CHAPTER 3

PERFORMANCE ENGINEERING

JAVA ENTERPRISE PERFORMANCE

ALOIS REITBAUER
KLAUS ENZENHOFER
ANDREAS GRABNER
MICHAEL KOPP
STEPHEN PIERZCHALA
STEVE WILSON

Compuware

THE TECHNOLOGY PERFORMANCE COMPANY

Java Enterprise Performance

Chapter 3

Performance Engineering: Approaching Performance Engineering Afresh

By

Stephen Pierzchala & Andreas Grabner

Chapter 3

Performance Engineering: Approaching Performance Engineering Afresh

What's in this chapter?

Most of us are used to waiting until the very end of the software-development process to evaluate the performance of new applications. Many reasons are given for doing things this way, including the following:

- Many feel that it does not make sense to determine the performance metrics of a system until the entire system is available and capable of running the new software.
- It is assumed that performance- and scalability-related issues can be identified in only a fully configured load-testing environment.
- It can be difficult to scale and apply the performance metrics of one environment to another, so we wait until our large, production-related server environments are ready before we begin performance analysis.

These all make sense, but you can also see that there's a problem with this approach. By waiting, we risk delays in the entire process, and missed delivery goals.

Figure 3.1 shows a typical burn-down chart used in agile development. During development, the team works on user stories that could be fit into the iterations planned for the next release. Too often we see performance-related testing pushed to the end of the last development sprint. This approach is great if you need to push a lot of new features into a software product, but it jeopardizes the planned release date and can wreak havoc with quality standards. Waiting until the end to focus on performance usually unveils problems that are too big to fix in the time planned for the testing phase. This either leads to missed goals as feature or quality cuts have to be accepted, or it leads to missed deadlines.

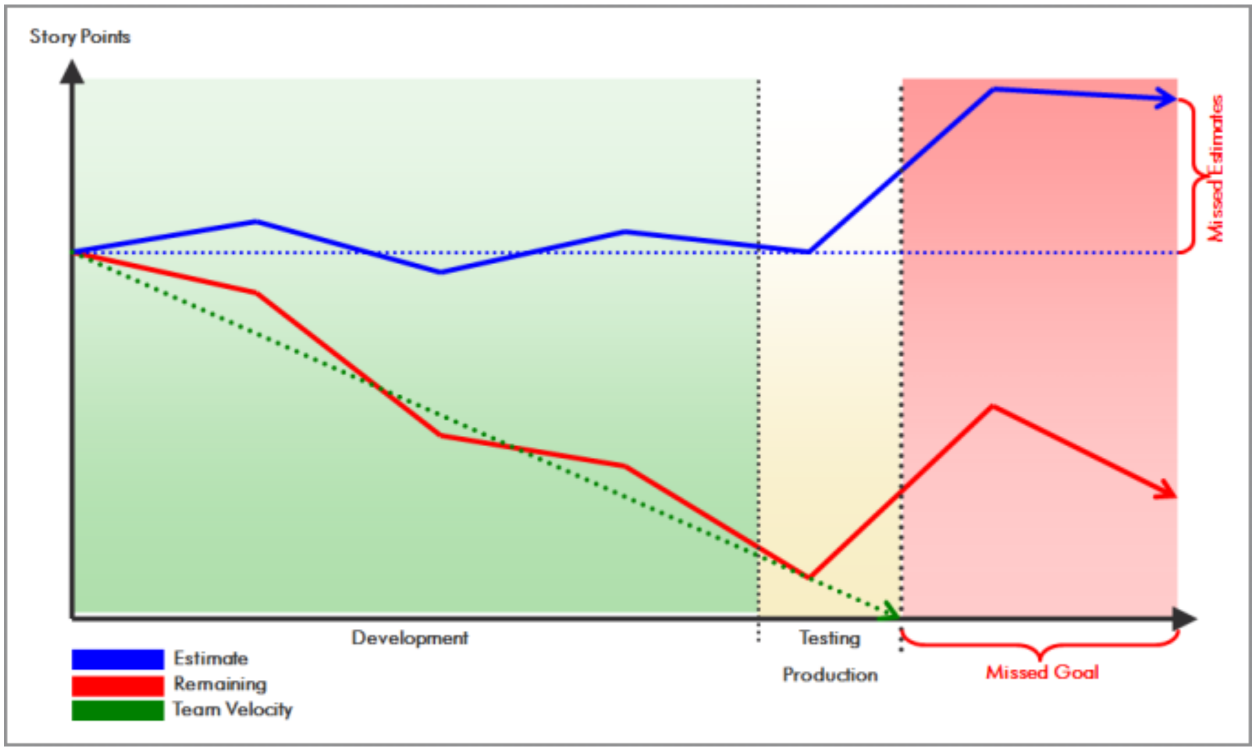


Figure 3.1: Pushing performance testing to the end of the release cycle often uncovers problems that can jeopardize the original project goals and deadlines.

The good news is that there are many options for testing and ensuring the performance and scalability of an application during the development phase. By focusing on performance early, many performance and scalability problems can be identified and eliminated before they ever make it into the final test phase (which is still very important).

In this chapter we will look at important performance-engineering methods that can be used during development. This includes dynamic architecture validation in the development environment, small-scale performance tests in continuous integration, and enforcement of development best practices to avoid common performance problems. We will also look at real-life examples showing how performance engineering in development and continuous integration have helped software companies maintain their agility while implementing new stories and improving overall quality.

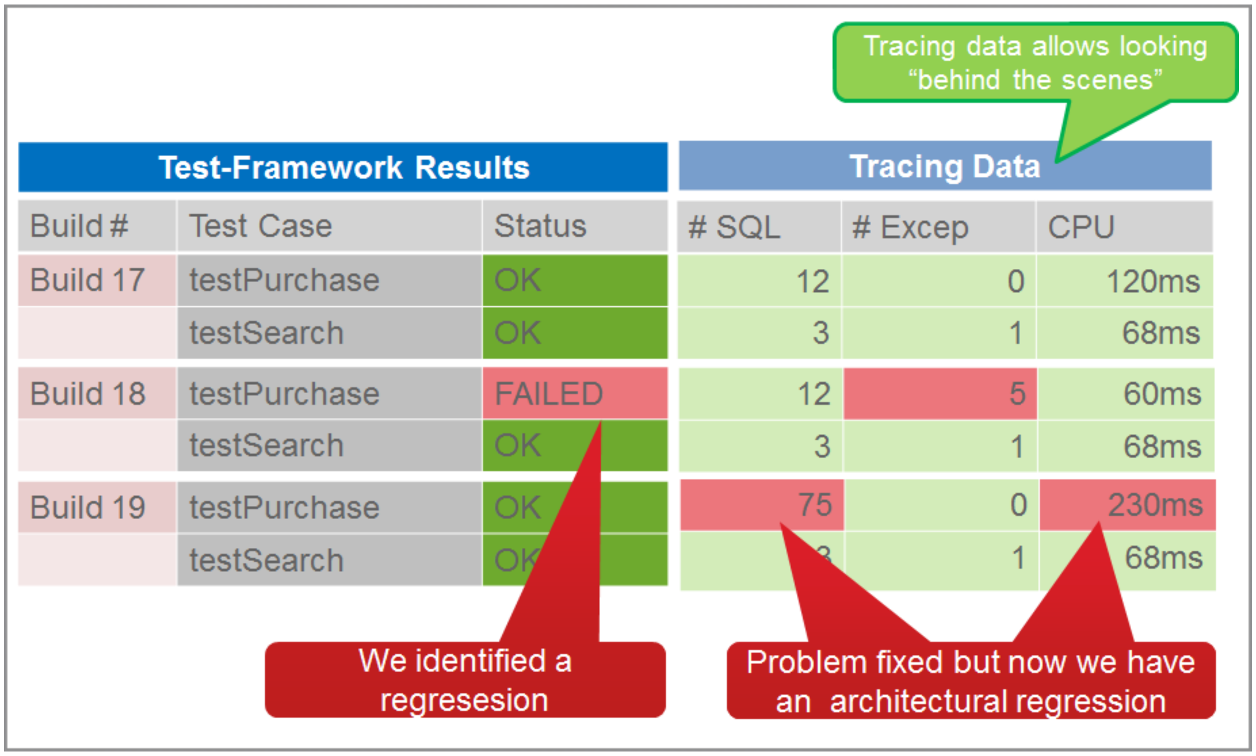
In an upcoming section of this chapter, *Load Testing—Essential and Not Difficult*, we will discuss traditional performance testing and large-scale load testing, which remains important but can be streamlined using the techniques we discuss in the first part.

The performance methodologies we'll discuss are based on agile principles and are best *Load Testing—Essential and Not Difficult* integrated into the agile software process of continuous improvement. So we'll begin with a brief discussion of how this works and why it's a useful and important tool for software and performance engineers.

Adopting Agile Principles for Performance Evaluation

Recent advances in agile software development and agile best practices, such as continuous integration and test-driven development, make it easier to determine performance metrics at the code and component level during development. For instance, we can use a unit test to examine the metrics for any given component continually, checking execution time, number of database queries executed, or CPU usage every time there's a programmatic change. This way any problems introduced during development can be diagnosed and fixed right away, allowing us to build better software—with fewer issues making it into the load-testing phase.

Figure 3.2 shows the benefits of a unit-test framework that allows us to verify the functionality of code. For most companies, it is any easy step to add tracing data to test metrics already used for performance analysis, scalability, and component behavior. Regressions or problems with specific implementations can be identified with the next test run.



Test-Framework Results			Tracing Data		
Build #	Test Case	Status	# SQL	# Excep	CPU
Build 17	testPurchase	OK	12	0	120ms
	testSearch	OK	3	1	68ms
Build 18	testPurchase	FAILED	12	5	60ms
	testSearch	OK	3	1	68ms
Build 19	testPurchase	OK	75	0	230ms
	testSearch	OK	3	1	68ms

Tracing data allows looking "behind the scenes"
We identified a regresesion
Problem fixed but now we have an architectural regression

Figure 3.2: Tracing performance, scalability, and architectural metrics along with unit-test executions allows easy identification of regressions and general implementation problems.

Transitioning from Traditional to Continuous Performance Engineering

Continuously verifying architectural metrics and fixing problems as they happen in development leads to shorter throughput times in load and performance tests because we are already testing a higher-quality software product. In addition, automation in test and build management makes it possible to automate a large number of testing tasks, which can then be regularly implemented as development progresses.

Figure 3.3 compares the traditional approach to performance management with the continuous approach. Instead of one long testing phase with an additional long fixing phase at the end of the development cycle, we've implemented continuous performance testing during development, thus achieving a shorter testing phase (with a resultant long fixing phase before we can release the product). Integrating performance engineering into development slightly prolongs development, as it is an additional task. But we save so much on testing and are able to achieve such a consistently high-quality product that the gain far outweighs the price.

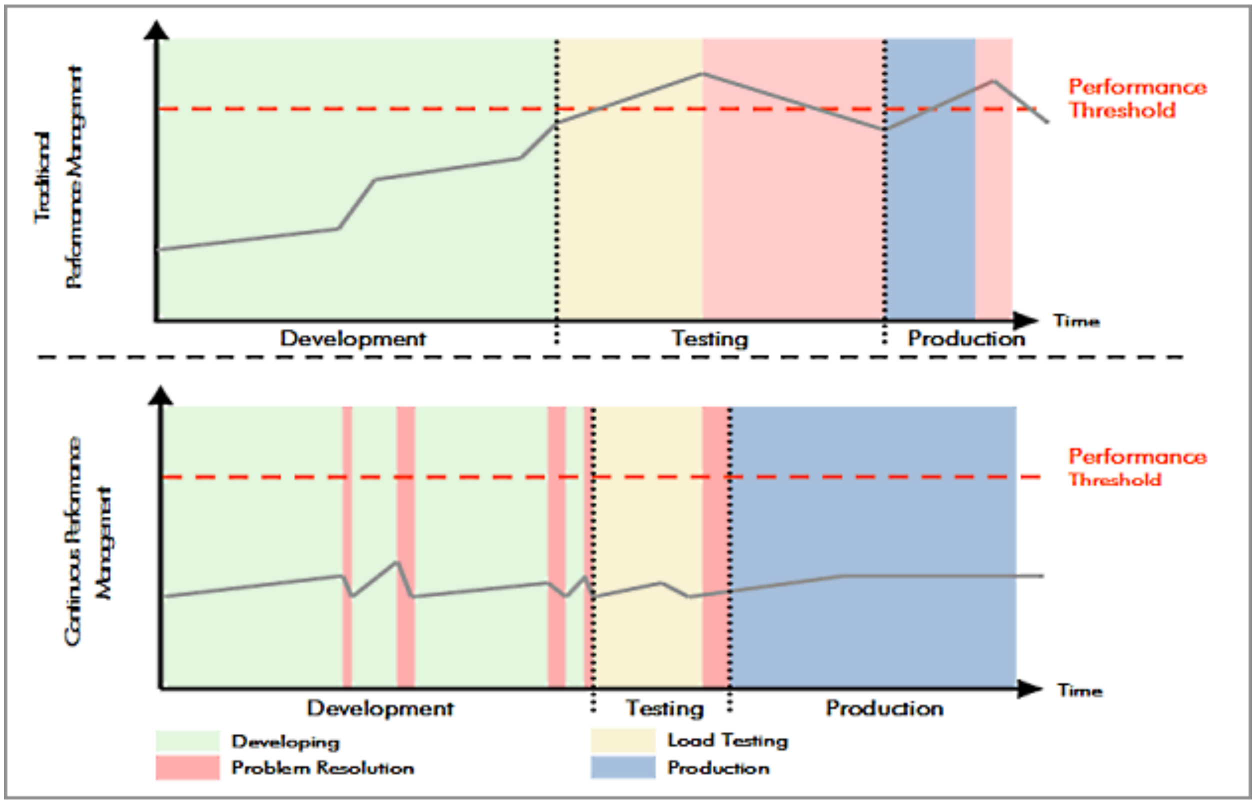


Figure 3.3: Adding Continuous Performance Testing into Development shortens the actual Test Phase and eliminates long Problem Resolution tasks before production release.

Transitioning from traditional to continuous performance engineering requires some effort, not unlike adopting agile development methodologies. I've worked with companies that made both transitions, which we can refer to as *the full transition*. The key benefit we saw was not simply a shorter *end of the release* testing phase. In fact, intense performance and load testing are still necessary. The main advantage we experienced was in consistently higher software quality and a nearly trivial fixing phase after the final performance tests, which allowed the engineering teams to plan with greater confidence.

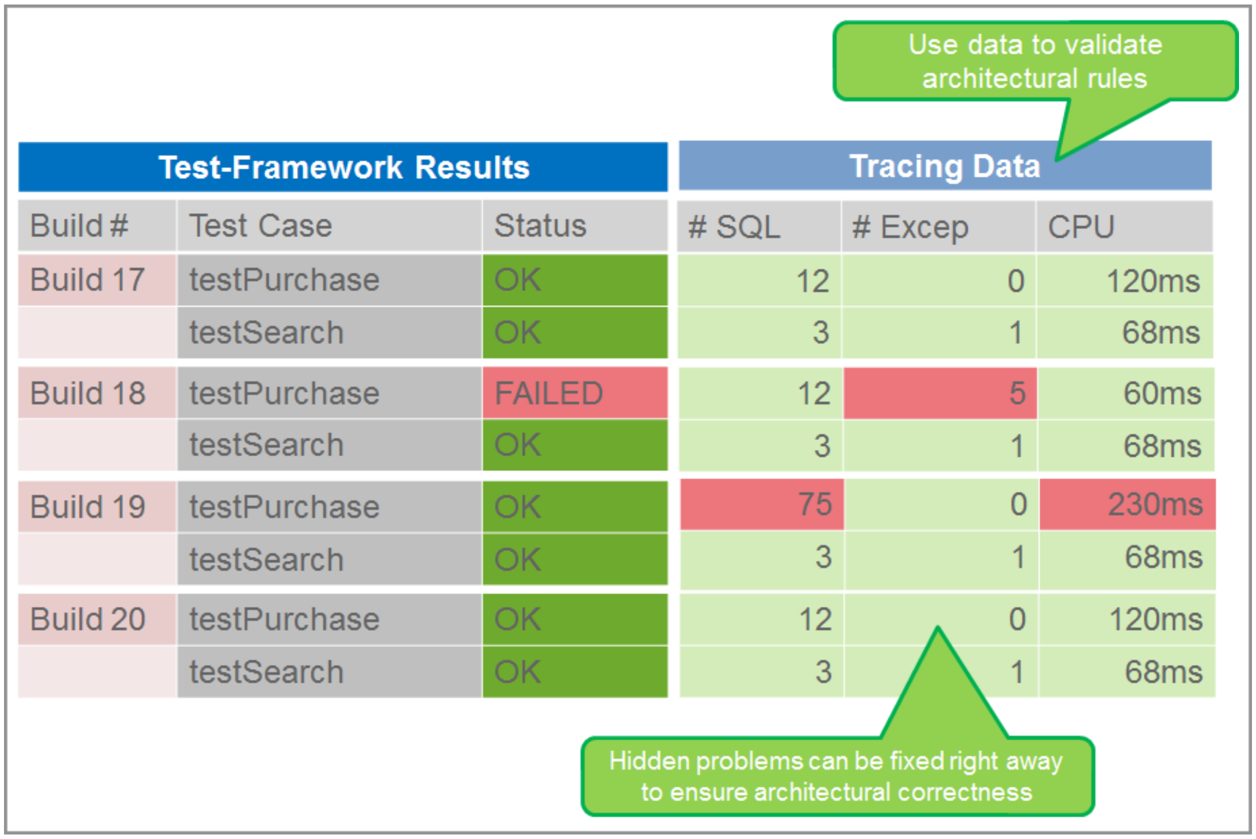
Now let's have a closer look at the three most important methods of modern performance engineering during development: dynamic architecture validation in the development environment, small-scale performance tests in continuous integration, as well as enforcement of development best practices to avoid common performance problems. We can't say it enough times: both large-scale load testing and traditional performance testing remain essential elements of performance engineering, and are thoroughly covered in the upcoming *Load Testing—Essential and Not Difficult* section.

Employing Dynamic Architecture Validation

A relatively new methodology in the area of performance engineering is dynamic architecture validation. In contrast to static validation, it checks the runtime behavior of the application for potential problem patterns. By identifying common architectural implementation errors, many potential performance issues can be caught and fixed early in the development process, including excessive database retrievals or making too many remote service calls, which can introduce architectural problems like latency bottlenecks.

Dynamic architecture validation allows testing an application before it has been fully implemented, as we can analyze the dynamic behavior of individual components that can be unit-tested. It also makes it easy to identify architectural regressions that have been introduced with code changes by comparing the dynamic behavior between test runs.

