Tricentis

Getting started with performance testing

Tricentis

Table of contents

CHAPTER 1: INTRODUCTION TO PERFORMANCE TESTING

Applications are becoming more and more involved, with shorter development cycles. This requires new, Agile development and testing methodologies. Application performance as part of the global user experience is now the key aspect of application quality.

"Old school," sequential projects with static qualification/implementation/test phases that put off performance testing until the end of the project may face a performance risk. This is no longer acceptable by today's application quality standards.

This white paper provides practical information on how to execute efficient performance testing in this new and more demanding environment.

The state of complexity in modern apps

One of the primary drivers behind this shift to modern load testing is the growing complexity of the IT landscape:

- Most users are using mobile devices, thin clients, tablets, and other devices to reach the information.
- Complex architectures that are shared by several applications at the same time are being built.
- New technologies offer a range of solutions (AJAX framework, RIA, WebSocket, and more) that improve applications' user experience.

Historically, applications have been tested to validate quality in several areas: functional, performance, security, etc. These testing phases answer to user requirements and business risks. However, the dialogue has changed; the conversation is no longer about quality but about user experience. The user experience is a combination of look-and-feel, stability, security, and performance.

Performance: Imperative to a successful user experience

Performance is a key factor in the success of the user experience. This is due to advances in technology, the complexity of the architecture, and the locations and networks of the users. Load testing was a nice addition to the development process, but now it has become an essential testing step.

Load and performance testing answers the following questions:

- Is the application capable of handling a certain number of simultaneous users?
- Is the average response time for pages acceptable under this set load?
- Does the application revert to normal behavior after a load peak?

- How many users can the application handle while maintaining an adequate response time?
- What is the load threshold above which the server(s) begins to generate errors and refuse connections?
- Does the server(s) remain functional under high load, slow down, or crash?

Like any testing activity, performance testing requires proper methods and logic.

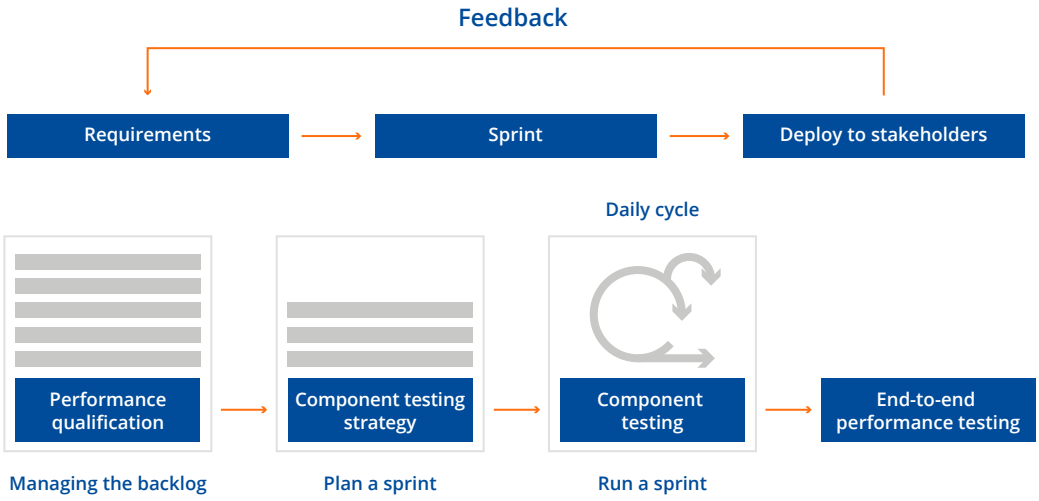
When an application passes performance testing but then fails in production, it is often due to unrealistic testing. In these cases, it is easy but misguided to blame the testing itself or the tools used to execute it. The real problem is usually test design without the correct basis. It is necessary to ask, "What did we need to know that, if we had known it, would have allowed us to predict this failure before production?" In other words: "How can we deliver efficient performance testing?"

Phases of a load testing project

Current development methodologies such as Agile and DevOps allow for the creation of applications that quickly answer to customers' needs. These methodologies involve updating the project organization and require close collaboration between teams.

In these methodologies, the project life cycle is organized into several sprints, with each sprint delivering a part of the application.

In this environment, the performance testing process should follow the workflow below.



A performance testing strategy should be implemented at an early stage of the project life cycle. The first step: performance qualification. This defines the testing effort of the whole project.

An "old school" approach to performance testing would force the project to wait for an assembled application before performance validation would begin. In a modern project life cycle, the only way to include performance validation in an early stage is to test individual components after each build and implement end-to-end performance testing once the application is assembled.

