Early Performance Testing

Eric Proegler

eproegler@gmail.com, @ericproegler

Abstract

Development and deployment contexts have changed considerably over the last decade. The discipline of performance testing has had difficulty keeping up with modern testing principles and software development and deployment processes.

Most people still see performance testing as a single experiment, run against a completely assembled, code-frozen, production-resourced system, with the "accuracy" of simulation and environment considered critical to the value of the data the test provides. But what can we do to provide actionable and timely information about performance and reliability when the software is not complete, when the system is not yet assembled, or when the software will be deployed in more than one environment?

Biography

I have worked in testing for 15 years, and specialized in performance and reliability testing for 12. I work for Mentora Group, a national testing consultancy, from my home in Mountain View, California.

I design and conducts experiments that use synthetic load to help customers assess and reduce risk, validate architecture and engineering, and evaluate compliance with non-functional systems requirements. I test in a wide variety of contexts, using tools appropriate to the job at hand. Some of the applications I've tested recently include Oracle E-Business ERP systems, a hospital's provider portal, a large scale "second-screen" mobile app, a large B2B EDI Translation clearing house, a custom SaaS application, and e-Commerce web sites.

I'm an organizer for WOPR, the Workshop on Performance and Reliability, and a Community Advisory Board member for STPCon. I've presented and facilitated at CAST, WOPR, STPCon, and STiFS. I have recently started engaging with tester certification and testing standards to help our community best respond to these economic tactics.

In my free time, I spend time with my family, read, see a lot of stand-up comedy, seek out street food from all over, play video games, and follow professional basketball.

1 Introduction

Iterative development processes has changed software testing from a gating activity occurring at the end of a development process into a collaborative one that occurs throughout the development process. Testers participate earlier in projects, and the role of automation in testing has increased. The orchestration of building software, installing it to test environments, and running extensive automated checks without human intervention is now widely practiced. Software under test is changing frequently or even constantly; some would say continuously. What hasn't happened broadly is the adaptation of performance testing practices to keep up with these new development methodologies.

A traditional performance test involves creation of a high-fidelity simulation of production workload, run against software that is fully developed and has completed functional testing, executing on hardware that is equivalent in resource capacity and configuration to the production environment. Realism can be simulated to various levels of "accuracy", depending on time and budget, with various degrees of success, depending on available information about the system under test.

1.1 Challenges in Simulation Tests

We will call this approach a "Simulation Test" for the rest of this paper, to distinguish from a "Load Test". A Load Test is a multi-threaded automation test that could be a Simulation Test, or it could be some other load that is not intended to Simulate a specific workload. Load testing tools are used to create Load Tests; the application of a "Realistic" user model, test environment, and completed software is necessary for a Load Test to become a Simulation Test, as we mean it here.

A frequently encountered challenge with Simulation Tests is that testing doesn't take place until software is completed. Waiting until software is complete means that testing occurs at the very end of the project. This exposes testing to time pressure, particularly when earlier phases of the project run past milestones.

Finding problems at a point in the schedule where the testing is the last gate to going live is of limited value. Either they are minor enough to be deferred, or serious enough to delay the entire project.

This problem is shared with functional testing, and was one of the major factors that led to the adoption of iterative software development processes. Most people still see performance testing as a single experiment, run against a completely assembled, code-frozen, production-resourced system, with the "accuracy" of simulation and environment considered critical to the value of the data the test provides. But what can we do to provide actionable and timely information about performance and reliability when the software is not complete, when the system is not yet assembled, or when the software will be deployed in more than one environment?

1.2 About This Paper

This paper describes approaches and techniques for providing performance feedback in iterative development processes, particularly when software isn't complete, components are not yet implemented, the production workload is not known, and/or production-class hardware is not available.

- *Section 2: Risks Addressed by Performance Testing*
- *Section 3: Getting Real About Realism*
- *Section 4: Performance Testing Iteratively*
- *Section 5: Alternative Performance Testing Techniques*
- *Section 6: Testing Incomplete Systems*
- *Section 7: Communicating Performance During Iterative Development*

2 Risks Addressed by Performance Testing

To engage with these new contexts, we should consider what risks lead us to performance testing in order to mitigate them. Here is one formulation of the classes of performance and scalability risks:

2.1 Scalability – Expensive Operations

If an activity take a long time to complete, that is, has a response time of a second or longer, it is likely to be of interest for optimization. This is because these longer running activities will almost certainly create a heavy computational load on the server, depending on their frequency.