Figure 3.4 shows what we can achieve by validating rules such as "You should not use more than 10 SQL statements." Even though a functional regression was identified with our testing framework in Build 18 and fixed in Build 19, an architectural regression was introduced that caused many more SQL statements than before to be executed. These hidden problems can be identified by looking at metrics per test case, leading to functionally and architecturally correct code. This raises code quality, ensures architectural correctness, and ultimately results in a better-performing and -scaling application.


Use data to validate architectural rules
Hidden problems can be fixed right away to ensure architectural correctness

Test-Framework Results			Tracing Data		
Build #	Test Case	Status	# SQL	# Excep	CPU
Build 17	testPurchase	OK	12	0	120ms
	testSearch	OK	3	1	68ms
Build 18	testPurchase	FAILED	12	5	60ms
	testSearch	OK	3	1	68ms
Build 19	testPurchase	OK	75	0	230ms
	testSearch	OK	3	1	68ms
Build 20	testPurchase	OK	12	0	120ms
	testSearch	OK	3	1	68ms

Figure 3.4: Looking at architectural metrics per test case allows us to identify and fix regressions as they are introduced.

We've mentioned the number of database statements as one of the common problem areas or problem patterns that can be identified through dynamic architectural validation. Let's now expand on this and discuss how to identify these and other common patterns.

Identifying Common Problem Patterns

The goal of dynamic architectural validation is to recognize potential trouble spots in the application runtime behavior that can lead to performance and scalability problems. For instance, executing 20 SQL statements to perform a login that could be achieved in a single SQL statement is an obvious problem. It might not be a problem on the developer's workstation, as he is the only one accessing the system. But under heavy load with thousands of users, a login requiring 20 times as many SQL statements will definitely stress the database unnecessarily.

The possible problems can be divided into two categories:

- Problems in runtime behavior are recognized due to characteristics in the process logic. An example is the N+1 query problem in Hibernate. One must recognize in this case that identical queries are carried out multiple times with different parameters. A web application causing a high number of remote calls or faulty caching would be another example.

- In contrast, other architecture problems require additional information for analysis. It takes a responsible architect to determine the necessary parameters and overall test conditions. Possible examples include transferring only limited data volume or avoiding synchronous remote calls (recognizable via the called method).

The responsible software architect(s) must define a solid set of architectural rules that fall into the two categories, such as these:

1. Setting a limit on the total number of database queries
2. Disallowing the same database query to be executed multiple times for the same operation

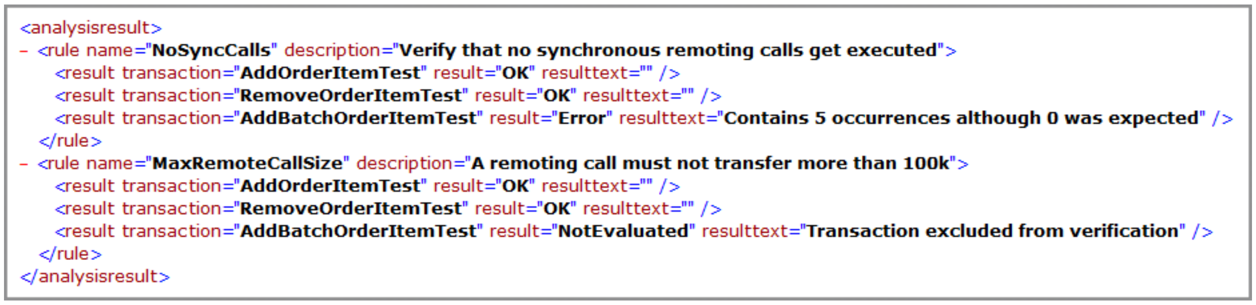
Practical methods for checking these rules depend to a large degree on the tools used, as well as on the degree of automation supported by these tools. Tracing or diagnostic tools that show the execution path in chronological order tend to be better for enforcing rules as complete information, such as number of methods executed, is captured.

Logging mechanisms that can log very detailed information about the execution path can also be used to validate certain rules. Here it depends on the granularity of log information available. When focusing on database activity, a logging framework that can generate log output for each database-query execution is often sufficient. This may require special configuration or custom extension of the logging framework to produce the required output.

You've learned that the rules for dynamic architecture validations are defined by the software architect. It is also the architect who makes sure that these rules are well understood by all developers in the form of mentoring or training, as well as during code reviews. This ensures that code is written with rules that lead to higher-quality code in mind from the start. The next step is to automate the validation of these rules instead of manually looking at measures and validating them against the rules. Let's discuss how that works.

Automating Performance Validation

Performance and architectural-rule validation can be automated and integrated into the build process using one of several tools available for the task. There are both open-source and commercial solutions, and capabilities vary over a range of ease of use, degree of automation, and level of rule-definition flexibility. For the purpose of showing how this works, we will look at dynaTrace and its capabilities. *Figure 3.5* shows a results file produced by dynaTrace. A set of architectural rules was validated against trace and diagnostics data gathered for a set of executed unit tests. The listing indicates which tests passed and failed according to the rules specified by the architect.

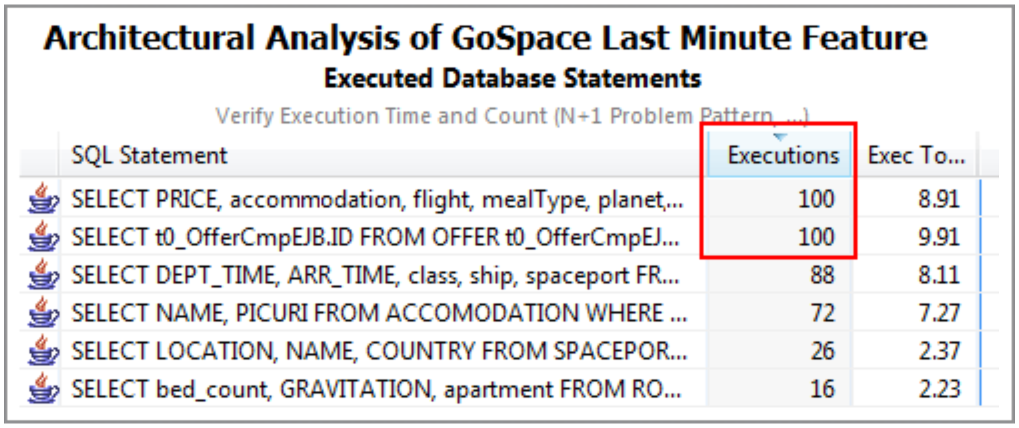


```
<analysisresult>
- <rule name="NoSyncCalls" description="Verify that no synchronous remoting calls get executed">
    <result transaction="AddOrderItemTest" result="OK" resulttext="" />
    <result transaction="RemoveOrderItemTest" result="OK" resulttext="" />
    <result transaction="AddBatchOrderItemTest" result="Error" resulttext="Contains 5 occurrences although 0 was expected" />
  </rule>
- <rule name="MaxRemoteCallSize" description="A remoting call must not transfer more than 100k">
    <result transaction="AddOrderItemTest" result="OK" resulttext="" />
    <result transaction="RemoveOrderItemTest" result="OK" resulttext="" />
    <result transaction="AddBatchOrderItemTest" result="NotEvaluated" resulttext="Transaction excluded from verification" />
  </rule>
</analysisresult>
```

Figure 3.5: During validation, predefined rules are automatically verified against test transactions.

Even if you choose to analyze runtime behavior using a debugger or by examining the corresponding log output, you'll have a better understanding of the application's behavior. The more complex the application, the more important it is to run these analytical tests.

Figure 3.6 shows another bit of sample output from a dynaTrace analysis of database calls for a test transaction. The report makes it obvious that some of the queries have been executed hundreds of times for a single operation. This clearly violates one of our architectural rules.



Architectural Analysis of GoSpace Last Minute Feature

Executed Database Statements

Verify Execution Time and Count (N+1 Problem Pattern, ...)

SQL Statement	Executions	Exec To...
SELECT PRICE, accommodation, flight, mealType, planet,...	100	8.91
SELECT t0_OfferCmpEJB.ID FROM OFFER t0_OfferCmpEJ...	100	9.91
SELECT DEPT_TIME, ARR_TIME, class, ship, spaceport FR...	88	8.11
SELECT NAME, PICURI FROM ACCOMODATION WHERE ...	72	7.27
SELECT LOCATION, NAME, COUNTRY FROM SPACEPOR...	26	2.37
SELECT bed_count, GRAVITATION, apartment FROM RO...	16	2.23

Figure 3.6: Several hundred SQL queries for one feature are too many; this violates the rules of the architecture.

It's difficult for application developers to understand the detailed behavior of all the components they are using, which means that inevitably, undesirable side effects will turn up during load testing. The cycle of test, fix, and retest can be quite time-consuming if you wait until the end of the project lifecycle. In my own experience, a large number of the problems revealed during load testing or in production could have been detected and avoided during development.

It's also important to realize that the benefits of dynamic architecture validation are not limited to the development phase. In particular, cache behavior should also be tested using this method, both during load tests and in production.

Performance-Testing in Continuous Integration

With agile development methods, we are able to test using stable systems either on a continuous basis or at regular intervals. This, in turn, has enabled a much larger degree of test automation and allows automating large parts of the test process, especially in the area of functional testing, where it has gained widespread adoption. We're now observing a similar trend in performance testing.

Automated performance testing is triggered from within the build process. To avoid interactions with other systems and ensure stable results, these tests should be executed on their own hardware. The results, as with functional tests, can be formatted as JUnit or HTML reports. Although test execution time is relevant, it is of secondary importance. Instead, additional log or tracing data can be used in case of execution errors. Most importantly, select those use cases that are most critical for the application's performance, such as searching in a product catalog or finalizing an order. It is necessary to identify and control trends and changes, and not useful to test the entire application under load.

Types of Tests

Use test frameworks, such as JUnit, to develop tests that are easily integrated into existing continuous-integration environments. For some test cases, it's appropriate to reuse functional tests or additional content validations. By applying the architecture validation discussed above to existing JUnit tests, it is no great effort to obtain initial performance and scaling analyses.

In our own work, we had a case in which JUnit tests verified a product search with a results list of 50 entries. Functionally, this test always returned a positive result. Adding architecture validation by means of a tracing tool showed an incorrectly configured persistence framework, which loaded several thousand product objects from the database, and from these returned the requested first 50. Under load, this problem would have inevitably led to performance and scaling problems. Reusing this test enabled the discovery of the problem during development, with very little effort.

Besides using JUnit for unit testing, many application developers use frameworks, such as Selenium, which are used to test web-based applications by driving a real browser instead of testing individual code components. Selenium and similar test frameworks make it easy to use these tests for single-run functional tests and can be reused for performance and load tests. *Listing 3.1* shows a Selenium test script that can be used both for functional tests as well reused for load and performance tests.

```java
public class Example {
public static void main(String[] args) throws Exception {
    // WebDriver driver = new HtmlUnitDriver();
    // WebDriver driver = new InternetExplorerDriver();
    WebDriver driver = new FirefoxDriver();
    // Open our Home Page
    ProductCatalogHomePage home = new ProductCatalogHomePage(driver);
    // Search for a product
    SearchResultPage result = home.searchForProduct("DVD Player");
    if(result.countResults() <= 0)
    throw new Exception("No Results returned");
    // Click on a Product
    ProductPage productPage = result.clickOnResult(1);
    // interact on the product page
    productPage.addReview("Should be a good product");
productPage.addToCart();
    driver.close();
}
}
```

Listing 3.1: Web test frameworks, such as Selenium, can be used for functional tests, plus load and performance tests.

When conducting performance tests in a continuous-integration environment, we like to design test cases with a minimum run or execution time. Ideally, it should be several seconds. When execution time is too short, the test accuracy is susceptible to small fluctuations. For example, with a test case of 200 ms, a fluctuation of 20 ms amounts to 10% of execution time, whereas it is only about 2% for a test case of one second.

Note: There are still situations when classic load tests are preferred, especially when testing scalability or concurrency characteristics. Most load-testing tools can either be remote-controlled via corresponding interfaces or started via the command line. We prefer to use JUnit as the controlling and executing framework because the testing tools are easily integrated using available extension mechanisms.

Dynamic architecture validation lets us identify potential changes in performance and in the internal processing of application cases. This includes the analysis of response times, changes in the execution of database statements, and an examination of remoting calls and object allocations. This is not a substitute for load testing, as it is not possible to extrapolate results from the continuous-integration environment to later production-stage implementation. The point is to streamline load testing by identifying possible performance issues earlier in development, thereby making the overall testing process more agile.

Adding Performance Tests to the Build Process

Adding performance tests into your continuous-integration process is one important step to continuous performance engineering. Most simply, these performance tests can use (or reuse) existing unit and functional tests, and can then be executed within familiar test frameworks, such as JUnit. *Figure 3.7* diagrams a typical continuous-integration process. It begins at the point when a developer checks in code, and continues through the build and test processes. It finishes with test and performance results before ending up back with the developer.

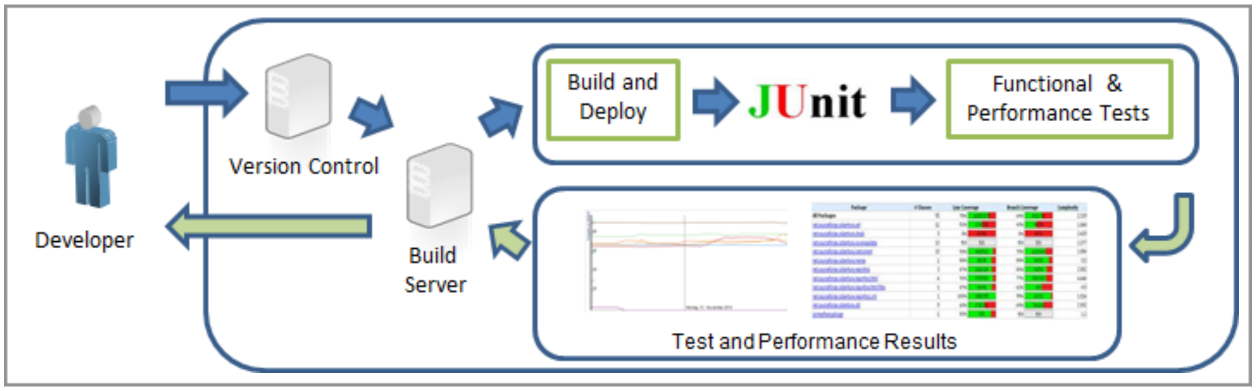


Figure 3.7: Adding performance tests to the mix of existing unit and functional tests in a typical continuous-integration process gives developers additional feedback on their code quality with respect to performance and scalability.

In addition to the functional results, performance-test results are analyzed and recorded for each build, and it is this integration of continuous testing with the agile ideal of continuous improvement that makes this method so useful. Developers are able to collect feedback on changed components in a sort of continuous feedback loop, which enables them to react early to any detected performance or scalability problems.

Conducting Measurements

Just as the hardware environment affects application performance, it affects test measurements. In order to facilitate comparisons between your performance tests, we recommend using the same configuration for all of your tests. And unless you're testing such external factors explicitly, it's important to eliminate timing variables, such as those caused by hard-disk accesses or network latency times in distributed tests. To achieve this goal, one must remove any volatile measurements and focus solely on measures not impacted by environmental factors.

The choice of tracing and diagnostic tools used to measure performance and determine architectural metrics can substantially affect the difficulty of this task. For example, to subtract the impact of garbage-collection runs, some tools can measure the total code-execution time. In the special case of eliminating the runtime impact of garbage collection from the JVM or Common Language Runtime, some tools automatically calculate a *clean* measure.

An alternative best-practice approach is to focus on measures that are not impacted by the runtime environment. So instead of looking at network latency or I/O, it is sometimes better to look at the number of hard-disk accesses, number of remoting calls, or data-transfer volume. These metrics will be stable regardless of the underlying environment.

Whether you focus on measures that can be impacted by the executing environment or not, it is important to verify your test environment to assure sufficiently stable results. We recommend writing a reference test case and then monitoring its runtime. This test case performs a simple calculation—for example, a high Fibonacci 3 number. Measuring the execution time and CPU usage for this test case should produce consistent results across multiple test executions. Fluctuating response times at the start are not unusual. Once the system has "warmed up," the execution times will gradually stabilize. Now the actual test can begin. *Figure 3.8* shows how one can use reference-test execution times to determine the point at which the system is stable.