CHAPTER 2: ESTABLISHING A PERFORMANCE TESTING STRATEGY

This is the first and most crucial step of performance testing. It defines:

- The performance testing scope
- The load policy
- The SLA (service level agreements) and service level objectives (SLOs)

It is never possible to test everything, so conscious decisions about where to focus the depth and intensity of testing must be made. Typically, the most fruitful 10% to 15% of test scenarios uncover 75% to 90% of significant problems.

Risk-based testing

Risk assessment provides a mechanism with which to prioritize the test effort. It helps to determine where to direct the most intense and deepest test efforts and where to deliberately test lightly, to conserve resources for intense testing areas.

Risk-based testing can identify significant problems more quickly and earlier on in the process by testing only the riskiest aspects of a system.

Most system performance and robustness problems occur in these areas:

- Resource-intensive features
- Timing-critical or timing-sensitive uses
- Likely bottlenecks (based on the internal architecture and implementation)
- Customer or user impact, including visibility
- Prior defect history (observations of other similar systems in live operation)

- New and modified features and functionality
- Heavy demand: heavily used features
- Complex features
- Exceptions
- Troublesome (poorly built or maintained) portions of the system
- Platform maintenance

Here is a list of questions presented by industry-expert Ross Collard to identify the different performance risks:

Situational view

- Which areas of the system operation, if they have inadequate performance, most impact the bottom line (revenue and profits)?
- Which uses of the system are likely to consume a high level of system resources per event, regardless of how frequently the event occurs? The resource consumption should be significant for each event, not high in aggregate simply because the event happens frequently and thus the total number of events is high.
- What areas of the system can be minimally tested for performance without imprudently increasing risk, to conserve the test resources for the areas which need heavy testing?

Systems view

- Which system uses are timing-critical or timing-sensitive?
- Which uses are most popular (e.g., happen frequently)?
- Which uses are most conspicuous (e.g., have high visibility)?
- What circumstances are likely to cause a heavy demand on the system from external users (e.g., remote visitors to a public website who are not internal employees)?
- Are there any notably complex functions in the system — for example, in the area of exception handling?
- Are there any areas in which new and immature technologies have been used, or unknown and untried methodologies?
- Are there any other background applications that share the same infrastructure, and are they expected to interfere or compete significantly for system resources (e.g., shared servers)?

Intuition/experience

- What can we learn from the behavior of the existing systems that are being replaced, such as their workloads and performance characteristics? How can we apply this information to testing the new system?
- What has been your prior experience with other similar situations? Which features, design styles, subsystems, components, or system aspects typically have encountered performance problems? If you have no experience with other similar systems, skip this question.
- What combinations of the factors you identified by answering the previous questions deserve a high testing priority? What activities are likely to happen concurrently and cause heavy load and stress on the system?
- Based on your understanding of the system architecture and support infrastructure, where are the likely bottlenecks?

Requirements view

- Under what circumstances is heavy internal demand likely (e.g., by the internal employees of a website)?
- What is the database archive policy? What is the ratio of data added/year?
- Does the system need to be available for 7 hours, 24 hours, etc.?
- Are there maintenance tasks running during business hours?

The answers to these questions will help identify:

- Areas that need to be tested
- The type of tests required to validate the performance of the application

Component testing

Once the functional areas required for performance testing have been identified, decompose business steps into technical workflows that showcase technical components.

Why should business actions be split into components? Since the goal is to test the performance at an early stage, listing all important components will help to define a performance testing automation strategy. Once a component has been coded, it makes sense to test it separately and measure:

- The response time
- The maximum calls that the component can handle

Moreover, component testing supports JMS, API, service, messages, etc., allowing scenarios to be easily created and maintained. Another major advantage of this strategy is that the components' interfaces are less likely to be impacted by technical updates. Once a component scenario is created, it can be included in the build process, and feedback on the performance of the current build can be received.

After each sprint, it is necessary to test the assembled application by running realistic user tests (involving several components). Even if the components have been tested, it is mandatory to measure:

- The behavior of the system with several business processes running in parallel
- The real user response time
- The availability of the architecture
- The sizing of the architecture
- Caching policy

The testing effort becomes more complex with the progression of the project timeline. In the beginning, the focus is on the quality of applications and then concentrated on the target environment, architecture, and network. This means that performance testing objectives will vary depending on the timeline of the project.

Test environment

It is imperative that the system under test be properly configured and that the results obtained can be used for the production system. Environment and setup-related considerations should remain top-of-mind during test strategy development. Here are a few:

- What data is being used? Is it real production data, artificially generated data, or just a few random records? Does the volume of data match the volume forecasted for production? If not, what is the difference?
- How are users defined? Are accounts set with the proper security rights for each virtual user, or will a single administrator ID be reused?
- What are the differences between the production and the test environment? If the test system is just a subset of production, can the entire load or just a portion of that load be simulated?