These types of issues are often a result of implementing tall stacks of scripting languages, technical frameworks, and APIs. Situations like this are opportunities to observe Wirth's Law: "Software is getting slower more rapidly than hardware gets faster."

When inefficiency is introduced, an increase in response time is the most likely way to find it. When inefficiency increases resource pressure, it can be difficult to trace resources back to a specific activity.

2.2 Capacity – System's Ability to Handle Load

In Simulation Tests (a single complex experiment that attempts to simulate a full workload), capacity is usually what is being tested. The capacity of a system is determined by a combination of hard and soft resources. When a resource is exhausted, the first limitation is encountered. This is where the system's capacity limit is until that resource is increased. These kinds of problems are often first presented as response times that are very sensitive to load.

Sometimes this first limitation is encountered against a hard resource, such as available CPU, memory, storage, or network capacity. This is the reason why "realism" in performance tests is prized; if a resource is available in less quantity than it will be in production, the limitation occurs sooner than it should.

Often, the bottleneck is a soft resource instead, such as a limitation on the number of database connections or sessions, licensing restrictions, or the configuration of runtime environments such as Java. Tests can expose these limitations, allowing for configuration changes or code refactoring to correct them or increase capacities.

2.3 Concurrency – Race Conditions and Contentions

A certain class of risk in software only appears under load in a multi-user system. These problems can be very difficult to recreate, but when they occur, they can cause significant problems for all users of the system, including causing the entire system to crash.

Generally speaking, they occur when a specific combination of states occur as a multi-user system serves multiple users. These conditions may be confused with very rarely occurring errors. Any rare set of conditions, whether based on concurrency or not, is much more likely to be found in load tests than typical (single-threaded) testing.

2.4 Reliability and Recoverability

Multi-user systems are required to service many thousands (sometimes millions) of sessions while maintaining acceptable performance characteristics, sometimes over extended periods of time. Problems such as memory leaks, disk space for database files, or programming logic errors can be exposed by long-running load tests, frequently called "Soak Tests". Most Simulation Tests end after an hour or two, meaning that they might not expose these problems if they are slower to appear.

3 Getting Real About Realism

Performance Testing, as traditionally practiced, is usually based on the idea that a single large-scale simulation (or series of repeated simulations) provides an opportunity to evaluate a multi-user system under operational conditions. Automated tools are used to create load against a system, making it behave similarly to how it will when users are operating the system after go-live. This thinking places a great deal of value on "Realism", or the degree to which the simulation is thought to resemble expected operational conditions. "Realism" is a very loaded term that implies several things.

The idea that performance testing must achieve "Realism" in order to have value is inaccurate, outdated, and often impossible. It must be set aside when we are testing rapidly changing or incomplete software, when we don't know what production will look like, or when testing against a different environment and configuration than we expect in production.

The Simulation Test constructs a scenario with many variables, attempting to create equivalent load on an equivalent system in order to create comparable conditions for evaluating the performance of the system under test. Under closer examination, creating an equivalency seems to become even more difficult. The model looks like this:

Will the complete, deployed System support:

(a, b) classes of users
performing (c, d) activities
at (e, f) rates
on (m) environment/configuration
under (n) external conditions
and meet x response time goals?

We usually believe those conditions to be the most representative conditions we can design with the knowledge we have in order to conduct a single performance experiment. We should be humble and cautious about how close to "Reality" and "User Experience" we actually get.

3.1 Realism in Workload

Variables (**a-f**) are components of the workload model. When a production system already exists, this information might be derived from log files, though some prediction of future state is still necessary. Without actual workload information, a model must be created, usually with input from a couple of sources, but always subject to possibly being very different from what reality turns out to be.

Whatever the source of the model and its "accuracy", load tooling imposes some conditions that are fundamentally different than how people use a system. Examples of this include rigid session arrival schedules, identical activity sequences per script, users that never abandon sessions, and traffic that is well distributed over a test instead of having organic resonances associated with start of day, end of day, lunch, and meeting times. Testers frequently reduce fidelity by simplifying the data used.

The author's experience is that load models are frequently intended, at least initially, to be an example of a peak hour on a peak day. Estimates, projections, and outright guesses are common at this stage. Stakeholders who may not have much knowledge of load modeling and/or the applications reality might insist on certain round number thresholds that must be accommodated. Finally, there is some rounding up, some "horse-trading" of less frequently used or difficult to script activities, and eventually, a model emerges that is believed to be representative – but turns out to not be very much like how it will be used as a live system.