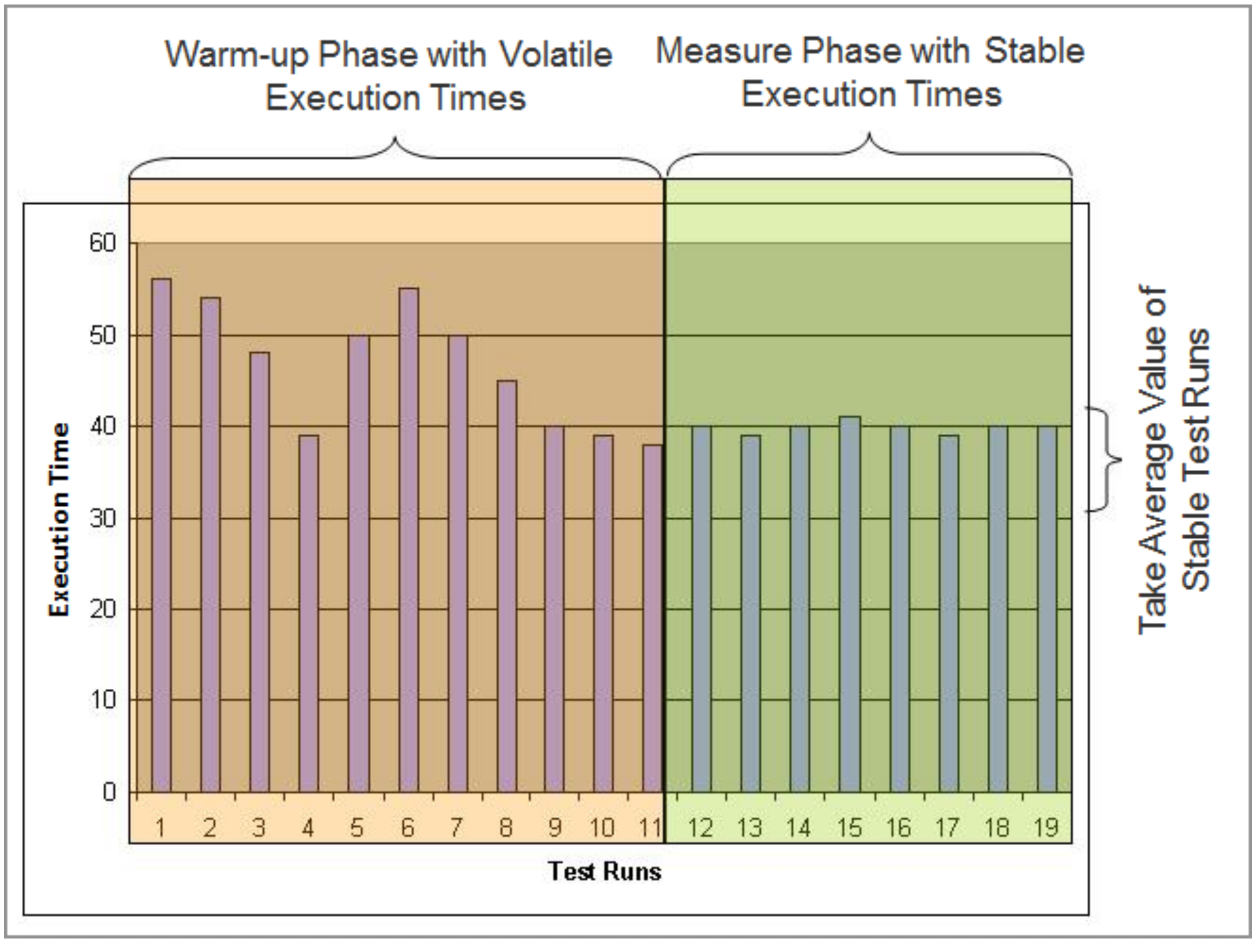
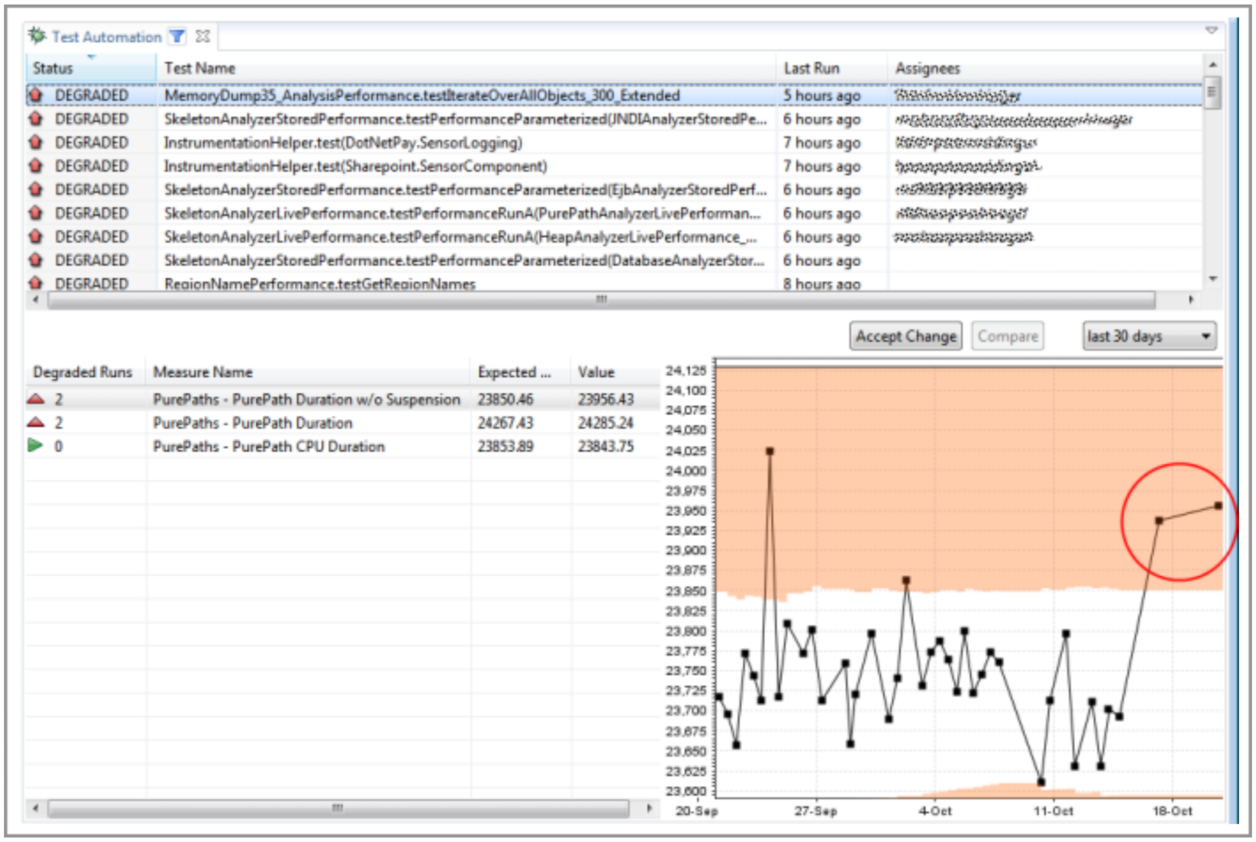


Figure 3.8: Using a reference test to assure system stability and consistent performance-test results

Analyzing Measurements

Some key metrics for analysis include CPU usage, memory allocation, network utilization, the number and frequency of database queries and remoting calls, and test execution time. By repeating the tests and comparing results, we're able to identify unexpected deviations fairly quickly. (Obviously, if the test system isn't stable, our data will be worthless.)

In *Figure 3.9*, you can see timing results across multiple test runs. A regression analysis of the data shows on obvious performance degradation and we are then able to follow up with a detailed analysis of each component covered by the test. Also, we can identify potential scaling problems by analyzing the performance of individual components under increasing load.



Status	Test Name	Last Run	Assignees
DEGRADED	MemoryDump35_AnalysisPerformance.testIterateOverAllObjects_300_Extended	5 hours ago	[redacted]
DEGRADED	SkeletonAnalyzerStoredPerformance.testPerformanceParameterized(JNDIAnalyzerStoredPe...	6 hours ago	[redacted]
DEGRADED	InstrumentationHelper.test(DotNetPay.SensorLogging)	7 hours ago	[redacted]
DEGRADED	InstrumentationHelper.test(Sharepoint.SensorComponent)	7 hours ago	[redacted]
DEGRADED	SkeletonAnalyzerStoredPerformance.testPerformanceParameterized(EjbAnalyzerStoredPerf...	6 hours ago	[redacted]
DEGRADED	SkeletonAnalyzerLivePerformance.testPerformanceRunA(PurePathAnalyzerLivePerforman...	6 hours ago	[redacted]
DEGRADED	SkeletonAnalyzerLivePerformance.testPerformanceRunA(HeapAnalyzerLivePerformance_...	6 hours ago	[redacted]
DEGRADED	SkeletonAnalyzerStoredPerformance.testPerformanceParameterized(DatabaseAnalyzerStor...	6 hours ago	
DEGRADED	RegionNamePerformance.testGetRegionNames	8 hours ago	

Degraded Runs	Measure Name	Expected ...	Value
2	PurePaths - PurePath Duration w/o Suspension	23850.46	23956.43
2	PurePaths - PurePath Duration	24267.43	24285.24
0	PurePaths - PurePath CPU Duration	23853.89	23843.75

Figure 3.9: Regression analysis based on the execution time of a performance test

Using this more-agile approach of testing during development, we're able to run regression analyses for each new build while the code changes are still fresh in the developer's mind. The cost of fixing such errors days, weeks, or months later can be considerably higher, almost incalculably so if the project ship date is affected.

Figure 3.10 graphs the performance behavior of individual components under increased load. The results clearly show that while three of the four components scale well, the business-logic component scales poorly, indicating a potentially serious problem under production.

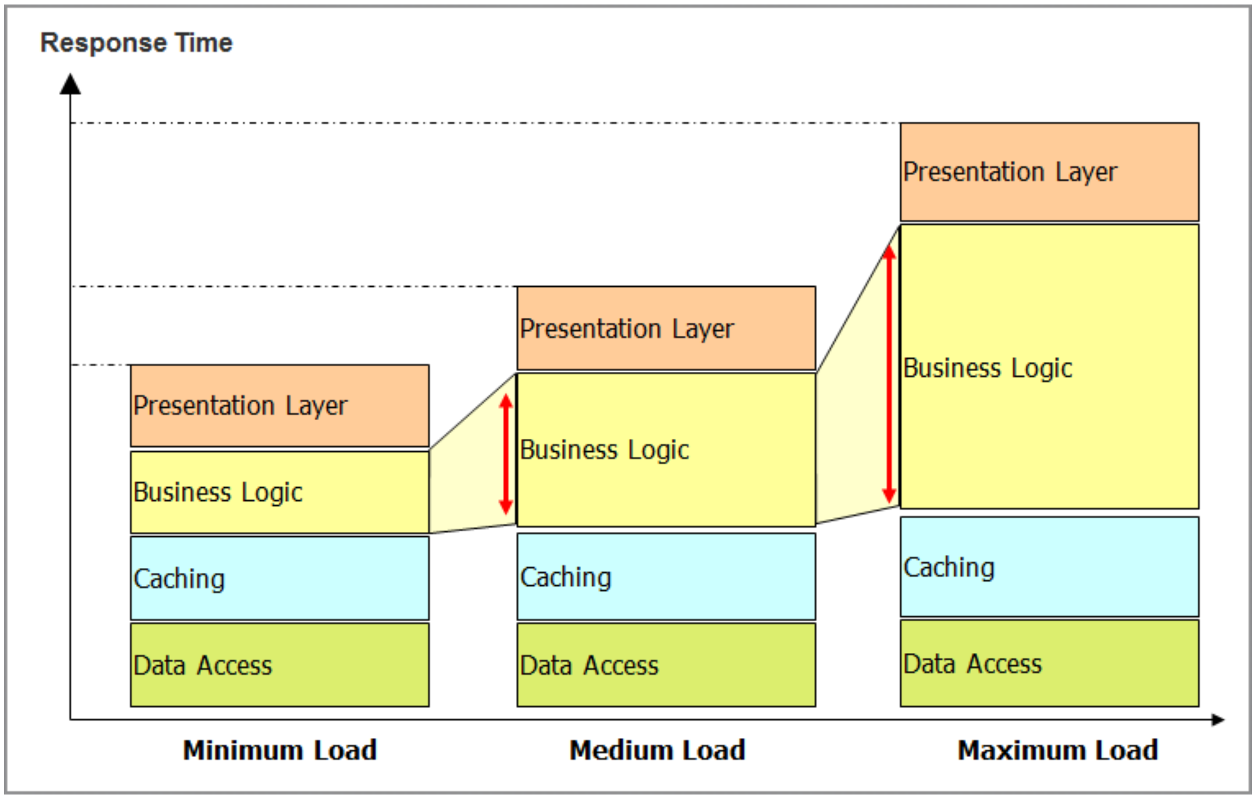


Figure 3.10: Testing under increasing load leads to additional conclusions regarding scalability.

Performing these tests for each individual build is often not possible. Providing a separate load-test environment can be time-consuming or even impossible, so it's not usually possible to perform these tests for each build. However, if you're able to conduct more-extensive load and performance tests—for example, over the weekend—it can make a big difference in the long run. You can provide better insights into the performance of the overall system, as well as for individual components, and it needn't have a negative impact on your developers, who might be waiting for the next build.

Regression Analysis

Every software change is a potential performance problem, which is exactly why you want to use regression analysis. By comparing metrics before and after a change is implemented, the impact on application performance becomes immediately apparent.

However, not all regression analyses are created equal, and there is a number of variations to take into account. For instance, performance characteristics can be measured by general response time, *black-box tests*, or by individual application components, *white-box tests*. Changes in overall system behavior are often apparent in either case, but there are also times when improved performance in one area might mask deterioration in another area

In *Figure 3.11*, you can see how the introduction of a software cache layer relieves pressure on the database layer. Storing frequently called objects in memory is more efficient than retrieving the object every time through a database request. This looks like a good architectural change, as we have more resources available on the database to handle other data requests. In the next step it's decided to increase the cache size. A larger cache requires additional memory but allows for caching even more objects and further decreases the number of database accesses. However, this change has a negative impact on the cache layer's performance, as having more objects in the cache means more memory usage, which introduces higher garbage-collection times.

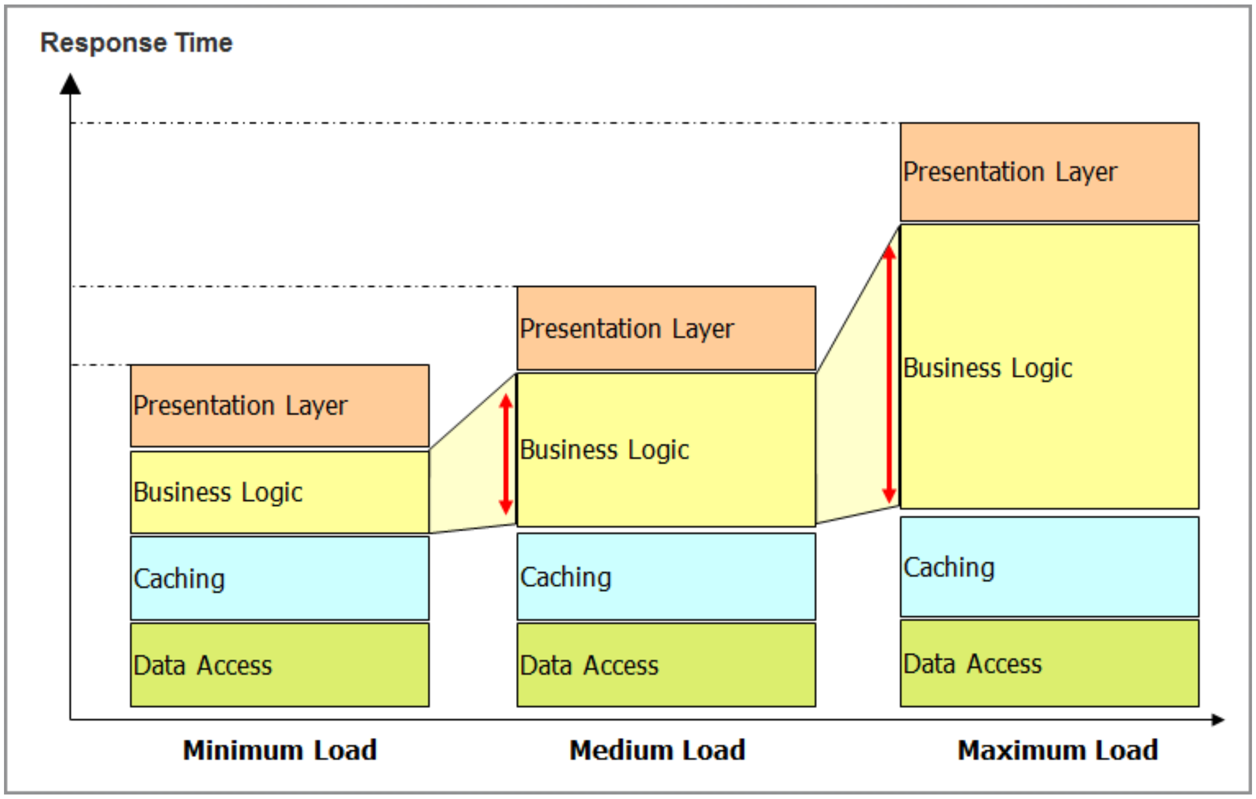


Figure 3.11: Optimizing and changing a component has effects on other components.

This example shows that you shouldn't optimize your components before making sure that there are no side effects. Increasing the cache size seems like a good practice to increase performance, but you must test it to know for certain.

When and how often should you do regression analysis? Here are three rules of thumb:

- Perform regression analysis every time you are about to make substantial architectural changes to your application, such as the one in the example above. In order to identify a regression of substantial architectural changes, it is often necessary to have the application deployed in an environment where you can simulate larger load. This type of regression analysis cannot be done continuously, as it would be too much effort.

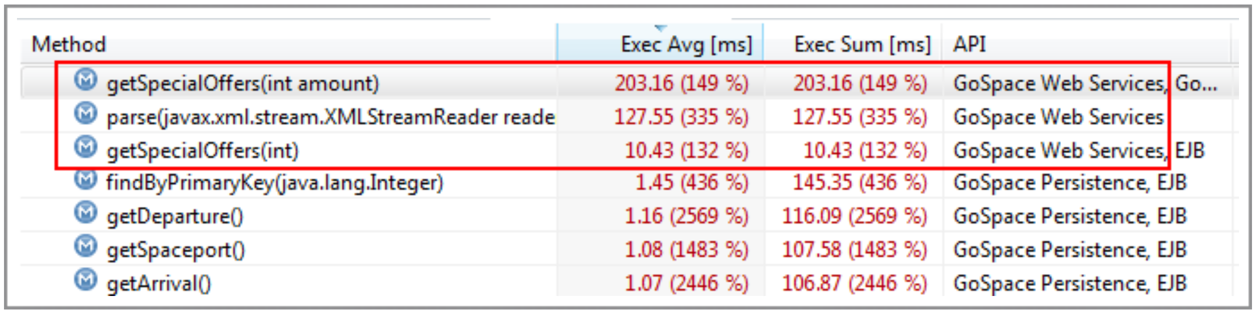
- Automate regression analysis on your unit and functional tests as explained in the previous sections. This will give you great confidence about the quality of the small code changes your developers apply.

- Perform regression analysis in the load- and performance-testing phases of your project, and analyze regressions of performance data you capture in your production environment. Here the best practice is to compare the performance metrics to a baseline result. The baseline is typically a result of a load test on your previously released software.

Comparing results of load and performance tests or results captured in a production environment will not be as easy as comparing the results of two unit or functional tests, as there are many measures that get captured. Therefore the first steps to analyze the captured data are as follows:

1. Focus on high-level metrics, such as response times of transactions or service requests.
2. Drill deeper into the measures of transactions that show a regression.

Figure 3.12 shows the comparison of a load test against the baseline results. You can see that certain web service calls are much slower than in the baseline. As a next step, we would dig deeper into this particular problem to identify what exactly changed in these web service calls.



Method	Exec Avg [ms]	Exec Sum [ms]	API
getSpecialOffers(int amount)	203.16 (149 %)	203.16 (149 %)	GoSpace Web Services, Go...
parse(javax.xml.stream.XMLStreamReader reade	127.55 (335 %)	127.55 (335 %)	GoSpace Web Services
getSpecialOffers(int)	10.43 (132 %)	10.43 (132 %)	GoSpace Web Services, EJB
findByPrimaryKey(java.lang.Integer)	1.45 (436 %)	145.35 (436 %)	GoSpace Persistence, EJB
getDeparture()	1.16 (2569 %)	116.09 (2569 %)	GoSpace Persistence, EJB
getSpaceport()	1.08 (1483 %)	107.58 (1483 %)	GoSpace Persistence, EJB
getArrival()	1.07 (2446 %)	106.87 (2446 %)	GoSpace Persistence, EJB

Figure 3.12: Identify regressions in service calls by comparing a load-testing result with the baseline.