It is important that the test environment mirror the production environment as closely as possible, but some differences may remain. Even if tests are executed against the production environment with the actual production data, it would represent only one point in time. Other conditions and factors would also need to be considered.

Devise a test plan

The test plan is a document describing the performance strategy. The test plan should include:

- Performance risk assessments highlighting the performance requirements
- Performance modeling: explaining the logic to calculate the different load policies
- The translation of the main user journey into components
- The description of the different user journeys with the specific think time per business transaction
- The dataset(s)
- The SLA
- The description of each test that needs to be executed to validate the performance
- The testing environments

The test plan is a key artifact of a well-designed and -executed performance testing strategy, acting as evidence that a team has satisfactorily considered the critical role performance plays in the final end-user experience.

In many cases, project teams ensure the delivery of performance test plans as part of feature work during planning and development cycles by requiring them as part of their "definition of ready." Though each feature story or use case may not require the same level of performance testing, making the thought process a hard requirement for completion leads to better systems and an improved mindset over the end-to-end quality of what the team delivers.

CHAPTER 3: MODELING PERFORMANCE TESTS

The objective of load testing is to simulate realistic user activity on the application. If a non-representative user journey is being selected or if the right load policy is not being defined, the behavior of the application under load will not be able to be properly validated.

Performance test modeling doesn't require any technical skills, only time to fully understand the application:

- How do users work on the system?
- What are their habits?
- When and how often are they using the application? And from where?
- Is there any relationship between external events and activity peaks?
- Is the company business plan related to the activity of the application?
- Is the user community going to expand in different geographies?
- Is there a marketing plan to market/promote the application? If yes, who is the audience?
- Are there any layers of architecture shared with another system?

To fully understand the application during performance modeling, the following roles should be involved:

- Functional architect
- Business analysts
- Project leader

Of course, the different roles will provide different feedback, but the idea is to understand the application, the end users' habits, and the relationship of the application with the current or future organization.

System logs or database extractions are also very useful. These would easily point out the main components and functional areas currently used in production. They would also retrieve the number of transactions/hour per business process.

- When considering the user load, it is important to be aware of the difference between the concurrent number of users and the number of users within the expected audience. An application delivered to 5000 users may have only 500 users accessing it concurrently.
- If there isn't any estimation on the maximum concurrent users, then the number of users can be calculated based on the number of transactions/hour per business actions.

Establish service level agreements (SLAs) and service level objectives (SLOs)

Service level agreements and service level objectives are the keys to automation and performance validation. The project leader and functional architect need to define ideal response times for the application. There are no standard response times.

An SLA/SLO will allow the performance engineer to give a status on the performance testing results. With the SLA/SLO, performance testing can be easily automated to identify performance regression between several releases of the application.

If the importance of component testing and of including performance testing in an early stage of the project is properly understood, the SLA or SLO must be defined at the component level for the tests to be properly automated.

Test selection

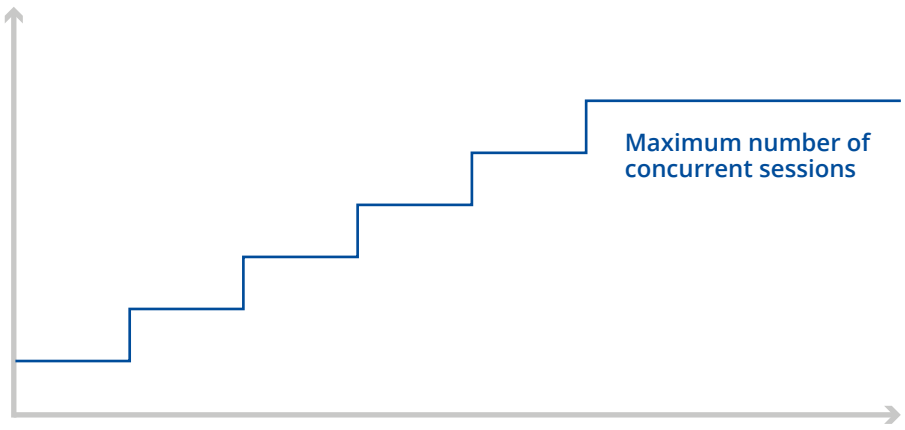
Each test that is run will address one risk. Each test will give a status on the performance requirements.

Of course, most projects focus only on reaching the limit of the application. Reaching the application's limit is important to validate the sizing and the configuration of the architecture, but it most likely won't answer to all of the performance requirements.

There are other tests that need to run to properly qualify the performance of the application. These tests will help validate performance requirements:

- **Unit test:** Many projects overlook the power of unit testing. Don't start running load tests if the performance is not acceptable with one single user. Unit testing allows the application to be instrumented with deep-dive analytics tools and provides useful information to the developers.