Data density and caching is rarely considered. In production, some data will be accessed often, and some will be accessed rarely. Testing often attempts to achieve something like this, but rarely succeeds.

The nature of people using software is essentially ignored. People use different navigation paths to the same places, click multiple times, when software isn't fast enough, get distracted and delay or leave active sessions, and show a variety of behaviors that simulation ignores. Scripts execute software in identical fashion, avoiding wide swathes of what people would do in the software.

None of this accounts for how load evolves and varies over days, months, and years.

3.2 Realism in Environment

Other challenges are encountered in addressing variables (**m, n**) – the environment. Most people start off thinking about this in terms of machine resources being equivalent; too many insist on "Identical". When machine resources were dedicated to a specific group of physical servers, it was possible to assess and project available resources with some degree of accuracy. In a world where resources are abstracted in large, dynamically adjusted pools by virtualization, SANs (Storage Area Networks), and cloud-based infrastructure, this is an incredibly difficult task. Comparable resources are critical to assessing capacity, but it can be extremely difficult to understand how much capacity is available at any one moment – and it changes constantly.

This dynamism in deployed resources is due to the impact of other business activities that contend for the shared resources. Soft resources like authentication are also subject to pressures from other applications. The net result of sharing with all of the other applications is that the pool of available resources is changing second-to-second, based on demands from other applications that are almost certainly out of scope for the project.

Modern execution environments include virtualization software, storage firmware, operating systems, presentation frameworks, and many more software components. The trend of hardware is to be more and more abstracted by software. Not only does this increase the variability of available resources moment to moment, it means that there is a whole other class of environment configuration associated with software versions and patching.

3.3 Realism in Response Times

The nature of client software has also changed considerably. The most significant factors here include execution environments on mobile devices, the amount of code and interpreted code running at the client, and highly variable network conditions over wireless links. While server requests can be successfully abstracted by load tools, the reported response times do not include client rendering, which in many cases is now the majority of the human-experienced response time.

4 Performance Testing Iteratively

Simulation Tests as a final verification before go-live have difficult pre-requisites to meet in a waterfall process. In iterative development, quality feedback is desired throughout the process. Performance tests in this context have to yield information in minutes, not hours, in order to keep up with the speed of builds. To provide feedback at the speed of iterative development, it is necessary to make tests faster, simpler, cheaper, and more repeatable.

Once we've moved on from the expectation for simulated workloads to represent "Realism", we can design performance tests that meet these requirements. Instead of testing 10 workflows with specific numbers, proportions, and pacing of activity, we can test 1 at a time with tests designed for repeatability.

The concept I'd like to introduce here is Calibration – between builds, between environments, and between changed configurations. It is more important to track improvement and degradation as the project proceeds, and less important to attempt a specific projection of production experience. By tightening the feedback loop, we can provide actionable performance feedback while the changes that have been made are still well-known and fresh in people's minds.

4.1 Test Cases for Calibration

Test cases should be simple and easy to repeat. They should not seek to cover every permutation in the software, but to exercise the most commonly used paths in the software, and help reliably calibrate across builds of software.

Consider the example of establishing a user session. As software continues to be built, the login process for each user will include more and more code as a user session model becomes more sophisticated. As new elements are added, the efficiency and cost of processing the new elements can be evaluated. If a change is expensive, it can be quickly identified as such, and optimized or reworked.

As the available user session options increase, e.g., levels of application privilege, additional test cases can be added as appropriate.

4.2 Load Models for Calibration

Traditional approaches to performance testing use delays between activities, and between iterations. These delays serve an important purpose in spreading out load demand across a test, and make an individual session's flow more like a human-generated session.

Delays make it easy for the system under test to handle the load, and minimizes contention (See Section 2.3, Concurrency – Race Conditions and Contention). If we are testing to mitigate risk, that may be against our purpose. Delays make it harder to see small differences between two builds.

In some cases, we can better serve the goal of identifying small differences between builds by removing delays altogether. Consider a test of 10 threads, executing 10 iterations each. These 100 total iterations might consist of several steps each. Traditional performance testing spreads this test out over many minutes, using think times and pacing to make it easier for the server to handle.