The difference between any two test runs can be determined except when comparing method-call results. Anything that can be measured over time is fair game for regression analysis. So in addition to performance, we can measure the effects of increased load over time and use this data to test application scalability. This is especially useful for testing optimization scenarios, where you might want to test general optimization effectiveness versus the effectiveness for a specific scenario.

We had a situation in our own product development where the goal was to improve performance in analyzing a memory dump. We achieved significant improvements for heaps up to 2 GB, but as the heap grew beyond that size, the new algorithm led to greater deterioration. This problem was discovered using an automated test with different heap sizes in order to find out whether the new implementation actually scaled.

Figure 3.13 shows the performance-regression dashboard used to track this problem. You can see that the initial regression was identified on May 19th. After fixing the problem, the initial problem for that heap size went away, but a regression was immediately identified for another heap size. These bug fixes had a positive but also a negative impact on the overall application performance.

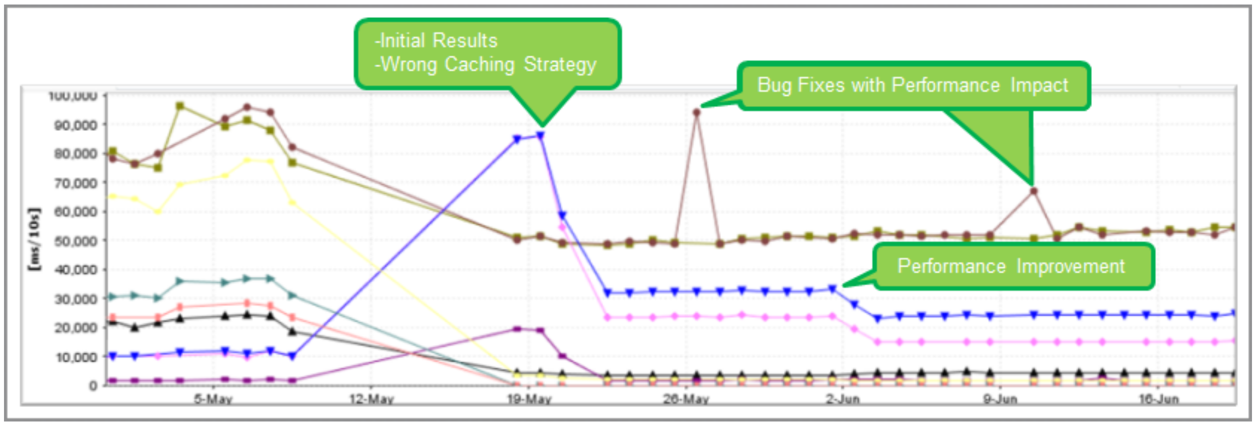


Figure 3.13: Identifying performance regression through continuous performance engineering

This example emphasizes how important it is to test specific scenarios where we might expect problems. By testing the same code with different input values, it's possible to determine the potential side effects of a code fix.

Automating this type of regression analysis allows you to identify problems as they get introduced in the code. It allows your developers to immediately fix problems and eliminate side effects, but only as long as the memory of what they have been working on is fresh. All these efforts result in higher-quality code, a smoother development cycle, and shorter testing cycles.

Enforcing Development Best Practices

In the previous sections of this chapter we have taken a detailed look at the technical aspects of performance engineering in development and as part of continuous integration. In this section we discuss the most common problem patterns and consider the organizational and conceptual aspects of each.

Plan and Define Quality

Quality starts with a well-thought-out definition of new features, along with a detailed features and capabilities description—the new-feature *story*. The software architects should review every story before it becomes part of a final development sprint by enforcing the following additional requirements:

- **Testability**: Every new piece of code must be testable, and as result of the story implementation engineers must produce unit and functional tests. These tests help verify the functional correctness of the implementation. When using code-coverage tools, it is recommended that you specify the code-coverage percentage of these tests, as well.

- **Architectural rules**: Developers must make sure that their code adheres to the defined architectural rules, such as “No duplicate database query for one operation” or “Do not transfer more than 100 KB per remoting call.” Having unit and functional tests allows you to automate rule validation, as discussed earlier in this chapter.
- **Performance requirements**: A new feature must have performance requirements, such as “The Save operation must not take longer than 500 ms with 100 concurrent users on the system.” It is important to define the performance as well as the load condition under which this performance requirement must be met.
- **Documentation**: Code and end-user documentation improve quality. Enforcing a high level of code documentation allows developers to better understand what the code is supposed to do and with that, to make better code changes in the future. End-user documentation makes it possible to test new features as they’re expected to be used. This enables better use-case testing and ensures that the functional quality of the product is high.

Enable Your Engineers with Tools

As discussed throughout this chapter, developers need tool support to analyze the behavior of the implemented code, verify architectural correctness, and check performance and scalability. Everything starts with the developers, and developers require the right tools to ensure that their code adheres to all defined architectural rules. This includes the following:

- **Profiling tools**: CPU or memory profiling allows developers to analyze the performance of their application code and to identify algorithms that don’t perform well, overused or incorrectly used objects (possible causes of memory leaks), or high memory usage.
- **Tracing and diagnostic tools**: These tools show how code impacts application architecture, and provide the input for architectural-rule validation. When developers have the same tools used later on by architects, they are able to examine performance metrics before they commit their code changes, in a process of continuous integration.
- **Testing tools**: In addition to unit tests, there are many types of tests that developers should execute on their local workstation before committing any code changes. Make it standard practice to run functional tests that test code through the end-user interface, as well as small-scale load tests. Open-source tools, such as Selenium or JMeter, are good, easy-to-use tools.
- **Real vs. test data**: For data-driven applications, using a local subsetted copy of the database is often not sufficient to detect many database-related problems. Giving developers access to a copy of a real production-like database helps them find and eliminate data-driven problems from the start.

Educate Your Engineers

dynaTrace continually educates engineering staff through different channels:

- **Classroom training**: Engineers get mandatory training to understand the application they will be working on. Even though engineers typically work on only a subset of the application, understanding the full application allows them to better predict where their code changes may have an impact.
- **Access to literature**: Architects pick literature that explains and teaches new concepts in development. This literature must be made available to engineers so that they can learn the latest developments and advance their own skills. Deciding to use a third-party framework library should trigger purchases of literature about that framework to make sure the framework is used in the best possible way. Without making proper information available, you may end up implementing code based on sample applications seen on the Web instead of doing it the way explained by experts.
- **Code reviews**: During code reviews (another great best practice adopted from agile development) senior engineers have the ability to coach junior engineers. Coaching encourages high-level knowledge transfer that's reinforced by the shared experience of real-life, cooperative engineering.
- **Regular update meetings**: We established regular engineering update meetings to discuss project statistics and software quality. This allows every engineer to see how much impact his or her work has. For instance, when we ship a poor-quality product, we see an increase in customer-support complaints. With high-quality code, we have more time to implement new features. A regular update allows everybody to focus again on what is currently important.

Education time is well-invested time. It helps you engineers create higher-quality code with less time spent finding and fixing problems.

Automate, Automate, Automate, and Report

The more tasks you can automate, the better. You get faster results on code quality, which then allows you to tell your engineers whether they can continue coding new features or if they need to fix problems to bring software quality back on track. Not everything can be automated, but thanks to continuous integration and the availability of ever-advancing toolsets, we can automate things like unit, component, integration, functional, and load tests. We can automate performance analysis as well as architecture validation. All of this is possible, but it requires the dedication of the engineering team:

- **Engineers must** write testable code and the actual tests to verify this code before committing changes. Therefore, it's essential to test locally before committing.

- **Architects must** define and enforce architectural guidelines and rules. Provide coaching and help engineers implement the automation process.
- **Automation engineers must** invest in the continuous-integration process to automate test execution and provide meaningful reports to engineering.

The last piece of advice is on reporting: bring the results back to engineering as quickly as possible. At dynaTrace we use several dashboards that show the status of current development. We see the number of tests executed, how many of them failed and succeeded, and whether we have a stable build or not. These dashboards are visible throughout the office so that every engineer can see what's going on upon entering or leaving the building, or even just when getting a cup of coffee.

Load Testing—Essential and Not Difficult!

When project schedules are tight and releases frequent, the temptation to skimp on the load-testing cycle is great. However, not many customers have the loyalty or patience to put up with the performance issues likely to result from taking such ill-considered shortcuts. It's more likely to result in immediate and catastrophic system failures worthy of broadcast throughout the Twittersphere!

Perhaps we've exaggerated things a bit, but load testing is critically important because it answers four key performance questions:

- How well does my application scale?
- When does my application break?
- Can we handle the expected peak load with acceptable response time?
- How many resources do we need to handle expected and peak load?

Obviously, we must define an acceptable response time for each application, and this is something business, engineering, and operations must all agree on in advance. For instance, if business analysis shows that too many users leave when response time goes above 2 seconds for a catalog search, then engineering must optimize performance for this and operations must provide the required resources.

Figure 3.14 illustrates a typical response time, throughput, and CPU-usage graph. The higher the load on the system (throughput), the more CPU power is consumed to handle that additional load. When CPU resources (or any other system resource) are exhausted, we will see a rise in response time as the system takes longer to handle all incoming requests. When no more system resources are available, the system *breaks*. This results in even higher response times and lower throughput. Acceptable response time must fall to the left of the intersection point—ideally before CPU usage and throughput flatten out or start to fall (*see Figure 3.14*).

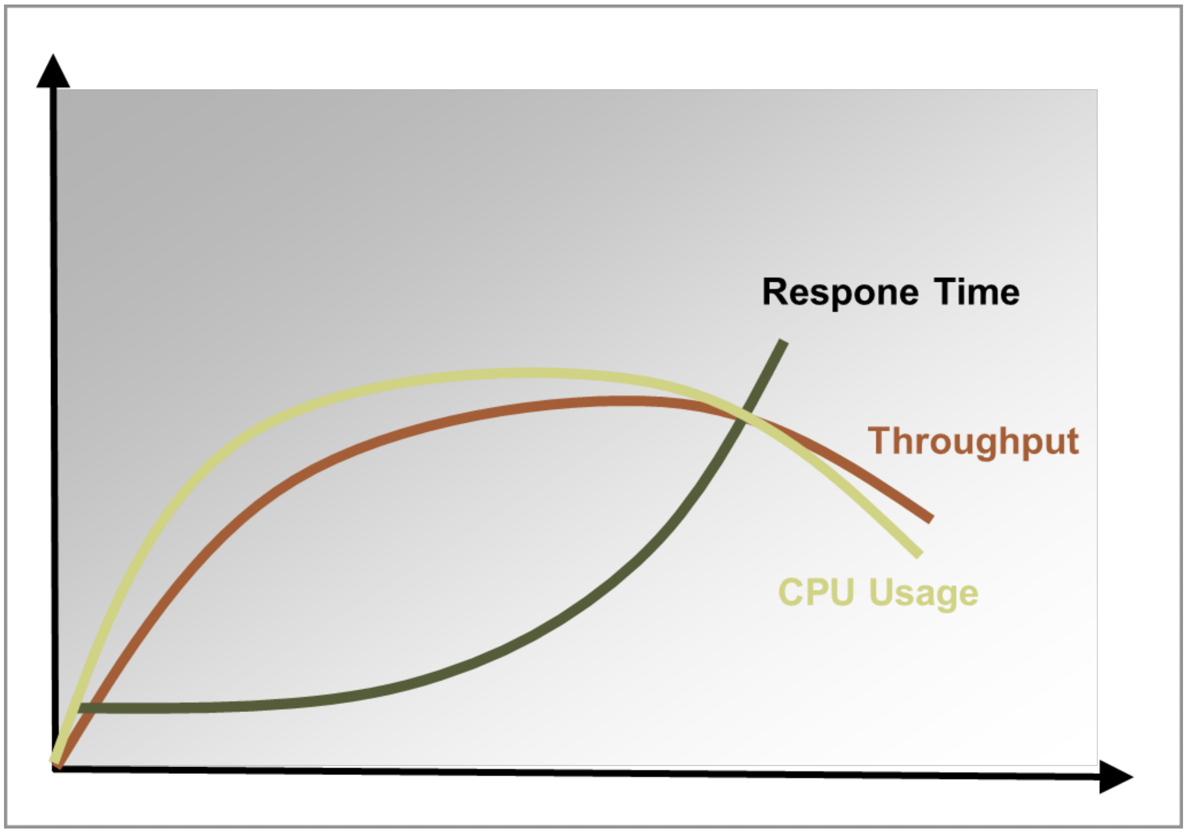


Figure 3.14: We can identify how much load can be handled in order to stay within acceptable response time.

Load Testing on the Web—It's Global!

In addition to load testing within the controlled environment of the enterprise IT department, web applications must be performant from just about any geographical location around the world (*Figure 3.15*). Using in-house load-testing tools to simulate HTTP requests on the protocol level—Load Testing 1.0—will not help us to answer questions like these:

- Can the application perform satisfactorily on all targeted hosts and browsers?
- Does it perform without problem in all geographical locations?
- Will it render correctly on all end-user devices, including mobile phones and tablets?
- Is the content-delivery network (CDN) properly configured?

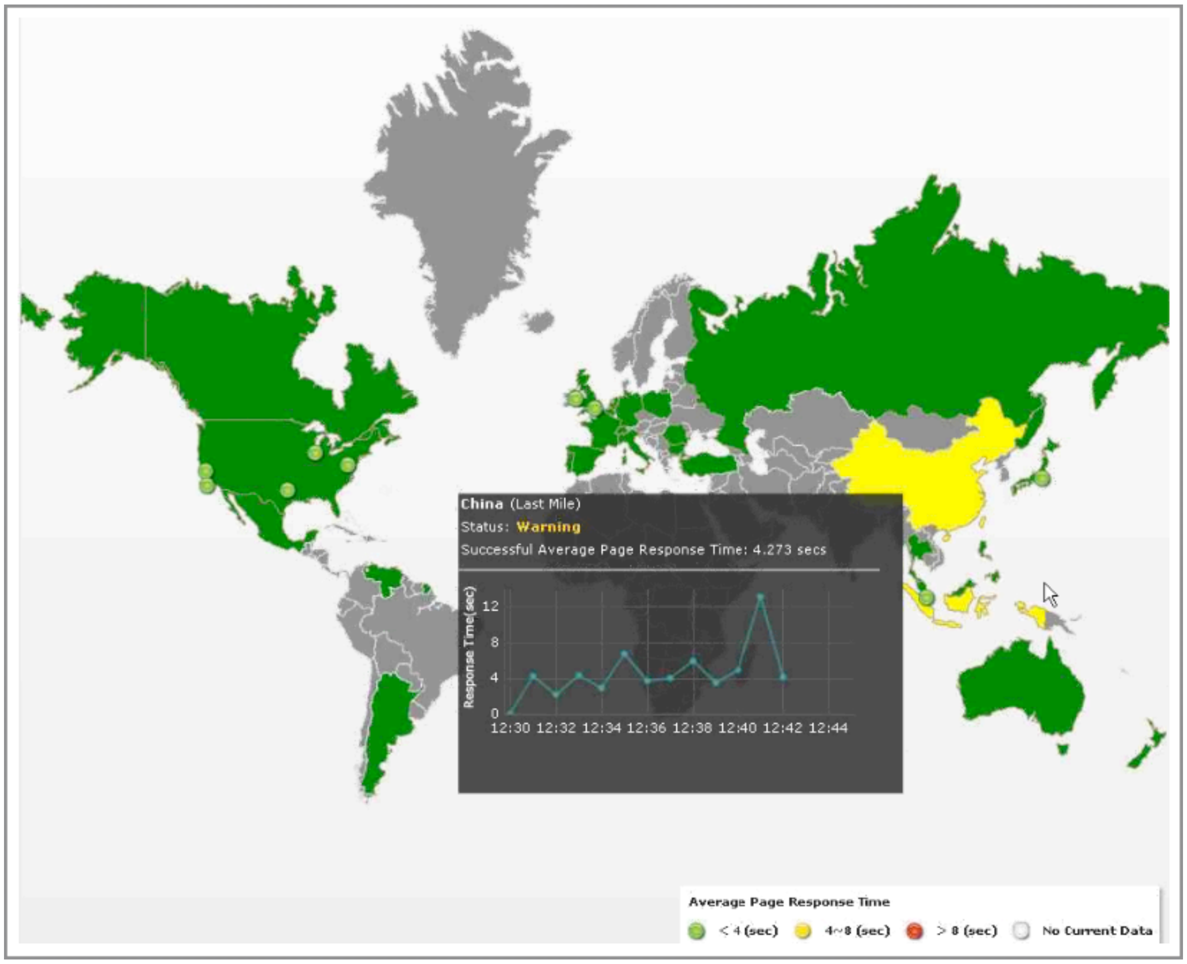


Figure 3.15: Modern web load testing includes global end-user performance analysis and highlights the geographical regions as well as specific browsers, with the respective performance characteristics.

In short, if you don't know the answers to the questions above, you can't release your product. Furthermore, it's not possible to know the answers without proper load testing!

You might ask, if proper load testing is so critical, why do so many companies minimize or completely avoid it? There are all sorts of excuses, including these:

1. It seems impossible to test realistic user load.
2. We don't have the tools, expertise, or hardware resources to run large-scale load tests.
3. It is too much effort to create and maintain test scripts.
4. Commercial tools are expensive.
5. We don't get actionable results for our developers.

Let's take each of these in turn.

1. It Seems Impossible to Test Realistic User Load

Indeed, it can be difficult to estimate a realistic user load or predict use cases, especially if you are about to launch a new website or service. At the same time, you should at least know how your new service will be used once launched, and how traffic is expected to evolve over time.

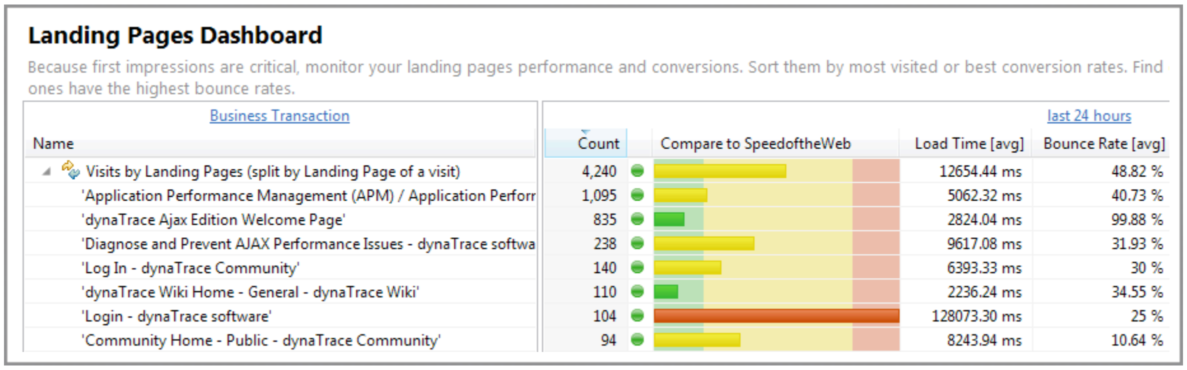
Estimating your traffic—It can help to see what your competitors have experienced by using online resources, such as Alexa, Compete, and Quantcast. You can read more about this on Sam Crocker's blog post, *7 tools to monitor your competitors' traffic,* http://socialmedia.biz/2011/01/10/7-tools-to-monitor-your-competitors-traffic/. Also, factor in how much money you spend on marketing and promotions and what conversion rate you expect. This will allow you to estimate peak loads.