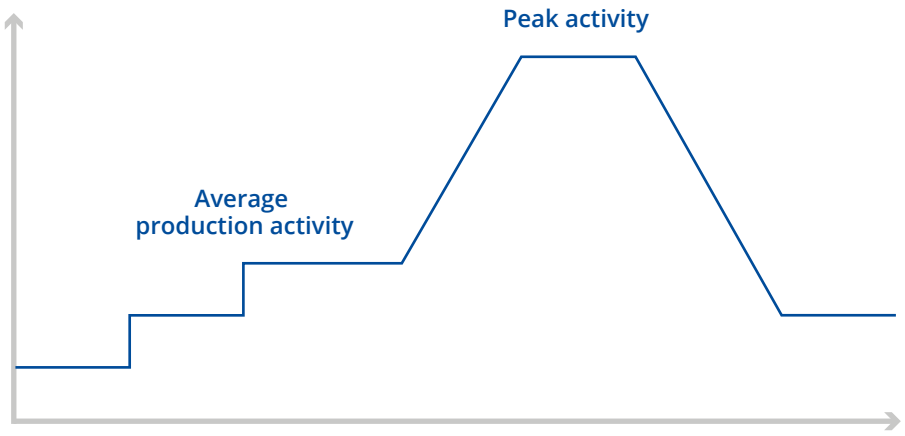
  This step should be a mandatory part of any load testing strategy.



- **Connection test:** Let's say that each morning, all of the application users connect to the system around the same time and then grab a coffee or chat with their colleagues. Even if there are no major activities on the business components of the application, the architecture has to handle this very important peak of user sessions. It should be obvious, then, to ensure that the system won't break every morning. The operations team's lives will be made much easier by validating this point. This test will ramp up all the expected users.

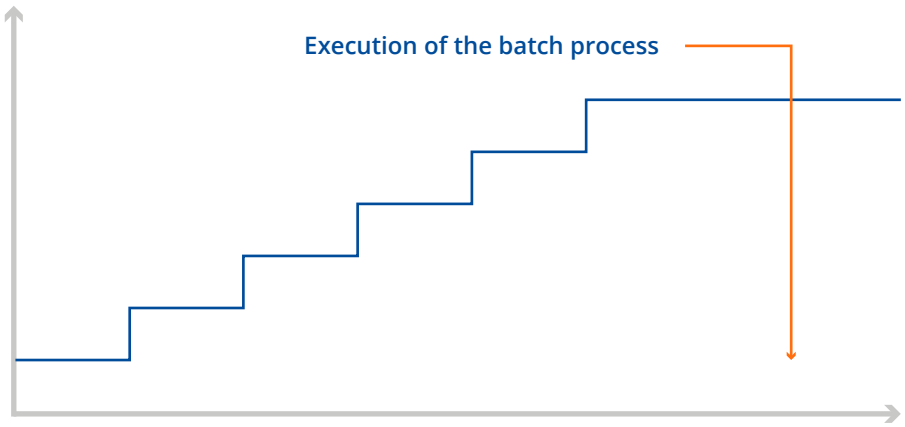
- **Production activity test:** Every application has major business actions. These actions often result in data being updated or inserted into the database. Therefore, there should be a good understanding of the number of records created per day, or per hour. The production activity test ensures that the system can handle the load related to the number of transactions/hour expected in production. This test will load and validate the behavior of the business components within the database. Two main requirements for this type of test are the number of transactions/hour per user journey and the appropriate think times.

- **Peak test:** Every application experiences peak volumes. Even if those peaks appear only once or twice each year, it is mandatory to ensure that the application can handle them. The purpose of a peak test is to simulate an important increase of users during a short period (e.g., a retail site may need to simulate the activity that would come as a result of a big sale on extremely popular products).



- **Soak test:** The application/architecture will most likely be available for several days or even weeks on end. If there are not dedicated times set for maintenance tasks, then it is important that the architecture will run without failure over a long period. A soak test will run a constant number of users over an extended duration. With this test, it will be easy to identify any memory leak or network conjunction and monitor the stability of the environment's current configuration.

- **Batch test:** Some applications are designed with asynchronous tasks or batches. What is the impact of a batch on the real users? Batch testing will run the production activity and trigger the execution of a batch at a specific moment of the test.



The purpose of this test is to identify if the batch has any impact on the user experience.