Instead, we can start 10 threads with no delays and intentionally swamp the server. Then, we watch how long it takes the server to complete all the work, in addition to how long individual activities take. This is a more pure test of computation, by forcing the system to work as fast as it can throughout. This kind of test also helps mitigate concurrency risks, by creating more opportunities for collision between sessions.

Another approach would be 100 sessions run consecutively on a single thread with no delays. This will help expose even slight increases in response time by multiplying their effects.

4.3 Horizontal Scalability

Virtually every application uses a horizontal scaling pattern of multiple servers with load balancing to quickly increase capacity to the desired level (in addition to providing high availability). This pattern assumes 2 servers can support twice as many users as 1, and 4 can support four times.

This approach can be wound backwards. If production is going to support 500 users with 4 servers, 1 server should support 125. It is still necessary to validate the number of users 1 server can support in order to project how many servers to provision.

Testing with one server is easier to analyze, reduces the number of variables, and allows us to calibrate between software builds.

4.4 Testing Infrastructure

Many deployment contexts depend heavily on virtualization. This can greatly complicate repeatability. A recent desktop-class machine should be more than enough for calibrating results across builds and configurations reliably in a way that virtualization cannot match.

The goal shouldn't be "comparable" capacity – it should be reliable, repeatable capacity achieved through dedicated resources. By controlling variables tightly, we can trust our results now and in the future.

Load tools, separated components and services, even databases can be put on the same box as the primary system under test. This allows for the simultaneous measurement of all of the components at once, and removes the chance for network conditions or other demands to impact response times.

4.5 Performance Testing Tools and Performance Testing in CI

Performance testing tools are simply multi-threaded automation tools. They can be scripted to execute automatically, which means they can be added to a Continuous Integration (CI) process. Load Tool Vendors such as SOASTA and BlazeMeter are recommending their tools for this specific purpose.

Automating interpretation requires some thoughtfulness. What will be measured? What increase in response time should lead to further investigation (or turn a status yellow or red, depending on the process)? Who is the person who understands well enough what has changed in the last build that they can properly interpret and triage the change?

4.6 Extending existing/single-threaded automation

Any language used for automation should have the ability to record elapsed times between and during activities, even if it is of the form (endtime – starttime). This data can then be logged, providing a measurement of change across many builds. This also captures client-side and rendering time accurately, unlike load test tools.

Automation extends senses – but doesn't replace them. Watch for trends, and be ready to follow up and drill down when something has changed.

4.7 Recordkeeping

It is essential to be confident that results mean what they seem to mean. We can have that confidence if we can reproduce previous findings – or measure the delta. Environments and components change all the time. With careful records, we can reconstruct previous results, and use calibration techniques to measure the effects of specific components changing. We can also verify our instruments and environment by repeating and calibrating against previous tests.

5 Alternative Performance Testing Techniques

There are other techniques besides load testing for gathering performance feedback faster. The cost and startup time for getting load testing set up and started can be high. Some of these approaches can provide value that load testing can't. These techniques generally do not consider the resource characteristics of a loaded system or consider concurrency, but they instead look very closely at the responsiveness of a single user's interaction with the system.

5.1 Stopwatch

A simple, fast performance test is to use a stopwatch to record the time between a click, and when the software has completed servicing the request. These times can be retaken and reproduced very cheaply and quickly. This provides a quick, simple measurement of performance.

Quantifying user experience is frequently an implicit or explicit goal of load testing. Since load testing typically plays server requests back, it does not include client-side rendering and script execution time. Interacting with the software under test's interface captures client-side and rendering time accurately, unlike load test tools.

5.2 Extending Functional Automation

Adding timers to automation scripts is relatively simple. Setting (and abstracting) a threshold for flagging results is a more difficult. Recording the timers over time takes a little more work. The net result of this can be well worth the effort; responsiveness of these user activities can be tracked throughout the project.

When automation is GUI-based, instrumenting automation can provide user experience response times.

5.3 Screen Captures/Videos

Screen capture programs can record how software responds from a user perspective, preserving that exact experience indefinitely. For usability studies, this technique is invaluable; for performance testing without a load tool, it provides a mechanism for capturing user experience in a sharable way that is easy to share with non-technical stakeholders, and includes client-side and rendering time.

This technique can be leveraged with Load Testing to provide a specific representation of how the system performs at peak load, communicating real user experience in a way a table of response times cannot.