What you know (analytics) and what you think will happen (growth estimates)—Estimating realistic load becomes easier when you are updating an existing site. You likely have end-user data from a tool like Compuware UEM, Google Analytics, or Omniture, and you can review a history of request volume in your web-server logs. With this data, you should have a good understanding of current transaction volume, customer behavior, and customer profiles (location, browser, connectivity). Factor in the new features, how many new users you expect to add, and the effect of any new launch promotions planned. Finally, make sure you talk to your marketing folks so that you don't have any rude surprises that bring down the system and waste their marketing efforts and dollars!

Combining all this data allows you to answer the following questions:

- What are the main pages and transaction paths I need to test?
- What's the peak load and what is the current and expected page load time?
- Where are my users located, geographically?
- What browsers do my customers use?
- What are the main browser/location combinations we need to test?

Figures 3.16, 3.17, and 3.18 give you some examples on how we can extract data from services such as Compuware UEM or Google Analytics to better understand how to create realistic tests.



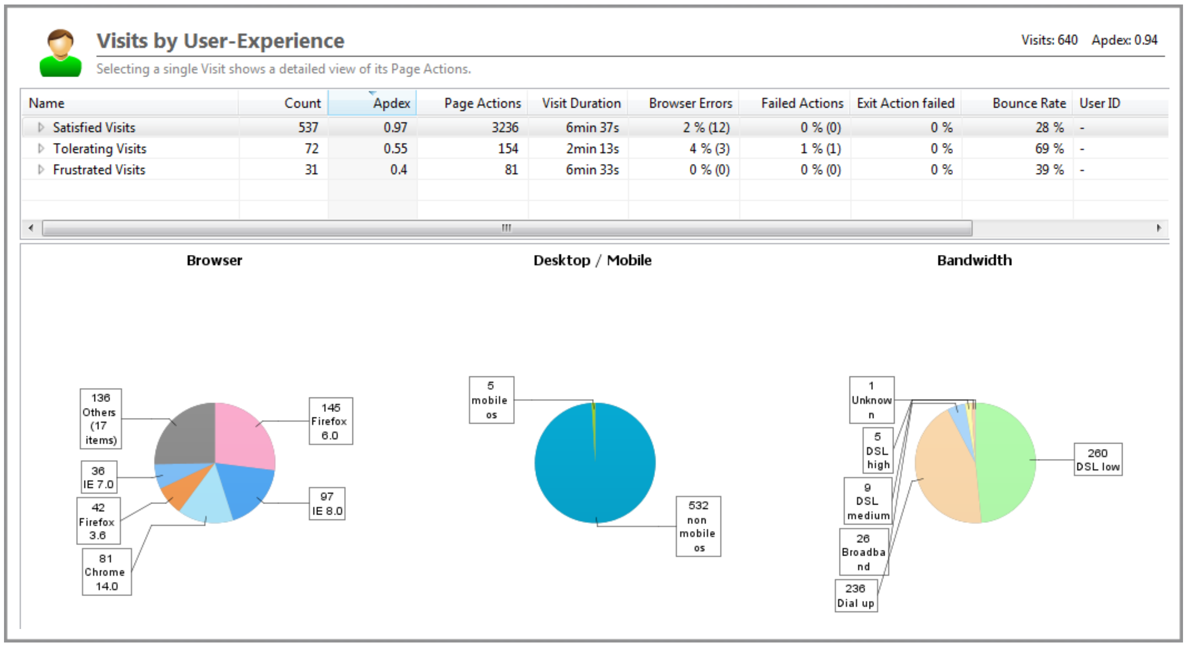
Landing Pages Dashboard

Because first impressions are critical, monitor your landing pages performance and conversions. Sort them by most visited or best conversion rates. Find ones have the highest bounce rates.

Business Transaction | last 24 hours

Name	Count		Compare to SpeedoftheWeb	Load Time [avg]	Bounce Rate [avg]
Visits by Landing Pages (split by Landing Page of a visit)	4,240	●		12654.44 ms	48.82 %
'Application Performance Management (APM) / Application Perforr	1,095	●		5062.32 ms	40.73 %
'dynaTrace Ajax Edition Welcome Page'	835	●		2824.04 ms	99.88 %
'Diagnose and Prevent AJAX Performance Issues - dynaTrace softwa	238	●		9617.08 ms	31.93 %
'Log In - dynaTrace Community'	140	●		6393.33 ms	30 %
'dynaTrace Wiki Home - General - dynaTrace Wiki'	110	●		2236.24 ms	34.55 %
'Login - dynaTrace software'	104	●		128073.30 ms	25 %
'Community Home - Public - dynaTrace Community'	94	●		8243.94 ms	10.64 %

Figure 3.16: Use analytics to identify pages on which end users "land" on your web site. They're typically the homepage, promotional pages, or pages indexed by search. Make sure to focus your testing on these landing pages, as users' first impressions determine whether they stay at or leave your site.



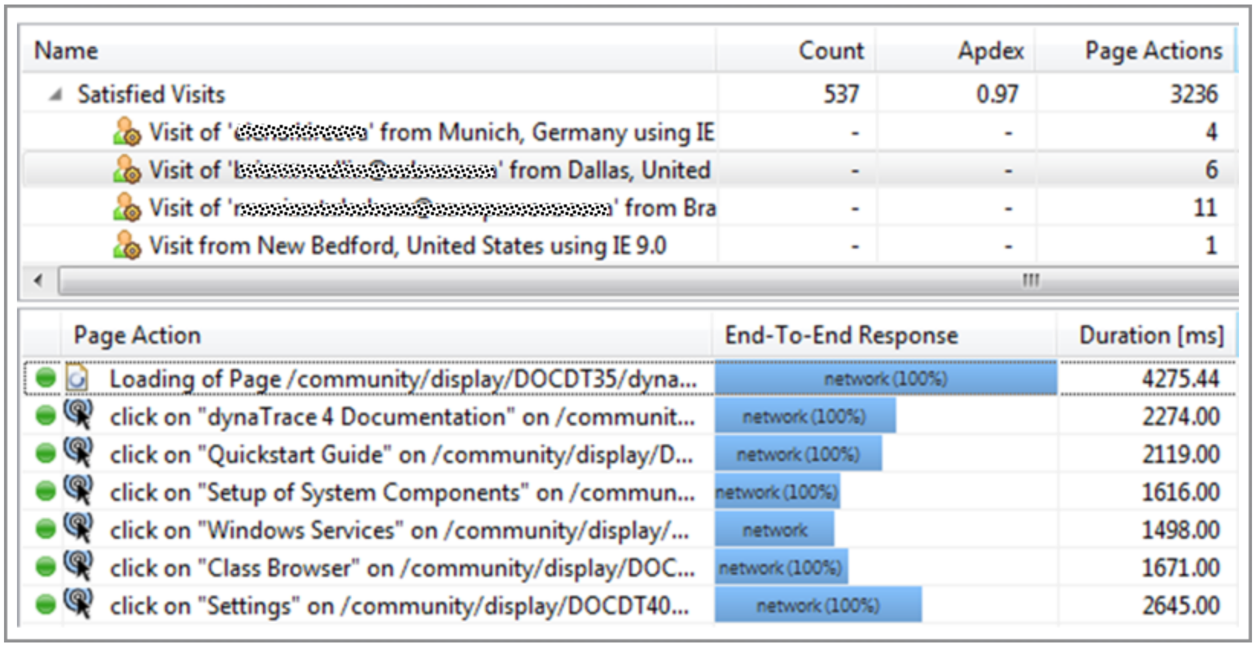
Visits by User-Experience

Selecting a single Visit shows a detailed view of its Page Actions.

Visits: 640 Apdex: 0.94

Name	Count	Apdex	Page Actions	Visit Duration	Browser Errors	Failed Actions	Exit Action failed	Bounce Rate	User ID
Satisfied Visits	537	0.97	3236	6min 37s	2 % (12)	0 % (0)	0 %	28 %	-
Tolerating Visits	72	0.55	154	2min 13s	4 % (3)	1 % (1)	0 %	69 %	-
Frustrated Visits	31	0.4	81	6min 33s	0 % (0)	0 % (0)	0 %	39 %	-

Figure 3.17: Analytics gives us insight into the actual distribution of browser and browser versions of our real end users. These and the available bandwidth have a major impact on end-user response time. This information allows us to do more-realistic tests as we can mimic our real end users' environments.



Name	Count	Apdex	Page Actions
Satisfied Visits	537	0.97	3236
Visit of '[redacted]' from Munich, Germany using IE	-	-	4
Visit of '[redacted]' from Dallas, United	-	-	6
Visit of '[redacted]' from Bra	-	-	11
Visit from New Bedford, United States using IE 9.0	-	-	1

Page Action	End-To-End Response	Duration [ms]
Loading of Page /community/display/DOCDT35/dyna...	network (100%)	4275.44
click on "dynaTrace 4 Documentation" on /communit...	network (100%)	2274.00
click on "Quickstart Guide" on /community/display/D...	network (100%)	2119.00
click on "Setup of System Components" on /commun...	network (100%)	1616.00
click on "Windows Services" on /community/display/...	network	1498.00
click on "Class Browser" on /community/display/DOC...	network (100%)	1671.00
click on "Settings" on /community/display/DOCDT40...	network (100%)	2645.00

Figure 3.18: Analyzing click sequences of real users allows us to model load-test scripts that reflect real user behavior.

Don't forget about CDNs, proxies, network latency, etc.—Not every end-user request makes it to our application environment. Many components impact load times, including connection speed, browser characteristics, latency, content-delivery network, and geographic location. There are cloud-based testing services that can execute load from different data centers using different browsers and different global locations.

2. We Don't Have the Tools, Expertise, or Hardware Resources

In most environments, load testing is not a daily activity, making it difficult to justify the amount spent on commercial tools to generate load, hardware resources necessary to simulate load, and the training investment for personnel.

But this is fast becoming an obsolete mindset because a new generation of cloud-based load-testing services obviates the need for such large investments. The key advantages include the following:

- **Tools**—You pay for only load tests performed, and not for the time software sits on the shelf becoming out-of-date.
- **Resources**—The hardware resources to generate load are provided and managed by the service provider.
- **Skills**—The service is staffed by testing experts who generate scripts and maintain the testing environments every day.

3. It's Too Much Effort to Create and Maintain Test Scripts

This might be a valid claim if it weren't for two simple solutions:

- **Free vs. commercial tools**—While free load-testing tools offer easy record/replay scripting, they don't always support a scripting language sufficient for creating and maintaining custom scripts. A number of commercially available tools make it much easier to solve scripting issues, though you do have to pay for this solution.
- **Tools vs. services**—For organizations unwilling or unable to use testing tools efficiently, cloud-based load-testing services usually include professional script generation and script maintenance as part of the subscription fee.

4. Commercial Tools Are Too Expensive

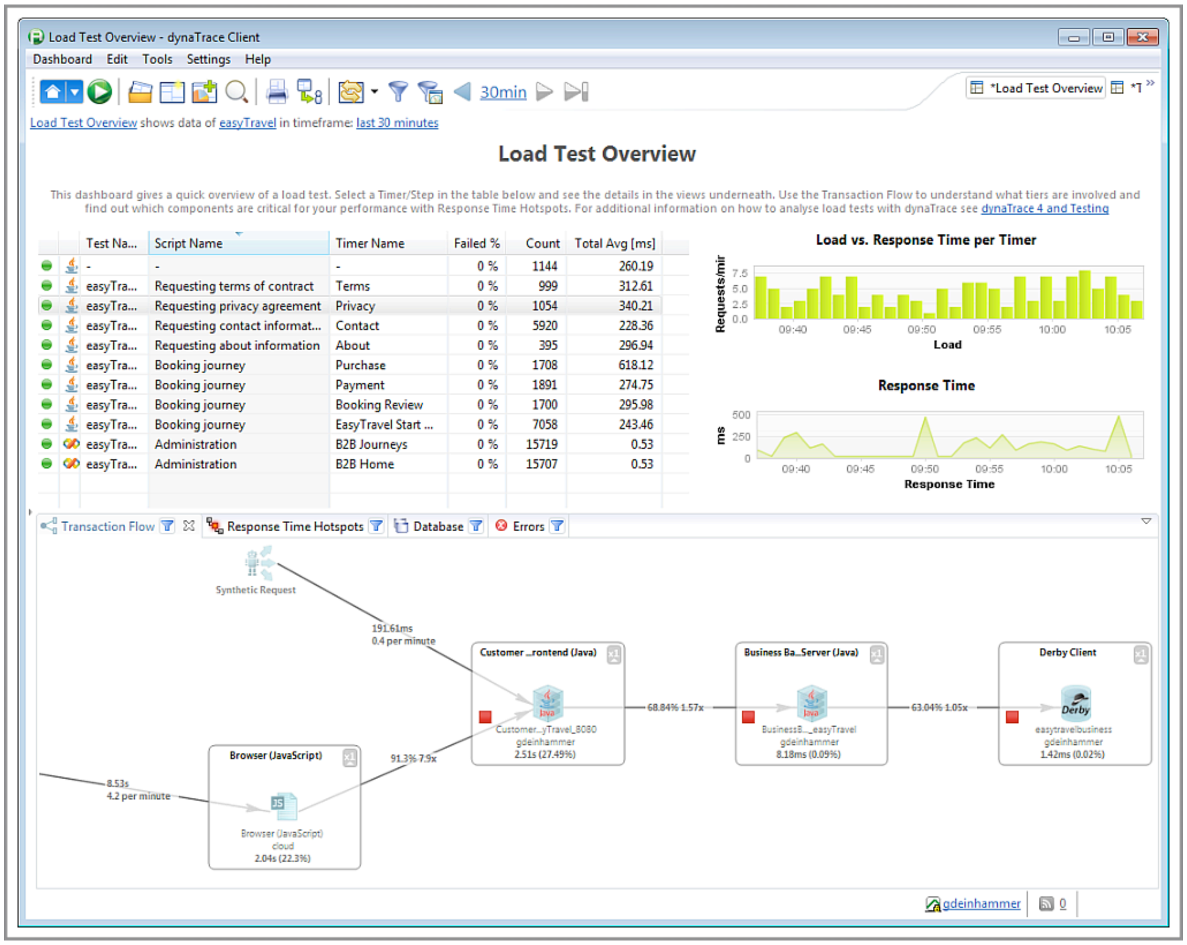
Perhaps, but if the alternative is software failure due to lack of scaling, then it's hard to justify cutting corners. As you've probably guessed, cloud-based load-testing services are a viable, and often preferred, alternative. For one thing, any costs associated with this service are for actual virtual users and execution time, so there's no wasted expenditure. We can evaluate this on the basis of cost per virtual user hour. You need only define three variables—test frequency, user load, and service rate—to calculate the actual cost.

5. We Don't Get Actionable Results for Our Developers

It is not enough to run a load test, report a problem to your developers, and expect them to be able to fix it. Imagine you are told that an application fails to meet the performance criteria when loaded to 100 or more virtual users. You must return to the code, add diagnostic log output, and have the tests run again just to produce some actionable data. This leads to an inefficient cycle of testing and retesting that can jeopardize project schedules and frustrate all involved.

This is old-style testing and it's not up to the task of dealing with modern, web-based applications. For this you need an application performance–management solution, which works alongside load-testing tools to provide very specific actionable data. With minimal overhead and without additional code, it's possible to pinpoint problematic database statements, inefficient synchronization code, heavyweight web service calls, and wasteful memory management, as well as a host of other items.

The dashboard in *Figure 3.19* shows a exactly whether it is the Web-, App- or Database Server that contributes to response time of executed tests. Clicking on Database, Errors or Response Time Hotspot provides the list of methods that took long to execute, the SQL statements that were called on how long they took to execute as well as which errors happened in the application.


Load Test Overview - dynaTrace Client
Dashboard Edit Tools Settings Help
30min
*Load Test Overview
Load vs. Response Time per Timer
Requests/min
Load
Response Time
ms
Transaction Flow
Response Time Hotspots
Database
Errors
Synthetic Request
191.61ms 0.4 per minute
Customer ...rontend (Java)
Customer...yTravel_8080 gdeinhammer 2.51s (27.49%)
68.84% 1.57x
Business Ba...Server (Java)
BusinessB..._easyTravel gdeinhammer 8.18ms (0.09%)
63.04% 1.05x
Derby Client
easytravelbusiness gdeinhammer 1.42ms (0.02%)
Browser (JavaScript)
91.3% 7.9x
8.53s 4.2 per minute
Browser (JavaScript) cloud 2.04s (22.3%)
gdeinhammer

Load Test Overview shows data of easyTravel in timeframe: last 30 minutes

Load Test Overview

This dashboard gives a quick overview of a load test. Select a Timer/Step in the table below and see the details in the views underneath. Use the Transaction Flow to understand what tiers are involved and find out which components are critical for your performance with Response Time Hotspots. For additional information on how to analyse load tests with dynaTrace see dynaTrace 4 and Testing

Test Na...	Script Name	Timer Name	Failed %	Count	Total Avg [ms]
-	-	-	0 %	1144	260.19
easyTra...	Requesting terms of contract	Terms	0 %	999	312.61
easyTra...	Requesting privacy agreement	Privacy	0 %	1054	340.21
easyTra...	Requesting contact informat...	Contact	0 %	5920	228.36
easyTra...	Requesting about information	About	0 %	395	296.94
easyTra...	Booking journey	Purchase	0 %	1708	618.12
easyTra...	Booking journey	Payment	0 %	1891	274.75
easyTra...	Booking journey	Booking Review	0 %	1700	295.98
easyTra...	Booking journey	EasyTravel Start ...	0 %	7058	243.46
easyTra...	Administration	B2B Journeys	0 %	15719	0.53
easyTra...	Administration	B2B Home	0 %	15707	0.53

Figure 3.19: Analyzing a particular test scenario shows us the response time and load distribution over time, as well as detailed diagnostics information. This allows us to analyze performance and scalability issues of individual tested sites and scenarios.

Convincing the Team that Load Testing Is Worth the Effort

You should now be well prepared to convince management and the engineering/testing team to undertake a complete battery of load tests. We've answered most of the common objections and shown that it is both worth the effort and not as difficult as many assume.