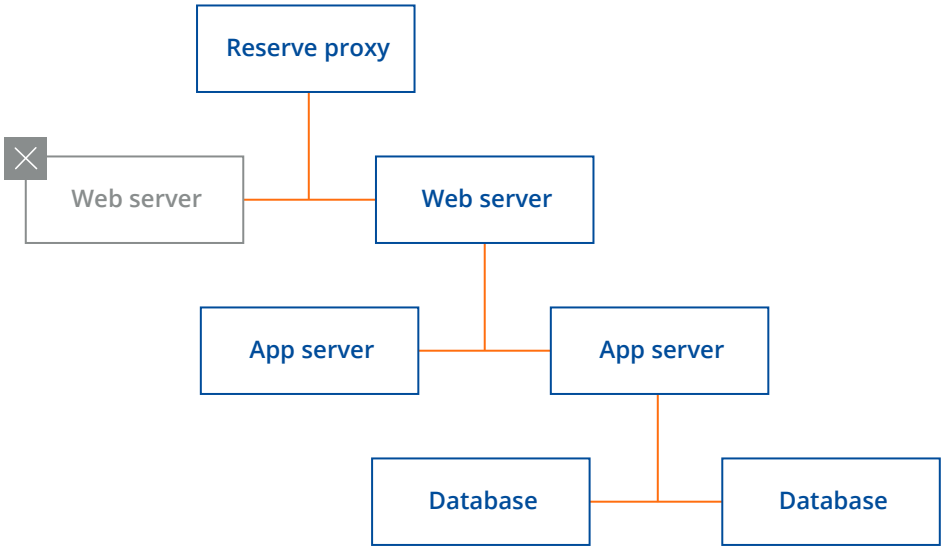
Some other load tests could be utilized depending on the constraints of the business and the locations of the users. There are always events related to the company or organization that will affect the load of the application.

For example, the load designed for a trading application will depend on the opening hours of different markets. The application will have three typical phases: the Asian users, then the European users combined with a part of the Asian users, and then the U.S. users combined with the European users. Every time the markets open or close, there are peaks of trading activity. Therefore, testing would include different types of load combining different combinations of business cases.

On the other hand, there are tests that are designed to validate the availability of the platform during maintenance tasks or a production incident:

- **Failover test:** The purpose of this test is to simulate a production failure on the environment. This kind of test is mandatory to validate the availability of the environment and to make sure the failover cluster mechanism reacts properly. When the architecture has N nodes, it is important to validate that the N-1 or N-2 nodes can handle the expected load so as not to experience event cascade during a production problem. The operations team is also interested in whether or not they can complete their maintenance tasks without having to set the application into a maintenance state. Most high-availability applications have many nodes on different layers of the architecture: web servers, application servers, database, etc. There should be one test per layer.

- **Recovery test:** This test is, in fact, the opposite of the degradation test. The test is initiated by stopping one of the nodes and then restarting the node while the application is loaded. The purpose of this test is to see how the node reacts to a heavy load just after being restarted.

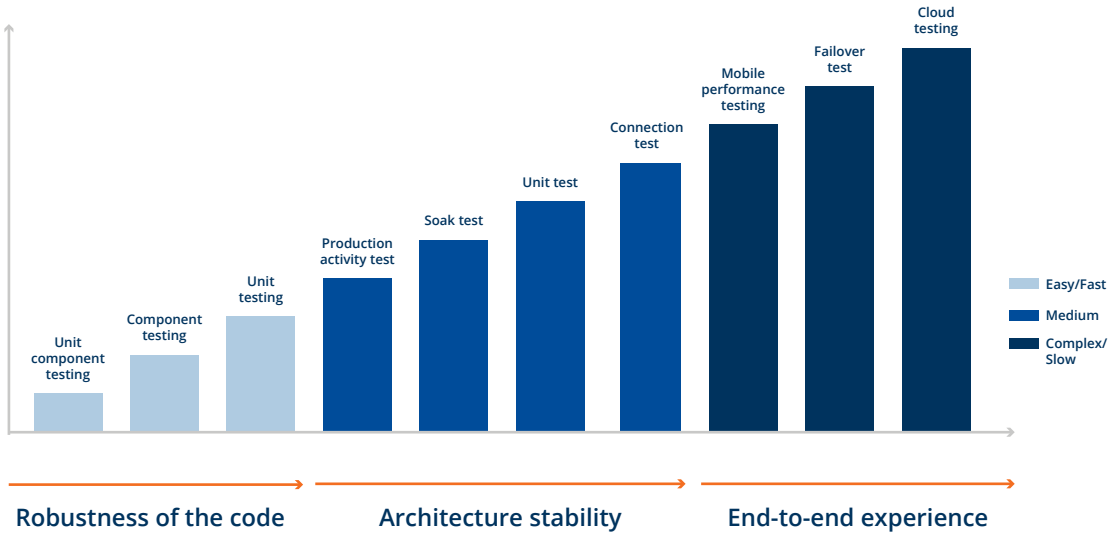


Additionally, there may be operational situations to include during the load test to measure application behavior when the cache is being cleaned up.