5.4 Log Files

Web-based systems write every request to a w3 log format, which can be configured to include server-response times (time-taken). Depending on the software under observation, other logging information might be available – or logging might be added early on as a supportability/testability story/requirement.

When a log file exists, it can be mined for data after the fact. It can be processed to generate summary data, and that summary data can be stored for future use in comparisons and trending.

5.5 Waterfall Charts

For websites and web applications, a waterfall chart can help understand the components of a web page. There are many ways to generate these, including free online sites like http://www.webpagetest.org/.

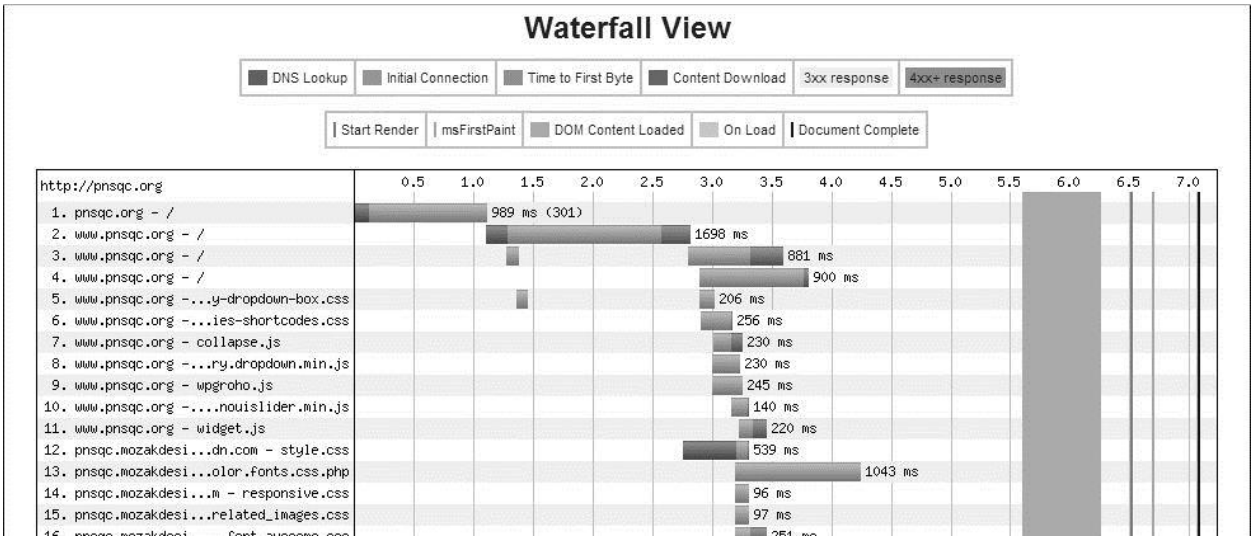

Figure 1: Waterfall Chart

These can be captured throughout the project, and used as a mechanism for tracking progress and identifying problem areas. Tools such as YSlow use this kind of data to suggest web page optimizations.

5.6 Browser Proxies

Browser proxies such as Telerik's Fiddler or Charles Proxy collect highly detailed data at the request level, including payload sizes and response times. Results can be compiled throughout a project.