Discovering Performance and Scalability Problems—Load Testing at Work

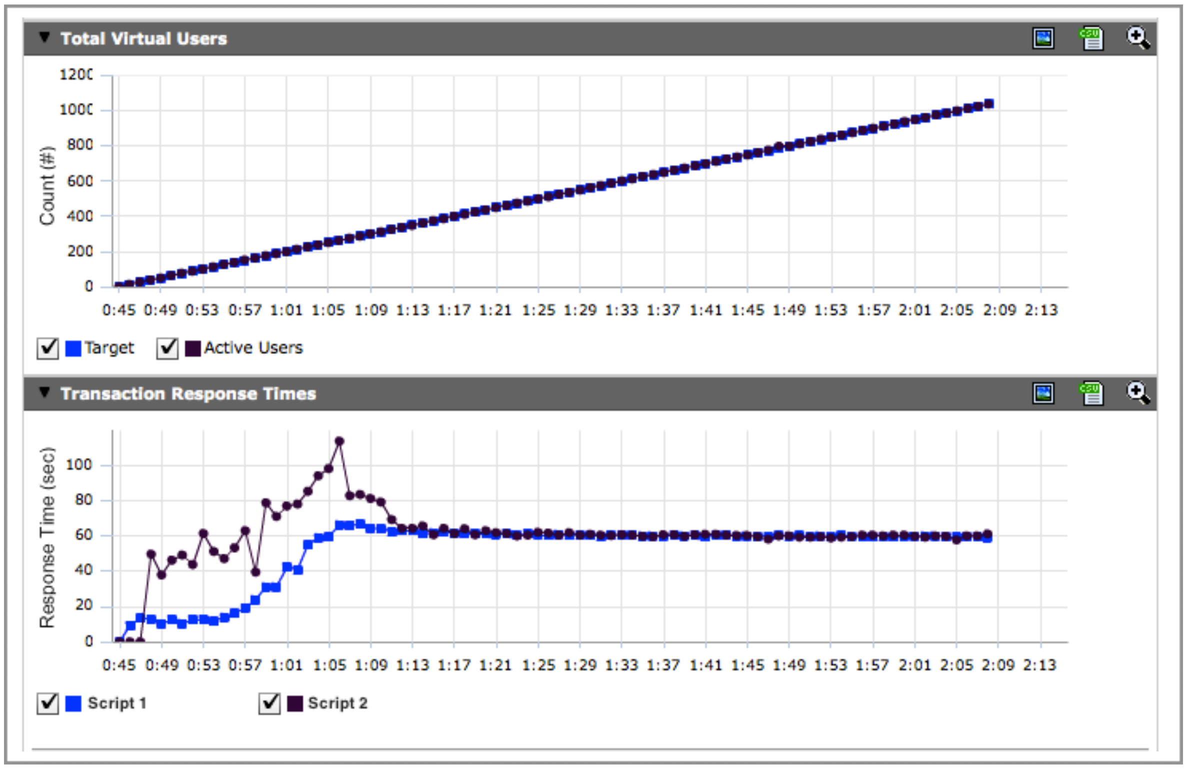
Load testing, properly executed, gives us confidence in the reliability of our applications, but not simply because we know our software has met or exceeded our testing standards. The process of load testing uncovers problems that, when fixed, improve software quality, reveal issues of deployment that can lead engineering to develop best practices, and can help developers identify and avoid potential maintenance issues down the road.

Problems exposed in load testing usually appear in four interrelated areas—network, server, application, and third-party components. Traditional load-testing methods are good at exposing the first three, but the fourth requires a different approach, which encompasses the entire application in the test scenario. We refer to this as Load Testing 2.0.

Network Problems

As the most fundamental of the four problem areas, network issues are likely to show up early and catastrophically. Problems range from the simple, such as an incorrect firewall license, to the bizarre.

In one bizarre instance, the load test was progressing in a nicely predictable fashion. Response time was increasing with load, when suddenly, response times leveled out. This would not necessarily be a bad thing, except there was also a sudden leap in page errors, as you can see in *Figure 3.20*.



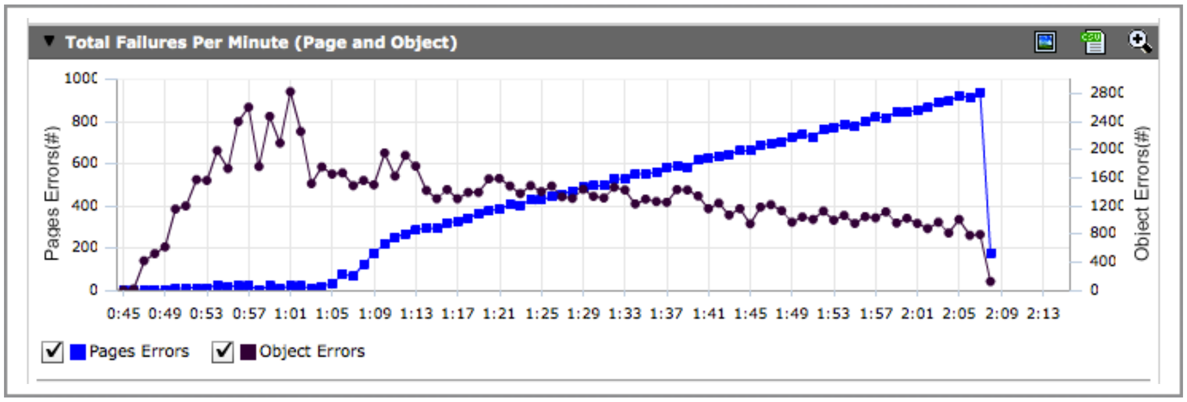


Figure 3.20: Use all of the input from your monitoring to track down testing anomalies. In this instance, a problem that appeared to be an application level issue was actually the result of an unexpected network redirect.

Immediately, all eyes fell towards the application layer as the customer and the load-testing team assumed this is where issues would arise as load increased. But the team monitoring the application layer reported an odd situation—there was no sign of any traffic arriving on the application-layer systems in the tools collecting data. Tracing the issues out from the application layer, the team could see traffic hitting the edge routers, firewalls, and load balancers, and then vanishing before making the leap to the web servers and application layer in the datacenter being tested.

One final area was investigated. As the test was being performed on only one of the company's datacenters, all other traffic had been redirected to the second datacenter. At about the time traffic stopped reaching the application layer at the datacenter being tested, response times at the current production datacenter started to increase.

How could this happen? This site was not being tested! Further investigation found that the load balancers on the production system were designed to redirect traffic automatically to the backup site above a certain level. When the load test reached that volume, all of the traffic was routed to the system that was currently serving customers, taking that site offline!

Without the network, traffic doesn't flow, data doesn't get delivered, and pages don't get displayed.

Server Problems

Server issues should be treated separately from application issues, as a configuration issue with the underlying server software is different than a problem with the code that is running on that server.

Consider a classic instance—one in which a web server was configured using the default settings. Under a light load, the server started rejecting most incoming connection requests, but allowing just enough through to be suspicious. When the web-server status was reviewed, it was clear that the configuration for maximum threads was set for the default value—in this instance, fewer than 100! A quick configuration on the master configuration that was pushed out to all of the web servers fixed this, and the test was restarted quickly.

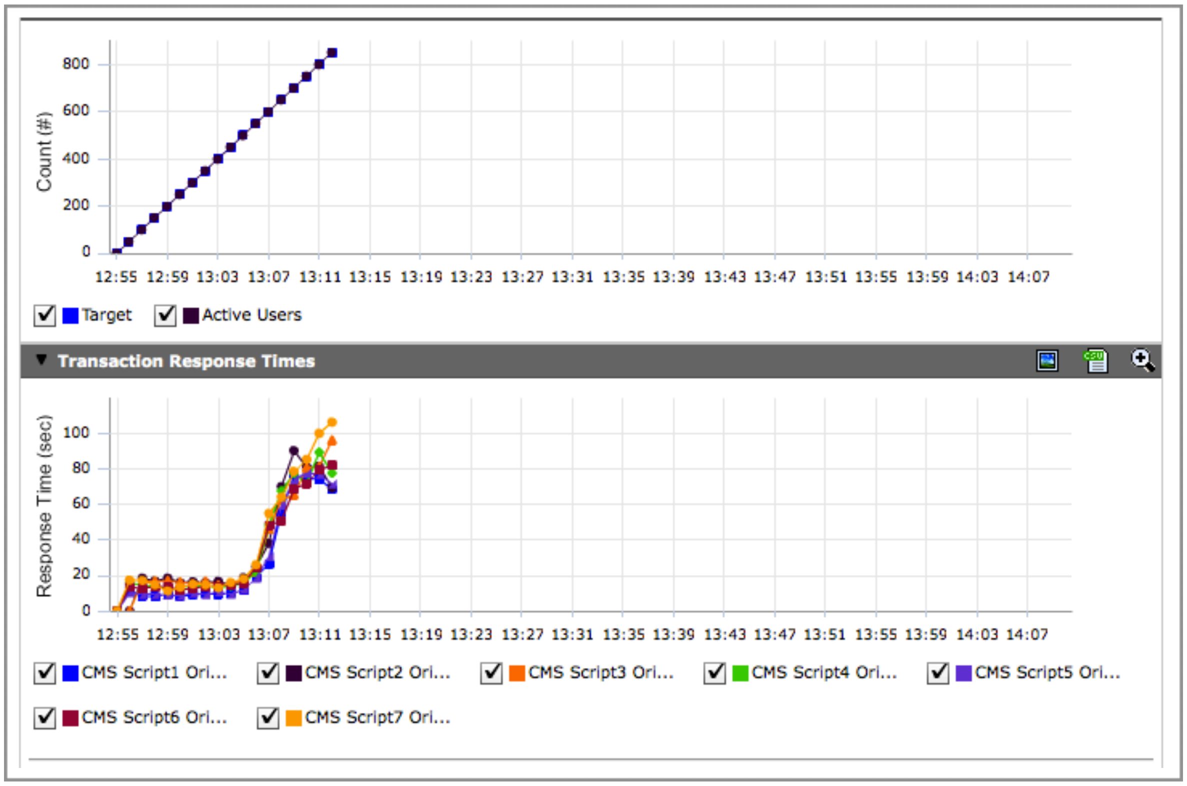
Application Problems

Under load, the complexity of thousands of requests per second, each one unique, each one requiring system resources, making database calls, being transformed into different formats for further processing, makes the application and data layers—where incoming requests are executed, processed, and formatted before results are passed back to the client—among the most critical parts of the application.

With modern web applications often composed of multiple complex application and data layers, finding the problem can sometimes be a challenge. For example, one customer took multiple testing cycles and a great deal of investigation to discover the two root causes of a critical performance issue:

- A mystery scheduled job ran at 6 minutes past every hour and slowed the application to the point of being unusable.
- One of the customer's two web-application servers was, because of the client's attempt to save money, also running a middleware server that handled load for the entire system.

Neither the scheduled job or the overworked middleware server had been caught in previous tests examining components of the system in isolation, and it wasn't until the entire system was brought together and put under load that these issues were detected (*Figure 3.21*).



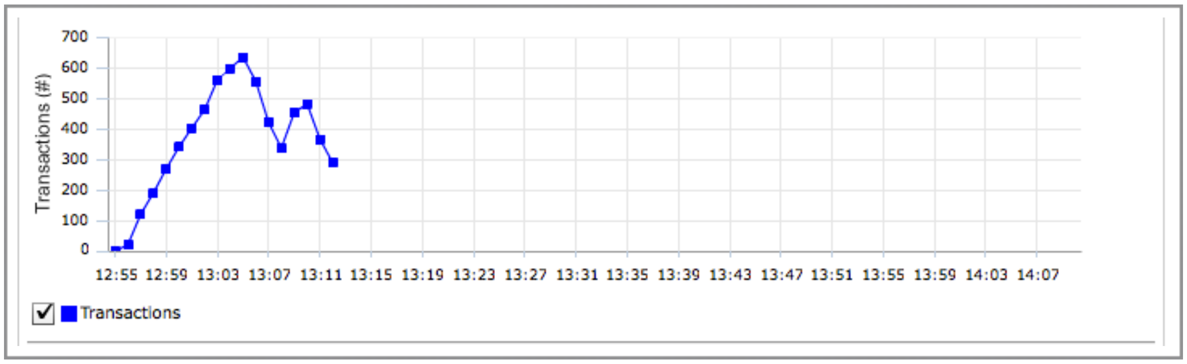


Figure 3.21: Application issues can stop a load test in its tracks. Here, the application ran out of resources at 500 virtual users and caused response times to increase exponentially.

In this instance, the application had not yet gone live. But the customer was running the load test at the traffic volumes projected for their upcoming busy season, making this issue more than just *two* technical concerns. If the application wasn't able to process and deliver the data with speed and efficiency under all loads, these *two* technology problems could have quickly become a core business problem affecting the revenue of the entire company.

Third-Party-Component Problems

Adding third-party content to the load-test equation is a tricky thing as, by its very nature, this content is external to your organization's direct control. Your team will have no insight into how the application is performing under load other than when performance begins to degrade.

In the delivery of modern web applications, third-party content can take multiple forms. An incomplete list includes the following:

- Content -delivery networks (CDNs)
- Ad providers
- Search services
- Analytics services
- Customer-satisfaction surveys
- Charts and images
- Social media.

Load testing with third-party content included in the content mix can prevent unexpected performance issues from being discovered by your customers after testing is completed. Consider the examples in the following paragraphs.

In a hypothetical example, if the analytics service demands that the tag be placed at the start of the page, and their tag calls are optimized for only a certain volume of traffic, this JavaScript could block the loading of the rest of your content in the customer browser, making the customer unhappy with you, and consequently affecting your brand and potentially decreasing revenue.

The use of traffic shaping or limiters by third parties can be detected only when the entire application is placed under a well-designed load. At a certain traffic volume, the third-party service you have selected to handle search, analytics, or even ad-serving may begin to throttle or reject incoming requests, slowing or blocking critical page elements and affecting overall customer experience. Testing the new volumes with the entire application is one way to prevent these surprises from appearing after the application has gone live.

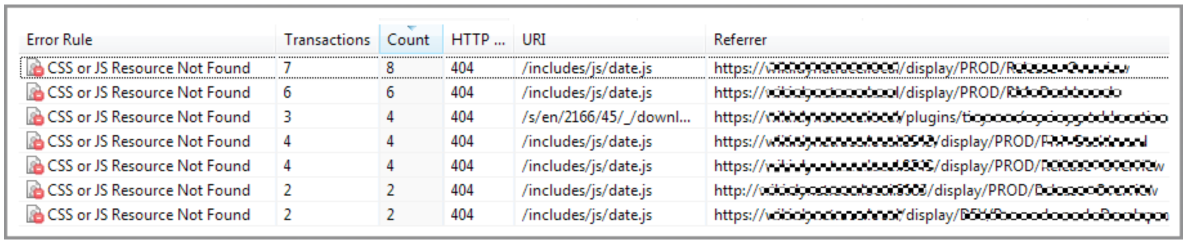
When load testing third-party content from an external perspective, problems may arise when the third party's infrastructure is incompatible with the testing format you have chosen. A key example of this is the use of distributed load generation from large datacenter or cloud providers. Concentrated load generation from a few distributed centers may sound perfect for testing a CDN, but there is a potential for issues, as this methodology may *short-circuit* the CDNs ability to appropriately distribute the load based on a customer's IP address. Involving the CDN in the planning and execution of the tests cycles will help overcome this while still validating the effectiveness of the content delivery.

Testing and validating the performance of this outside content allows you to validate all aspects of your application (including those you don't directly control) as able to handle the tested load. In the end, if your third-party content providers lack the ability to scale with your online application, they could quickly go from being a supporting partner to a dragging anchor.

Identifying Deployment Problems Early

Many mistakes are made when deploying an application into production. Let's have a look at an example and look over a check list to avoid things like missing files, misconfigured Web Server settings or outdated JavaScript framework files.

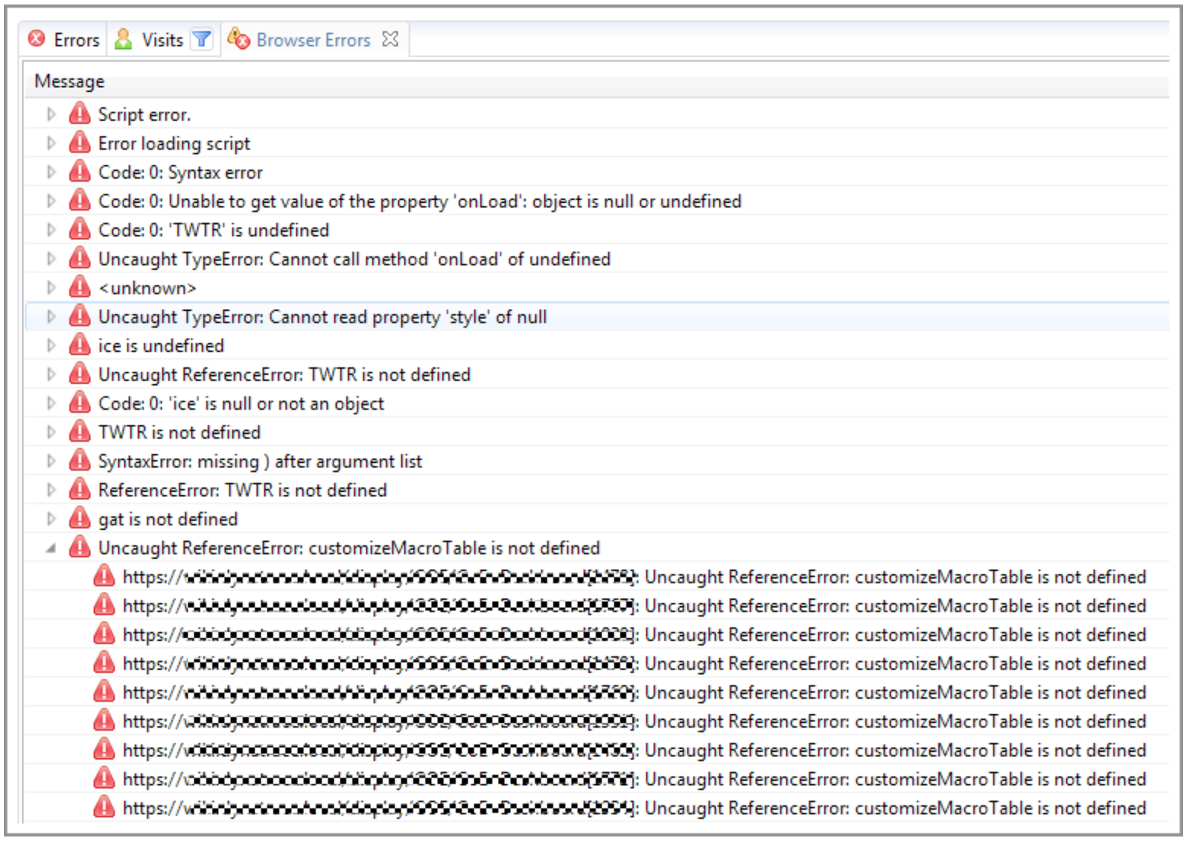
It may seem obvious that when deploying an application, it is important to deploy all of the application content at the same time, including static resources, such as CSS, JavaScript, and image files that can easily be forgotten, causing a cascade of errors (*see Figure 3.22*).