There is yet another point that could affect the performance of the application: the data. Often when load testing, a testing database with less data is used, or at least a database that has been anonymized for security purposes. The anonymous process usually creates records starting with the same letters or digits. So all of the indexes used during the load test will be incomparable to those of the normal production database.

Data grows quite fast in production. The behavior of the database is quite different depending on the size of the database. If the lifetime of the database is long, then it might make sense to validate the performance of different-sized databases to see if the limit is different between a light and heavy database. To achieve this kind of test, a large dataset that points to various areas of the database should be used; never use a list of names where all the accounts start with AA... AAA2... but instead, have a representative set of data pointing from A to Z.

The type and complexity of the test will change during the project life cycle:



Include think times

- An important element of performance design is “think time”
- Think time is the time needed for a real user between two business actions:
  – Time to read information displayed on the screen
  – Time to fill out information in a form
- Other real-user actions that do not cause interaction with the application servers

Because every real-world user behaves differently, think times will always be different. Therefore, it is important:

- To gather the think time of each business step in each scenario. Avoid using the same think time for each business step. There are always screens displaying information that end users need to read or forms they need to fill out before going to the next step of the business process. There should be a specific think time for each action
- To calculate the average minimum and maximum think times per action
- To let the load testing tool randomly choose a think time from a specified range (min and max)

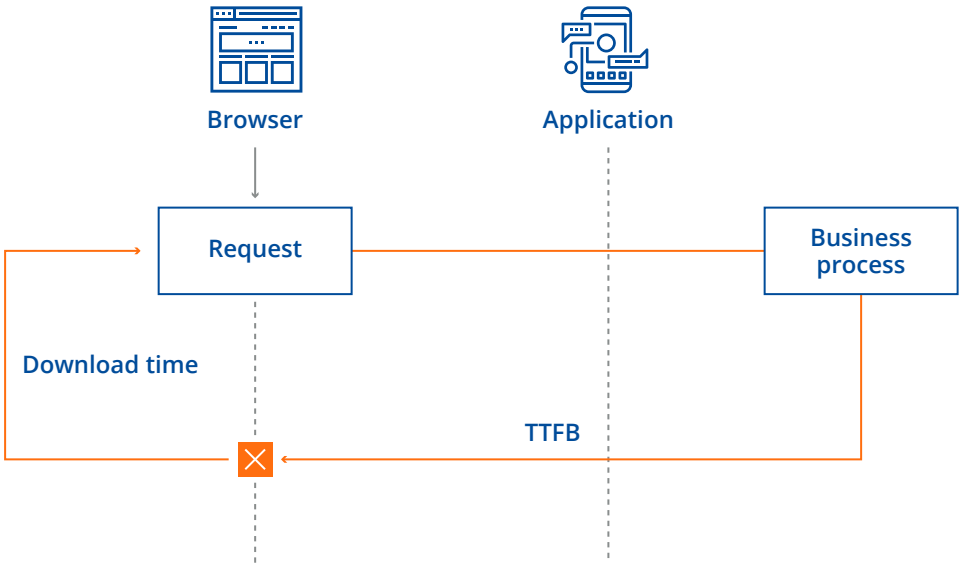
Imagine the application is a castle with big walls and big doors. The components that will be validated (via load test) are inside that castle. The wall will represent proxy servers, load balancers, caching layers, firewalls, etc.

If a test is run without including think times, then only the main door would be hit. Maybe it would break, but there is a big chance that it would lock everything out after a few minutes. Be gentle and smart, and hide from the defense. The main door will allow entrance, and the components located inside of the castle will be able to be properly load tested.

Validate the user experience

As mentioned earlier, once the application is assembled, testing objectives will change. At some point, the quality of the user experience needs to be validated.

Mobile applications, rich Internet applications (RIA), and complex AJAX frameworks are challenging in the way most are used to measure response times. In the past, measurements have been limited to download time and time to first buffer (TTFB).



This approach does not make sense because much of the rendering time that contributes to user experience depends on local ActiveX/JavaScript or native application logic. As such, testing measurements cannot be limited to TTFB and download time because the main objective here is to validate the global user experience on the application.

Measuring the user experience is possible by combining two solutions: load testing software (e.g., Tricentis NeoLoad) and a browser-based or mobile testing tool.

The load testing tool will generate 98% of the load on the application. The browser-based or mobile-based testing tool will generate the other 2% of the load to retrieve the real user experience (including the rendering time) while the application is loaded.

This means that the business processes and transactions to be monitored by the browser/mobile- based solutions need to be carefully identified.

Monitoring

Running tests without monitoring is like watching a horror movie on the radio. You will hear people scream without knowing why. Monitoring is the only way to get metrics related to the behavior of the architecture.