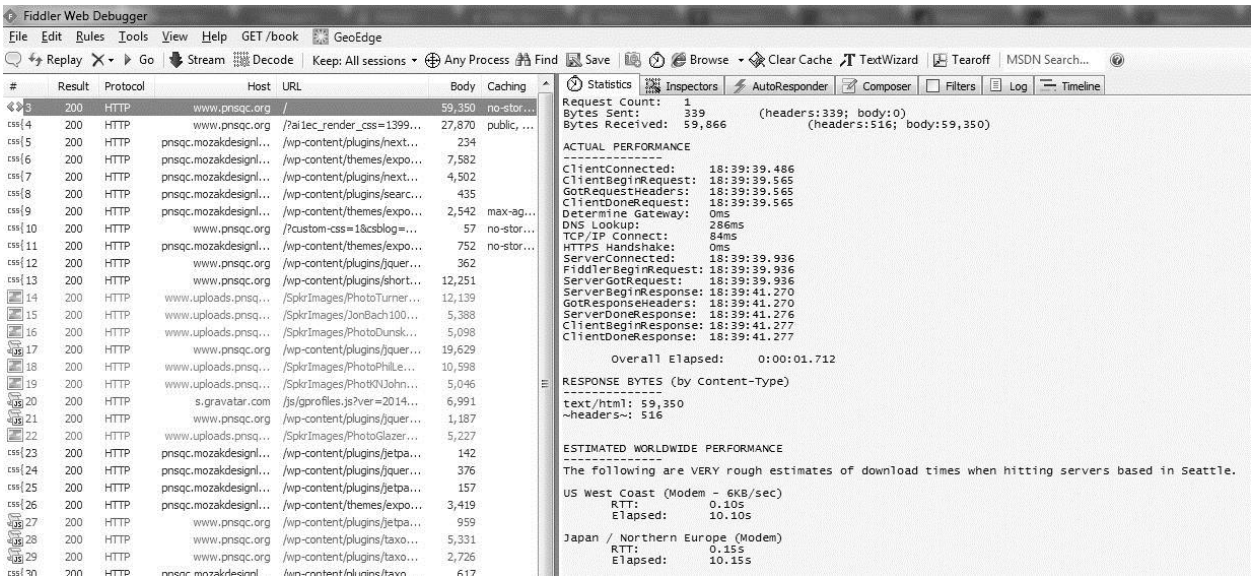

Figure 2: Fiddler Capture

5.7 Application Performance Management (APM)

APM is an approach to monitoring and measurement that captures response time and workload metrics from deployed software. By taking detailed measurements for every execution of the software, these tools can gather large aggregations of data at the provide insight into which pieces of a user activity take more time at any one point, and are responsible for increased response times.

These measurements are taken at the network level, at runtime (e.g., Java process instrumentation), as instrumentation included in code libraries, and increasingly, as defined business transactions. Feeding this data into big data platforms supports deep operational and analytical analysis, helping performance get better visibility outside of engineering groups and connect it more directly to productivity and revenue.

APM tools were initially conceived as an operations management tool, supporting performance troubleshooting of deployed software. Some of us immediately adapted APM for working with systems under test. Later tools were designed for Developers. Tools in this space include Compuware (DynaTrace, Vantage, etc), Riverbed's SteelCentral (formerly OPNET), AppDynamics, and New Relic.

APM approaches have replaced synthetic performance tests for large website such as Facebook and Bing. This is an exciting and disruptive technology that has already significantly changed how people think about and value performance testing.

6 Testing Incomplete Systems

Testing incomplete systems requires a strong knowledge of the system's architecture, understanding the difference between test coverage in functional and performance testing, and the ability to adapt tools to the job at hand. Most components have programming interfaces, which is a handy entry point for test code, particularly when it can be borrowed from developers and their test harnesses.

In this section, we'll consider a common architecture pattern – a Presentation Layer communicating via Web Service calls to a Business Logic Layer. These servers then communicate with the rest of the application, which includes a Message Bus, a Transactional Database and a Data Warehouse (with Metrics Capture Server), an Authentication Store, and an Application/Concurrent Processing Server. Diagrams and deeper analysis start on the next page.

6.1 Test Case Design

To design test cases, identifying key paths in the software as soon as they appear. Think about user activities like road systems – where are the highways? What will be meaningful to test throughout the project, and calibrate on?

One example is Login. Login/logoff and the related creation and destruction of session objects are expensive, and matter for all users. See Section 4.1 for more on this. Other examples include search and browse functions that are likely to appear early in the project, and are likely to represent key functionality for users.

6.2 Different Meanings of "Interface"

Seasoned automation developers have experienced the problem of brittle tests because of changing user interfaces. We can borrow from their experience and strategies to make tests that will last longer. We still need to assume that we will spend significant time maintaining and updating tests as software changes.

One easy place to start is at APIs (Application Programming Interfaces, or colloquially: Amateur Programming Interfaces). These are the same abstraction points developers typically use, meaning that in addition to be good points to measure at, they is likely to be documentation and test harness code already sitting around that we can adapt for performance tests.

When components are missing, techniques such as mocking, stubs, and auto-responders can be used to simulate components that are not in place yet. Here is another opportunity to talk with Developers about how they test these systems in progress, and repurpose existing code for test harnesses and automation.

6.3 Testing Pieces of a System

With these techniques, we can test pieces of a system. If the system is still being developed, or components are not yet assembled, we can still test what is there and assess how it performs and scales. Starting on the next page, we will discuss how to test parts and subsets of a system by examining the sample architecture.