Error Rule	Transactions	Count	HTTP ...	URI	Referrer
CSS or JS Resource Not Found	7	8	404	/includes/js/date.js	https://[redacted]/display/PROD/[redacted]
CSS or JS Resource Not Found	6	6	404	/includes/js/date.js	https://[redacted]/display/PROD/[redacted]
CSS or JS Resource Not Found	3	4	404	/s/en/2166/45/_/downl...	https://[redacted]/plugins/[redacted]
CSS or JS Resource Not Found	4	4	404	/includes/js/date.js	https://[redacted]/display/PROD/[redacted]
CSS or JS Resource Not Found	4	4	404	/includes/js/date.js	https://[redacted]/display/PROD/[redacted]
CSS or JS Resource Not Found	2	2	404	/includes/js/date.js	http://[redacted]/display/PROD/[redacted]
CSS or JS Resource Not Found	2	2	404	/includes/js/date.js	https://[redacted]/display/[redacted]

Figure 3.22: Analyzing real user traffic reveals missing JavaScript files that were not correctly deployed.

In particular, missing JavaScript files can result in the loss of features available to end users. In *Figure 3.23*, a data.js file was created so that users could select dates from a calendar control instead of entering them manually. Because the file was not deployed correctly, clicking the calendar icon yielded nothing but a log full of JavaScript errors.



```
Message
Script error.
Error loading script
Code: 0: Syntax error
Code: 0: Unable to get value of the property 'onLoad': object is null or undefined
Code: 0: 'TWTR' is undefined
Uncaught TypeError: Cannot call method 'onLoad' of undefined
<unknown>
Uncaught TypeError: Cannot read property 'style' of null
ice is undefined
Uncaught ReferenceError: TWTR is not defined
Code: 0: 'ice' is null or not an object
TWTR is not defined
SyntaxError: missing ) after argument list
ReferenceError: TWTR is not defined
gat is not defined
Uncaught ReferenceError: customizeMacroTable is not defined
    https://[redacted]: Uncaught ReferenceError: customizeMacroTable is not defined
    https://[redacted]: Uncaught ReferenceError: customizeMacroTable is not defined
    https://[redacted]: Uncaught ReferenceError: customizeMacroTable is not defined
    https://[redacted]: Uncaught ReferenceError: customizeMacroTable is not defined
    https://[redacted]: Uncaught ReferenceError: customizeMacroTable is not defined
    https://[redacted]: Uncaught ReferenceError: customizeMacroTable is not defined
    https://[redacted]: Uncaught ReferenceError: customizeMacroTable is not defined
    https://[redacted]: Uncaught ReferenceError: customizeMacroTable is not defined
    https://[redacted]: Uncaught ReferenceError: customizeMacroTable is not defined
```

Figure 3.23: Getting detailed error information from every browser and every end user makes it easy to identify deployment problems that haven't been found in testing.

How to deal with this problem? Many load-testing environments are not configured to reveal this sort of problem, for two basic reasons:

- Traditionally, load tests are executed at the HTTP level only, which will not execute all content.
- Load tests executed in-house do not test the accessibility of third-party components through CDNs.

It is therefore recommended that you take the following steps:

1. Test in a realistic environment, either directly on your production system or within a staging environment that is accessible over the Web.
2. Execute tests that cover all important use cases of your application to verify a correct deployment.
3. Repeat all load tests from geographically dispersed locations. This is the only way to assure that your CDNs are serving all regions and that all of your static content is being correctly deployed to every user.

Some cloud-based testing services offer additional testing services that will execute their scripts using real browser replay instead of HTTP-only testing. Running the tests with various browsers makes sure you test your pages as requested by real end users. The additional benefit of that approach is testing compatibility of browsers and verifying browser-dependent deployment; e.g., special JavaScript Framework versions for certain browsers.

Find more examples on deployment problems on the following blog: http://blog.dynatrace.com/2012/08/07/top-performance-mistakes-when-moving-from-test-to-production-deployment-mistakes/

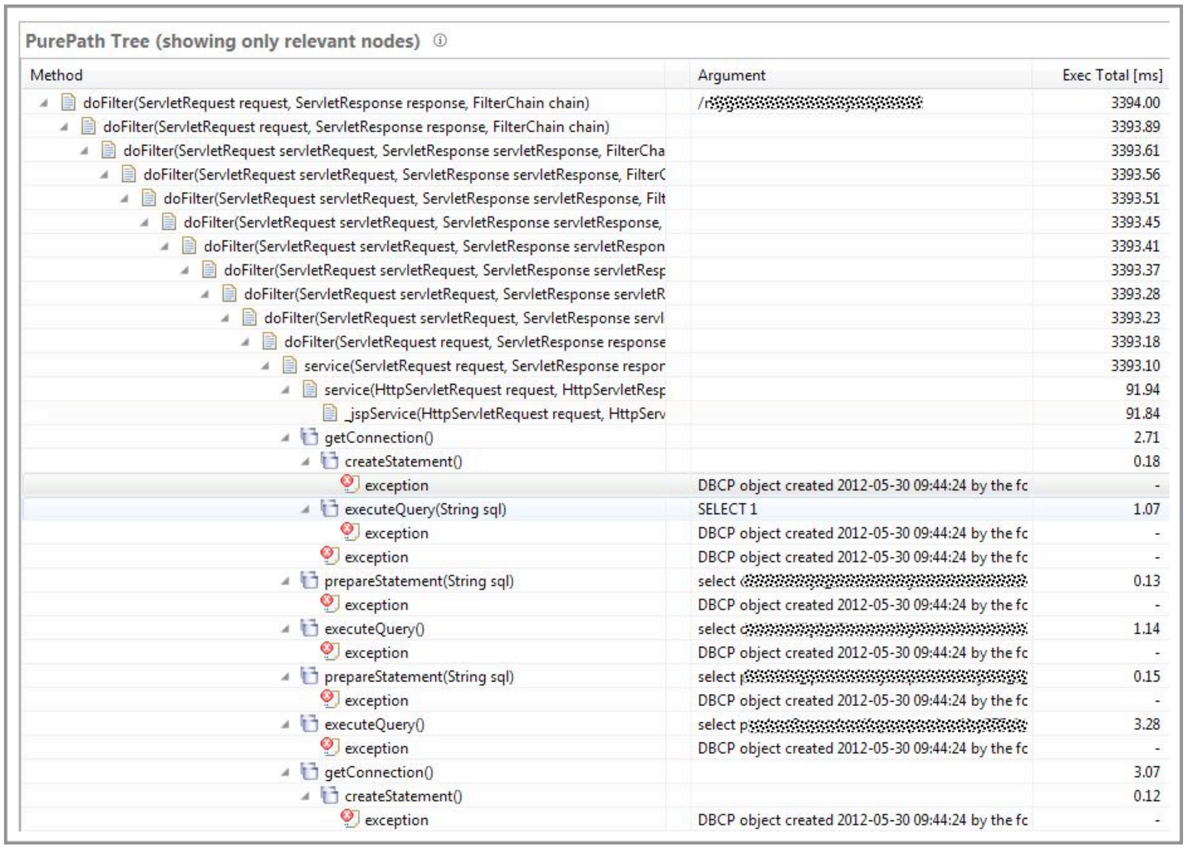
An Overabundance of Log Data

With logging frameworks, it has become easy to generate log entries and then troubleshoot problems with the analytical tools. So easy, in fact, that it's possible to overload your file system with excessive amounts of not-very-useful log data that can impact application performance through no fault of the application itself!

Here is one example, both web applications running on Tomcat, where the log settings create serious problems that show up during testing.

Excessive Use of Exception Objects to Log Stack Trace

In this example, the logAbandoned attribute of the Tomcat Connection Pool was set to true. This triggers a stack trace whenever a database connection is abandoned. However, this application executed several hundred queries for each transaction, and most queries were executed on separate connections. So every log entry resulted in nearly as many getConnection calls as executedQuery calls (*see Figure 3.24*).

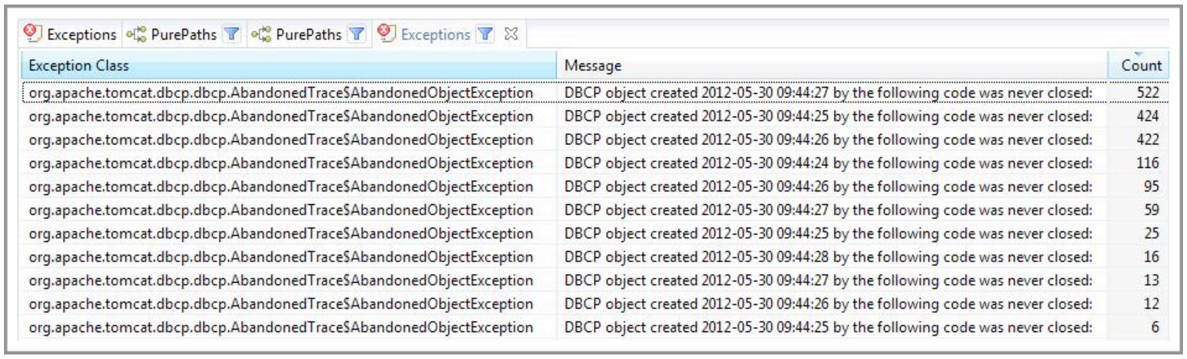


PurePath Tree (showing only relevant nodes)

Method	Argument	Exec Total [ms]
doFilter(ServletRequest request, ServletResponse response, FilterChain chain)	/r[redacted]	3394.00
doFilter(ServletRequest request, ServletResponse response, FilterChain chain)		3393.89
doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterCha		3393.61
doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterC		3393.56
doFilter(ServletRequest servletRequest, ServletResponse servletResponse, Filt		3393.51
doFilter(ServletRequest servletRequest, ServletResponse servletResponse,		3393.45
doFilter(ServletRequest servletRequest, ServletResponse servletRespon		3393.41
doFilter(ServletRequest servletRequest, ServletResponse servletResp		3393.37
doFilter(ServletRequest servletRequest, ServletResponse servletR		3393.28
doFilter(ServletRequest servletRequest, ServletResponse servl		3393.23
doFilter(ServletRequest request, ServletResponse response		3393.18
service(ServletRequest request, ServletResponse respor		3393.10
service(HttpServletRequest request, HttpServletResp		91.94
_jspService(HttpServletRequest request, HttpServ		91.84
getConnection()		2.71
createStatement()		0.18
exception	DBCP object created 2012-05-30 09:44:24 by the fc	-
executeQuery(String sql)	SELECT 1	1.07
exception	DBCP object created 2012-05-30 09:44:24 by the fc	-
exception	DBCP object created 2012-05-30 09:44:24 by the fc	-
prepareStatement(String sql)	select [redacted]	0.13
exception	DBCP object created 2012-05-30 09:44:24 by the fc	-
executeQuery()	select c[redacted]	1.14
exception	DBCP object created 2012-05-30 09:44:24 by the fc	-
prepareStatement(String sql)	select [redacted]	0.15
exception	DBCP object created 2012-05-30 09:44:24 by the fc	-
executeQuery()	select p[redacted]	3.28
exception	DBCP object created 2012-05-30 09:44:24 by the fc	-
getConnection()		3.07
createStatement()		0.12
exception	DBCP object created 2012-05-30 09:44:24 by the fc	-

Figure 3.24: The implicit creation of exception objects in order to log a problem when opening a database connection results in high overhead, as a single transaction creates hundreds of these exceptions.

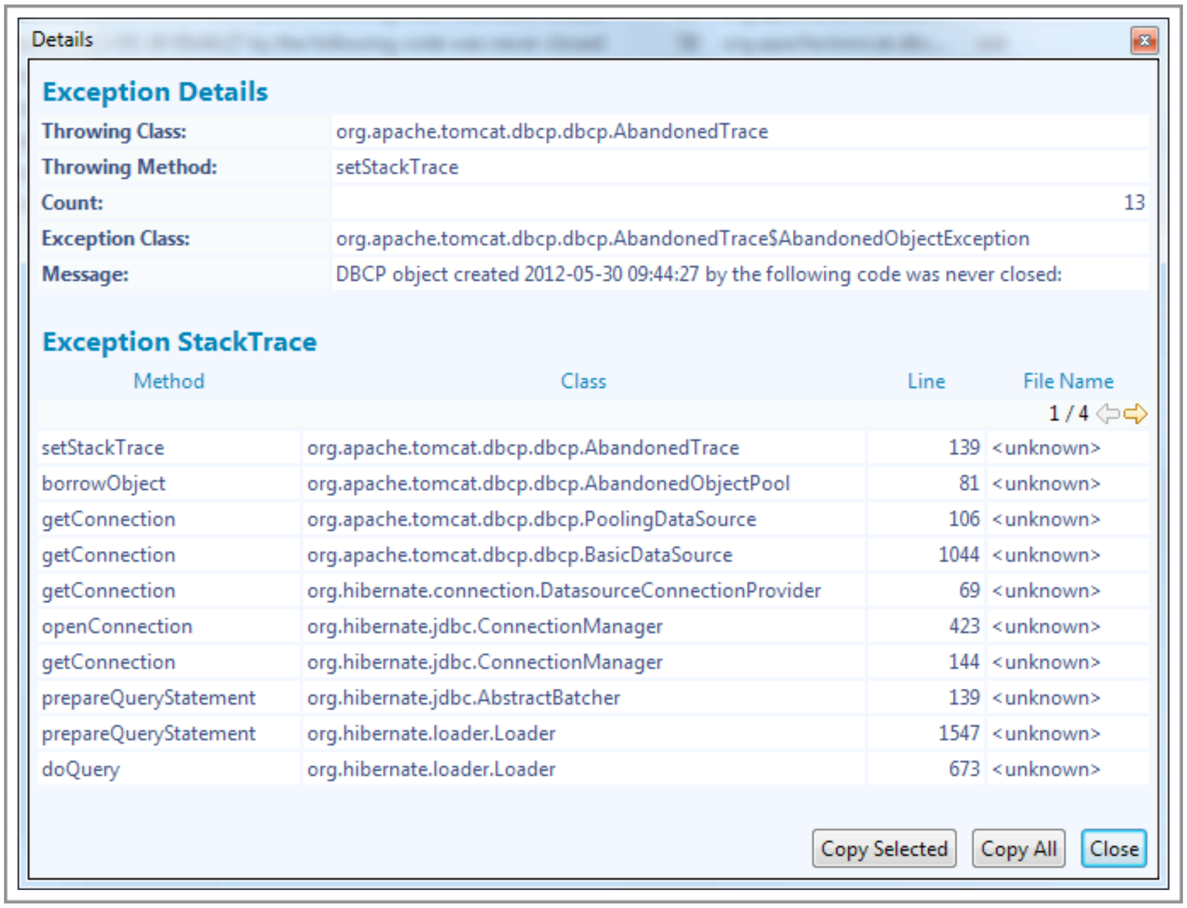
Focusing on the individual exception objects that were created (*Figure 3.25*) shows us more clearly how much redundant information is collected but doesn't provide a whole lot of additional useful information for troubleshooting.



Exception Class	Message	Count
org.apache.tomcat.dbcp.dbcp.AbandonedTrace$AbandonedObjectException	DBCP object created 2012-05-30 09:44:27 by the following code was never closed:	522
org.apache.tomcat.dbcp.dbcp.AbandonedTrace$AbandonedObjectException	DBCP object created 2012-05-30 09:44:25 by the following code was never closed:	424
org.apache.tomcat.dbcp.dbcp.AbandonedTrace$AbandonedObjectException	DBCP object created 2012-05-30 09:44:26 by the following code was never closed:	422
org.apache.tomcat.dbcp.dbcp.AbandonedTrace$AbandonedObjectException	DBCP object created 2012-05-30 09:44:24 by the following code was never closed:	116
org.apache.tomcat.dbcp.dbcp.AbandonedTrace$AbandonedObjectException	DBCP object created 2012-05-30 09:44:26 by the following code was never closed:	95
org.apache.tomcat.dbcp.dbcp.AbandonedTrace$AbandonedObjectException	DBCP object created 2012-05-30 09:44:27 by the following code was never closed:	59
org.apache.tomcat.dbcp.dbcp.AbandonedTrace$AbandonedObjectException	DBCP object created 2012-05-30 09:44:25 by the following code was never closed:	25
org.apache.tomcat.dbcp.dbcp.AbandonedTrace$AbandonedObjectException	DBCP object created 2012-05-30 09:44:28 by the following code was never closed:	16
org.apache.tomcat.dbcp.dbcp.AbandonedTrace$AbandonedObjectException	DBCP object created 2012-05-30 09:44:27 by the following code was never closed:	13
org.apache.tomcat.dbcp.dbcp.AbandonedTrace$AbandonedObjectException	DBCP object created 2012-05-30 09:44:26 by the following code was never closed:	12
org.apache.tomcat.dbcp.dbcp.AbandonedTrace$AbandonedObjectException	DBCP object created 2012-05-30 09:44:25 by the following code was never closed:	6

Figure 3.25: Several hundred exception objects are created just for a single web request. It is all redundant information that doesn't provide additional value.

When looking at the exception details of these individual exception objects the top stack-trace entry reveals which method actually created these objects in order to obtain the stack-trace information for logging (*Figure 3.26*).


Details

Exception Details

Throwing Class:	org.apache.tomcat.dbcp.dbcp.AbandonedTrace
Throwing Method:	setStackTrace
Count:	13
Exception Class:	org.apache.tomcat.dbcp.dbcp.AbandonedTrace$AbandonedObjectException
Message:	DBCP object created 2012-05-30 09:44:27 by the following code was never closed:

Exception StackTrace

1 / 4

Method	Class	Line	File Name
setStackTrace	org.apache.tomcat.dbcp.dbcp.AbandonedTrace	139	<unknown>
borrowObject	org.apache.tomcat.dbcp.dbcp.AbandonedObjectPool	81	<unknown>
getConnection	org.apache.tomcat.dbcp.dbcp.PoolingDataSource	106	<unknown>
getConnection	org.apache.tomcat.dbcp.dbcp.BasicDataSource	1044	<unknown>
getConnection	org.hibernate.connection.DatasourceConnectionProvider	69	<unknown>
openConnection	org.hibernate.jdbc.ConnectionManager	423	<unknown>
getConnection	org.hibernate.jdbc.ConnectionManager	144	<unknown>
prepareQueryStatement	org.hibernate.jdbc.AbstractBatcher	139	<unknown>
prepareQueryStatement	org.hibernate.loader.Loader	1547	<unknown>
doQuery	org.hibernate.loader.Loader	673	<unknown>

Figure 3.26: AbandonedTrace.setStackTrace is the method that creates exception objects and, with that, creates additional overhead.

