europe, and China. Spend three supercharged days with the best minds, companies, and people interested in the same things you are. Learn more at **velocityconf.com**.