Copies may not be made or distributed for commercial use

6.4 Example Application – Presentation Layer

First, let's consider abstracting the Presentation Layer, and addressing the backend servers (and supporting pieces). The Business Logic layer is frequently the scaling bottleneck in this architecture pattern. Web services are big buckets of functionality usually running a lot of interpreted code, and a lot of code written for the specific project.

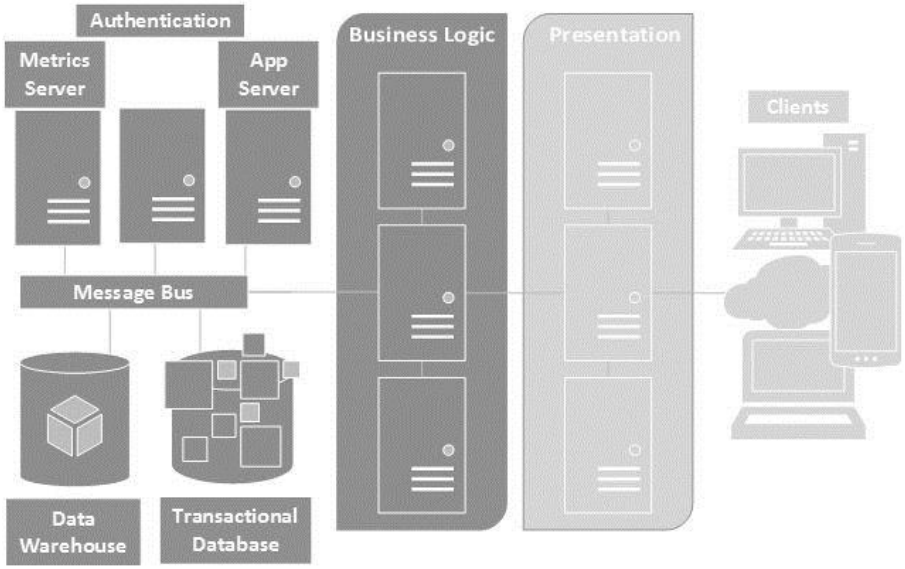

Figure 3: Example Architecture

We can test web services directly, making our tests more durable to changes to the user interface at the Presentation layer. We can abstract the presentation layer, even borrow code from it, and test the rest of the system back.

6.5 Example Component - Authentication

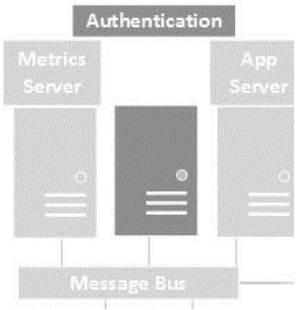

Figure 4: Authentication

At the opposite end of the spectrum from excluding one component is testing just one component.

For Authentication, we can first create a test that just creates sessions. Once we have that, we can answer some important questions about this essential component of the system, such as response time to create a session, the number of sessions that can be created, the arrival rate of sessions that can be supported, and the reliability of creating sessions.

6.6 Example Component: Message Bus

Many systems implement a Message Bus and publish-subscribe methods for communicating to other components. It is typically designed to make it easy to create and consume messages, so that components can easily be connected.

Figure 5: Message Bus

Architectural assumptions are made at this stage that must be correct for the project to be successful. As a key component and potential scalability challenge, this is an opportunity to reduce risk and add value to the project very early on.

First, find or make a test harness that allows creation (push) and consumption (pull) of messages. A good place to start is to ask how developers test. Once this harness is in hand, you can start to track metrics and answer important questions.

What is the response time to push a message? What is the response time to pull a message?

At what rate can messages be created? With one publisher? With multiple publishers?

At what rate can messages be consumed? What if message servers/dispatchers are added?

7 Communicating Performance During Iterative Development

Performance Testing generates a lot of data, and too frequently, stops there. Turning over large chunks of data generally isn't helpful to stakeholders, yet it continues to be a frequent output format from performance testing, jeopardizing value by making too difficult to understand what happened.

To get value from a test, we don't need data, we need Information. Analysis of data leads to Information – knowledge that we can trust and act on, and that our stakeholders are able to trust and act on.

Your value as a tester is the information you provide, and the action(s) it inspires. After information is generated, it needs to be shared. Make results visible, and recruit consumers of your information. Become an important project status indicator. Help people who can do something about it understand and care about the information you provide.

Here are some strategies for sharing information. Consider using all of them to share what is learned, and to continually demonstrate the value you are adding to the project.