However, many projects tend to lack performance monitoring due to:

- The lack of tools
- The fear of the requirements needed to enable monitoring

Though monitoring is not limited to the operating system of the different server(s), its purpose is to validate that each layer of the architecture is available and stable. Architects took time to build the smartest architecture, so it is necessary to measure the behavior of the different layers.

Monitoring allows for a more comprehensive understanding of the architecture and the investigation into the behavior of the various pieces of the environment:

- **Operating system:** CPU, memory, disk, and network utilization
- **Application server:** memory utilization, garbage collector, thread utilization, sessions
- **Web server:** worker, number of sessions
- **Database:** buffer pool, cache utilization, number of transactions, number of commits, % indexed queries
- **Caching server:** hit ratio

Many projects use production monitoring tools to retrieve metrics from the architecture. This approach is not recommended since production monitoring has a large granularity between each data point (every 2-5 minutes). In load testing, it is important to have monitored data collected at least every five seconds. The performance engineer needs to be given every opportunity to identify bottlenecks. When a peak appears for only a few seconds during a test, it is vital to have enough granularity to point out the bottleneck.

Monitoring requires technical requirements such as a system account, which services to be started, firewall ports to be opened, etc. Even if it seems difficult to meet those requirements, monitoring is possible if there is proper communication with operations; sending them these requirements in advance will facilitate that communication.

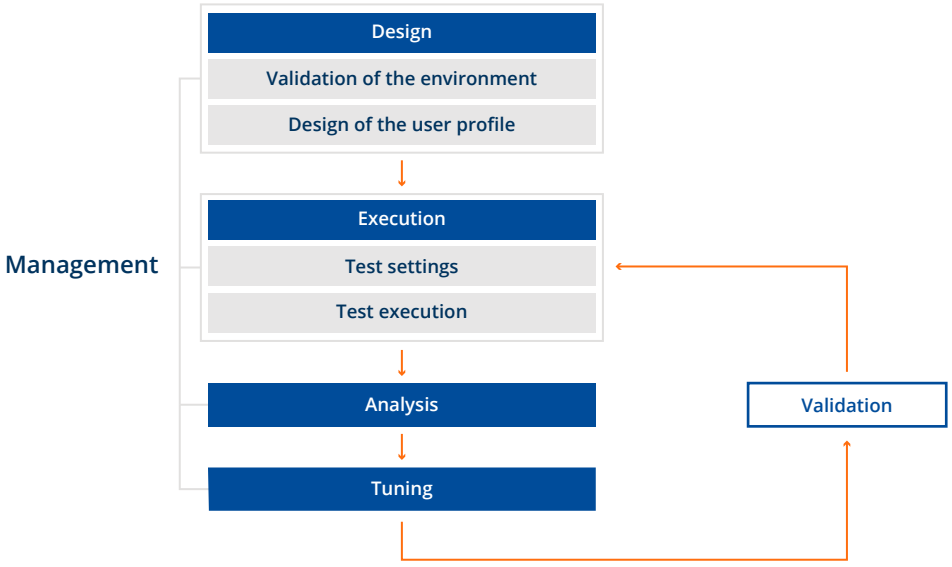
A key takeaway for monitoring: anticipate requirements at an early stage.

CHAPTER 4: EXECUTING A PERFORMANCE TEST

Each performance testing activity is mapped out in the following workflow:

Performance testing objectives will need to adapt depending on results. The performance engineer will analyze these results between testing and tuning. Ultimately, the main objectives of a performance test are to identify bottlenecks and suggest solutions.

Below are the three key steps in executing an effective performance test.



Design

The design phase is often known as the "scripting" phase, but new technology has made the design phase more complex.

Creating performance testing scripts is, in essence, a software development project. Sometimes automatic script generation from a recording is mistakenly interpreted as the whole process of script creation, but it is only the beginning. Only in very simple cases will automatic generation provide completed scripts; in most non-trivial cases, it is just a first step. Scripts need to be correlated (get dynamic variables from the server) and parameterized (use different data for different users). These operations are prone to errors because changes are made directly to the communication stream. Any mistakes at this point can be very dangerous because these types of mistakes usually cannot happen in the real world where the users interact with the system through a user interface or API calls.

AJAX applications, WebSocket, and polling technologies generate regular requests. These requests are not generated by the interaction of the user within the graphical user interface. Instead, many calls are generated by one or more individual page components. For example, on an eCommerce website, there is always a direct link to the user cart from the pages. This link needs to show the user the exact number of products currently in his or her basket. Most of the time, the number of products is displayed via a polling request (a technical call done every five seconds).

This type of call needs to be understood by the performance engineer. All "polling" mechanisms would need to be removed from the business step of the user journey and placed into a dedicated thread.

Another good example is adaptive streaming technology. Many testers often rely on the record/ playback approach. Unfortunately, this type of logic won't benefit any testing strategy.