How to deal with this problem? Load problems that arise during simulation will not necessarily turn up in the deployment system. However, they should serve as a warning to the testing team to make sure that test code is completely removed before moving an application to production.

More information on this particular problem can be found here: https://groups.google.com/forum/?fromgroups#!topic/mmbase-discuss/5x24EjBZMGA

Find more examples of logging on our blog: http://blog.dynatrace.com/2012/08/01/top-performance-mistakes-when-moving-from-test-to-production-excessive-logging/

Load Testing in the Era of Web 2.0

We begin this section with a kind parable for young or inexperienced load testers. When I was learning how to design and execute load tests, an experienced load tester taught me the gospel of test-execution effectiveness:

> *If something doesn't go wrong with some part of the application and/or the infrastructure during the test, then there is most likely something wrong with the test itself.*

Our testing techniques may change as our applications become more sophisticated, but we should remain ever mindful of this rule.

Testing Modern Web Applications

Your company has spent months building a new application and it is ready for testing. You are assigned to lead the testing team. You have enabled enhanced logging and monitoring for all parts of the application stack, the network, the outside perspective, and your third-party providers. Time is scheduled, a conference bridge is opened, and final approval is given. You generate the load and cross your fingers.

Success! The test goes off without a hitch! You are given final approval for release, but you have a nagging worry. Was the test valid? When your site reaches peak load, can you be sure that the customer experience will remain unaffected? More specifically, was the test configured properly to stress the application and all of your third-party content in a way that accurately models customer use? (*See Figure 3.27*.)

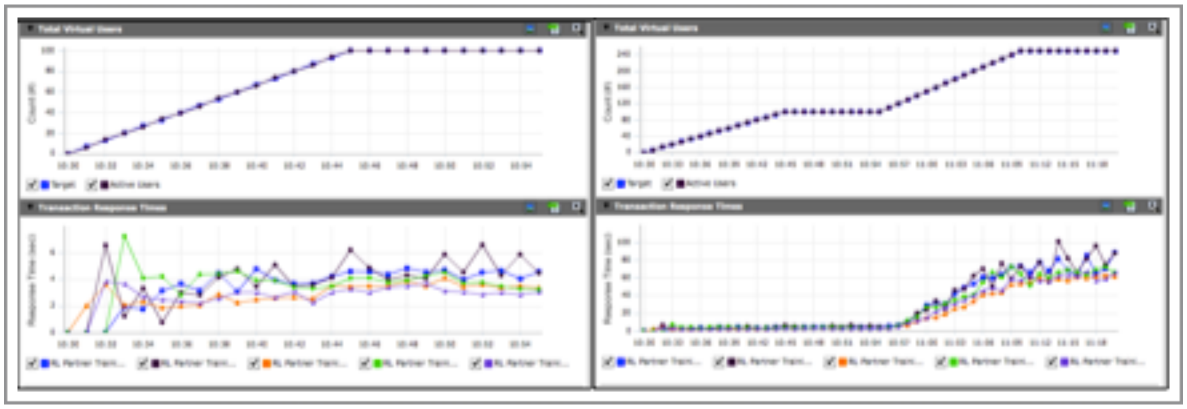


Figure 3.27: Which test is successful: left or right? Only your test design can answer that for you.It is worth noting that the easiest aspect of load testing is the execution. The harder, and far more important, part is designing tests that validate the test criteria. And to do this, we must be able to answer these five essential questions:

- What and why are we testing?
- How do we determine the correct load?
- How do we best simulate customer behavior?
- Who will be involved in the test executions?
- What's next?

Regardless of technology, designing load tests that answer these questions has generated far more meaningful results than any other approach I've tried.

What and Why Are We Testing?

The goal of a load test is to learn something about the complex interactivity and behavior of the application environment under load. But what will be tested will vary greatly. The team could be

- Ensuring that a new datacenter for load-balancing traffic geographically performs as well as the existing datacenter under a full system load in case it has to absorb all traffic in a failure situation
- Validating that the entire end-to-end application can withstand projected holiday-season traffic volumes based on the previous year's analytics
- Gathering data on the integration of a new tag-aggregation and -management system to support the traffic volumes of the post-holiday sale period

Each of these statements seems to only declare what will be tested. But looking closely allows a load-test team to quickly understand the importance of each test scenario. Taking each of the scenarios above one step further shows the why of the testing:

- If we don't validate the ability of a single datacenter to handle our entire load during peak or, at a bare minimum, normal traffic conditions, our company's revenue could be at risk if such a failover ever occurs.
- Last year, we were lucky to survive the load we saw on peak days. The 10% increase that business and marketing projects for this year could cause some serious capacity issues on our two to three busiest days, which are also the highest online-revenue days of the year.
- Last year our development team deployed third-party content to the site without fully testing the potential effects this could have, and we encountered performance degradations on peak days. To prevent this for the current year, we have selected a tag-management solution to help us centralize management and automate deployment of third-party content. We have recently read in industry articles that at the absolute highest volumes, customers using this solution last year saw unusual behavior on their busiest days. We need to validate that the solution is able to handle our projected traffic before it happens in real life.

Each scenario starts with a supposition, theory, or something that the team believes to be true. The goal of the load-test process is to put the system into a simulated scenario that has the potential to prove one or more of the performance beliefs incorrect while providing the data necessary to resolve the issue. Not having the right data after a test may require another round of testing; finding an issue and discovering that the instrumentation in place for the testing was the culprit causes the same issue. But turning on too much logging, and leaving it on by accident after testing is complete, might have a negative effect on application performance. (See Andi Grabner's blog post Top Performance Mistakes When Moving from Test to Production: Excessive Logging.)

How Do We Determine the Correct Load?

With the team knowing what they are testing and why, the next question is how much load to use. When talking to customers, we initially get statements that describe the required load in very vague terms, like these:

- We need to support 55,000 users.
- Our average hourly usage is 200,000 transactions.
- Our system can currently support 2,000 distinct user sessions.
- Customers average 5 minutes per session on our site.

Statements like these do not give us enough information to design a complete load-test traffic profile. Taking the example of a test that requires 55,000 users, we can build out an approach that puts some meat on the bones of the customer's request.

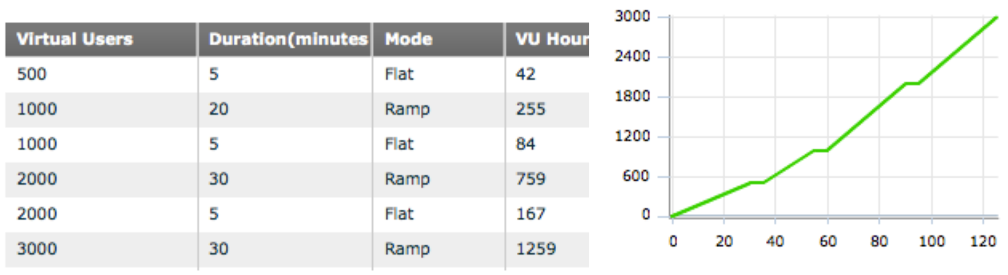
While this statement makes it seem like the testing team will need to spin up 55,000 virtual users to successfully load-test the system, we don't have quite enough information to make that decision yet. Load-test teams should always ask a few more questions to make sure that everyone is on the same page before testing begins.

Where did this number come from? If the application already exists and all that the load test does is test some changes or revisions, then there must be analytics data to support the 55,000-user number. If the application is brand-new but there are similar applications in existence, then the marketing team should be able to extrapolate potential figures from public information—and then share that data with you! If it is a new creation entirely, then some estimating may be required, based on type of launch, marketing, and business expectations.

When does this system see 55,000 users? In an hour? A day? A minute? This question is a follow-up to the first question, as it helps the load-test team determines the time period the application should be tested for. If the application sees 55,000 users over an hour, then a much lower level of virtual users is required for testing than if the application supports 55,000 users every second of every peak hour.

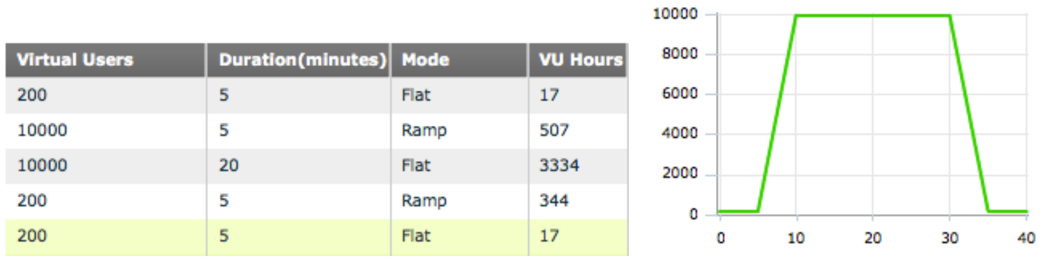
Were these users concurrent? If the site requires you to test a total of 55,000 users over the span of an hour, then you may need far fewer virtual users than if the requirements state that 55,000 users simultaneously or concurrently perform actions on the system over the span of an hour.

Is this a flash crowd or increasing volume? Some applications can encounter flash crowds (Figure 3.24), where usage leaps from very low to well beyond maximum in an incredibly short period of time, usually due to some marketing event—holiday sale, ticket on-sale, Super Bowl advertising, etc. The event may last only 15 minutes, but a failure during this time will cost the company millions in lost revenue, wasted marketing spend, and public brand damage.



Virtual Users	Duration(minutes	Mode	VU Hour
500	5	Flat	42
1000	20	Ramp	255
1000	5	Flat	84
2000	30	Ramp	759
2000	5	Flat	167
3000	30	Ramp	1259

a) Ramped Load



Virtual Users	Duration(minutes)	Mode	VU Hours
200	5	Flat	17
10000	5	Ramp	507
10000	20	Flat	3334
200	5	Ramp	344
200	5	Flat	17

b) Flash Crowd

Figure 3.24: How the load is delivered against a system must reflect the manner in which customers increase over time.

With a few targeted questions, the picture of the customer's requested 55,000-user load test becomes much more clear. Taking this approach provides a map for the load test—the volume of traffic is the destination, but it is critical for everyone involved in the load test to understand how they will get there.

Stefan Karytko adds some additional ramping examples at the Compuware APM Blog: *Web Load Test Ramping Best Practices*: Part 1 (http://apmblog.compuware.com/2011/10/21/web-load-test-ramping-best-practices-1/) and Part 2 (http://apmblog.compuware.com/2012/03/23/web-load-test-ramping-best-practices-part-2/).

How Do We Best Simulate Customer Behavior?

Getting the amount of load right is important. Creating a load that emulates what customers do when they're on the site is just as important as determining how many customers need to be tested. Not all visitors perform the same actions, view the same search results, or buy the same product when they are interacting with the application.

The testing team may test every dark nook and cranny when performing internal testing prior to release. But as the application creeps closer to public release, the test plan needs to become more customer-focused, highlighting the performance of two types of key customer paths:

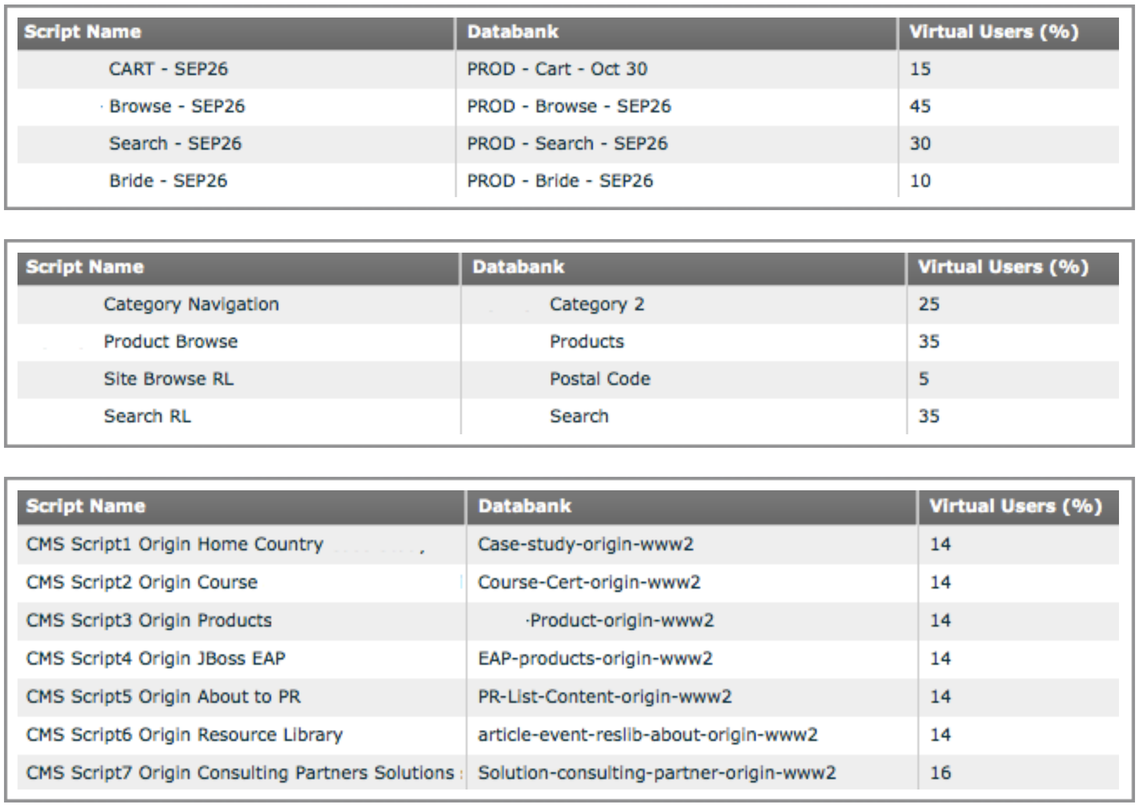
- High-value paths in terms of volume. These are the paths that see the most customer traffic and have the highest exposure to the outside world.

- High-value paths in terms of revenue. These are the customer paths that may not get a high percentage of the total traffic, but that are most important for the company's bottom line.

Using the filter of highest-value transactions, regardless of volume or revenue, most application-testing plans can be boiled down to a relatively small number of key business paths. For a retail application, some the typical paths are as follows:

- Homepage
- Search results to product page
- Browse through random category and product pages
- Add to cart
- Checkout and abandon (guest and registered customer)
- Checkout and pay (when applicable and appropriate)

Additional paths may be required for each individual retailer, but for the majority of testing, these paths cover a substantial percentage of customer traffic and touch on key technical and revenue components that affect the entire business., such as in the example seen in *Figure 3.25.*



Script Name	Databank	Virtual Users (%)
CART - SEP26	PROD - Cart - Oct 30	15
Browse - SEP26	PROD - Browse - SEP26	45
Search - SEP26	PROD - Search - SEP26	30
Bride - SEP26	PROD - Bride - SEP26	10

Script Name	Databank	Virtual Users (%)
Category Navigation	Category 2	25
Product Browse	Products	35
Site Browse RL	Postal Code	5
Search RL	Search	35

Script Name	Databank	Virtual Users (%)
CMS Script1 Origin Home Country	Case-study-origin-www2	14
CMS Script2 Origin Course	Course-Cert-origin-www2	14
CMS Script3 Origin Products	Product-origin-www2	14
CMS Script4 Origin JBoss EAP	EAP-products-origin-www2	14
CMS Script5 Origin About to PR	PR-List-Content-origin-www2	14
CMS Script6 Origin Resource Library	article-event-reslib-about-origin-www2	14
CMS Script7 Origin Consulting Partners Solutions	Solution-consulting-partner-origin-www2	16

Figure 3.25: Designing your business-process percentages helps shape the load test and generate a more realistic user load.

This set of paths touches on the majority of third-party tools: search and catalog indexing systems; advertising; analytics tags; external shopping-cart systems; payment-processing vendors; and, of course, CDNs. For other types of firms—banking, insurance, media, B2B services, etc.—the revenue and traffic filters can also be applied to design a load test to most closely represent the vast majority of customer traffic. Understanding the transactions that are of most value to the organization helps shape the way that the load you designed interacts with the application being tested.

Who Will Be Involved in the Test Executions?

The group that performs the load test is the most critical in the process. When you are running traditional load tests (inside the firewall, with internal application infrastructure and network components), assembling the team is fairly straightforward —all appropriate systems and network teams, with some involvement from vendor teams to support specific hardware or software components if needed.

When the focus moves to outside-in or web load testing, the number of people begins to grow. Now testing teams have to include not just people from the internal application and network teams, but also from connectivity providers, hosting providers, third-party services, and CDNs. This often makes load tests more chaotic, as the amount of data collected begins to balloon in size, with no centralized platform to collect, process, and report on telemetry. While this customer-focused approach delivers new insights that helps companies avoid potential performance disasters, some critical results may not be immediately found in the large and wildly varied datasets collected during a load test.

With organizations integrating a new generation of application-management tools into their systems alongside the outside-in load-testing tools, the number of people on a load-test call may, for the first time in many years, start to go down. These tools integrate metrics across a number of application layers in to a single interface, highlighting specific network, application, and code information more quickly. These systems can also integrate data from external monitoring systems, such as web performance-monitoring services, to further correlate performance issues that appear during load testing to specific points on the end-to-end path of an application.

The key is to identify all of the systems and services your application touches before a load test begins. Even with these new systems, people still need to know that a load test is occurring. If a third-party service begins demonstrating performance issues or an internal system needs to be more deeply analyzed due to a problem identified during the test, key members of the affected teams will need to be brought in to help troubleshoot and resolve the issues.

What's Next?

No matter the approach your team takes, the goal of load testing is to simulate the visitor traffic that is expected under peak conditions as accurately as possible. If there is a problem with the load or with the business-process distribution used during the test, the results could create a misleading picture of how the application and its various components will behave under load.

The rule with load testing is *assume nothing*! Just because the responsible team states that the application, web-server farm, load balancer, database, or firewall should behave in a certain way during the test, doesn't mean it *will* behave that way. It is better to find out that there is a gap between assumption and reality under intense and complex load scenarios before customers find out for you.

And if a test is successful, dig a little deeper. You and your team can be truly secure about a load test only when everyone can answer this question: Was the test actually successful, or did it just validate that one (or more!) of the test configuration parameters was incorrect?

Resources

Steve Bennett. *House MD: Solving Complex IT Issues Using Differential Diagnosis*. http://cdn.oreillystatic.com/en/assets/1/event/79/House%20MD_%20Solving%20Complex%20IT%20Issues%20Using%20Differential%20Diagnosis%20Presentation.pdf