7.1 Informal Communication

Meetings with peers and supervisors in attendance can be a tough place to receive feedback gracefully. It's also a terrible place to be wrong if you are providing that feedback, as it will impact your credibility and make it harder for you to be heard.

If you are confident in your information and the venue is appropriate, you may be willing to be declarative, certain it will be received well and you are prepared to handle any follow-up questions or challenges. If you are not sure what you see, or how it will be consumed, you might instead choose a more cautions path. Approach a key engineer and ask, "Can I show you something?" Vet findings before you broadcast.

7.2 Information Pushes

Once you have actionable information, push it out to the team. Emails about results are frequently used, but is critical to remember to start with a short summary that can be scanned on a mobile device. Information sources that are unclear or of low value are likely to be ignored.

Other venues for pushing out information are status reports, stand-up meetings, and gossip/grapevines. When you work in performance, you will probably be asked “How is performance these days?”

7.3 Reporting Frequency and Feedback Loops

A frequently encountered challenge with Simulation Tests is that testing doesn’t take place until software is completed. Waiting until software is complete means that testing occurs at the very end of the project. This exposes testing to time pressure, particularly when earlier phases of the project run past milestones.

Finding problems at a point in the schedule where the testing is the last gate to going live is of limited value. Either the problems are minor enough to be lived with and are not fixed, or serious enough to delay the entire project. Even positively identifying the source of the problems may be too time consuming to delay go-live. This decision is rarely made in a vacuum; stakeholders or team members who feel prestige or jobs are at stake may not think like engineers.

With iterative-speed performance testing, we should try to report as often as we can. Even if it we can’t retest every single sprint, we reduce the cost of rework by reporting performance and reliability issues as soon as possible.

7.4 Information Radiators

Intranet pages and whiteboards are two ways to make information available permanently, in a way that supports self-service consumption. Consider ways to present information that make it easy for consumers to understand progress over time, whether are deeply involved with the project or not.

Date	Build	Min	Mean	Max	90%
2/1	9.3.0.29	14.1	14.37	14.7	14.6
3/1	***9.3.0.47***	***37.03***	***38.46***	***39.56***	***39.18***
8/2	9.5.0.34	16.02	16.61	17	16.83
9/9	10.0.0.15	17.02	17.81	18.08	18.02
10/12	10.1.0.3	16.86	17.59	18.03	18
11/30	10.1.0.38	18.05	18.57	18.89	18.81
1/4	10.1.0.67	18.87	19.28	19.48	19.46
2/2	10.1.0.82	18.35	18.96	19.31	19.24

Calibration Results for Web Login, Search, View. Burst 10 threads iterating 10 times

8 Conclusions/Takeaways

I believe that these are the most important takeaways from this paper:

1. Traditional Performance Testing was originally conceived as a gate to production for completed software, depending on a perception of “realism” to prove value to stakeholders.
2. Iterative Development Processes caused functional testing to take place earlier and more often. Non-functional testing (performance, reliability, etc) should follow.
3. Performance Testing addresses specific risks. These risks can be tested against with “unrealistic” load tests just as well (or better) than with “realistic” tests. Spend time thinking and talking about “realistic” if you encounter resistance.
4. Simple test cases and scripts help make tests cheaper and faster to run, allowing measurements to take place much more often. Once you have a reliable measurement, you can calibrate with these measurement between builds, servers, components, and environments, and revisit previous results to ensure that all of the performance variables are understood.
5. There are other performance test techniques besides load testing. Consider using them to provide immediate feedback.

6. Incomplete Systems can be performance tested by analyzing the components of the system, testing individual and partial combinations of these components, and testing what is there, as it arrives.
7. Communicating performance feedback, actively and passively, is essential for performance testing effectively. Information that does not have any impact doesn't add value.

9 References

Many of these concepts were discussed at WOPR22 (The 22nd Workshop on Performance and Reliability, performance-workshop.org). WOPR22 was held May 21-23, 2014, in Malmö, Sweden, on the topic of "Early Performance Testing". Participants in the workshop included Fredrik Fristedt, Andy Hohenner, Paul Holland, Martin Hynie, Emil Johansson, Maria Kedemo, John Meza, Eric Proegler, Bob Sklar, Paul Stapleton, Andy Still, Neil Taitt, and Mais Tawfik Ashkar.

The rules of attribution for LAWST-style and LAWST–inspired workshops include a requirement that any publications from the meeting will list all attendees as contributors.