The goal is to qualify the behavior of the application under realistic load, not to validate the caching.

Replaying back the recording request with no correlation will only call the cache of the application:

- Web cache
- Application cache

Another important element of the design step is the dataset. Using a small set of data will generate the same query on the database. As previously mentioned, the point of a load test is not to qualify the efficiency of the database cache or simply generate deadlocks.

After the script is created, it should be evaluated for a single user, multiple users, and with different data. Do not assume that the system works correctly when a script is executed without errors. Instead, ensure that the applied workload is doing what it is supposed to do and that all errors are caught and logged. It can be done directly by analyzing server responses or, in cases when this is impossible, indirectly. It can be done, for example, by analyzing the application log or database for the existence of particular entries.

Many tools provide some way to verify workload and check errors, but a complete understanding of exactly what's happening is necessary. NeoLoad allows the user to create assertions to make sure that the response of the server is "normal," (e.g., the content is as expected). For example, receiving "out of memory" errors in a browser page instead of the requested reports would qualify as unexpected content. It is important to catch such content errors with the help of assertions.

NeoLoad and many load testing solutions will automatically capture HTTP errors for Web scripts (e.g., 500 "Internal Server Error"). If only the default diagnostics are relied upon, it can't be verified that the business transactions are doing what is expected.

New web design tends to avoid HTTP errors and displays exception errors in the applications. So if checking is limited to examining only the HTTP codes, then it can't be determined if the scenario is achieving the expected actions on the application.

Execution

When executing the actual test, take these challenges into consideration:

- Load testing architecture
- Effective test monitoring

Load testing architecture

The main objective is to load test the application, not the load testing software. Most load testing solutions have several components:

- **Controller:** Orchestrates the test and stores the results data — response time, hit/s, throughput, monitoring metrics, errors
- **Load generator:** Runs the load testing script against the application. This component will need to handle several hundreds of concurrent virtual users.

To avoid being limited by the load testing architecture, there need to be enough:

- Load generators to achieve the expected load
- Network bandwidth between the load generator and the application. Sizing load generators is a crucial step in defining the number of virtual users that a load generator can handle. This number of virtual users depends on many aspects:
  - Technologies used in the application to test
  - The complexity of the application to test
  - Network connection capacity

Starting a "ramp-up" (or "scalability") test of the application with one single load generator will help determine the maximum capacity of one machine. After determining the number of virtual users that one load generator can handle and the number of virtual users for the test, it is easy to calculate the number of generators necessary to run the test. To pinpoint the load limits of a load generator, it is necessary to watch its behavior under the ramp-up test load, and specifically:

- CPU
- Memory
- Throughput
- Hit rate
- Virtual users load

The first significant breaking point in the CPU, memory, throughput, or hit rate metrics represents the performance limit of the load generator. This point must be correlated with the number of virtual users generated. Serious issues may occur with the load generators if pushed beyond this number of virtual users.

A 20% to 30% security margin out of the sizing limits is recommended for the load generators in a heavy load test.

It is also important to avoid using the controller machine as a load generator. The controller is the heart of the test. It is always better to lose a load generator than a controller (the whole test result).

Effective test monitoring

As mentioned earlier, there is limited time to run different tests and proceed to monitoring. Time shouldn't be wasted by looking at the test data while the test is running.

If response time starts increasing due to web server saturation, stop the test and start tuning the web server. Almost every time testing is started on a new application and new environment, most of the layers (application server, web server, etc.) are not configured or tuned for the load of the application. So every test on a representative environment needs attention from the performance engineer to properly tune the environment.

Every new test utilizing a new dataset also needs attention. For example, stop the test if every test user is generating errors when it logs into the application.

On the other hand, if there is a good understanding of the application and the environment, test automation can be started without looking at the behavior of the application. Once the test is finished, analyze the results.

Analysis

Analyzing load testing results is a job in itself. Comprehensive knowledge of the following is required:

- The load testing design
- The technical layers involved in the application
- Modern architecture

This white paper doesn't explain how to analyze results, but here are some recommendations for reporting on the project's performance testing results.

Almost all load testing solutions allow for the creation of complex graphs that correlate data. The performance engineer's first inclination will be to show all of the graphs in the report.

Before creating this report, however, it is important to understand the role and the technical skills of the person validating or simply reading the report.

Based on these qualities, several different types of reports can be created. For example:

- Technical report for developers and operations showing only important graphs
- Decision-maker report giving a very simple status of the application's performance

The main purpose of the performance testing report is to give a clear status on the performance of the application. Results reports should be simplified, focusing on these three main application themes:

- Response times
- Availability
- Scalability

A graphical presentation (three pie charts) makes the results easier to understand for the decision-makers.

Ultimately, the performance report needs to highlight if the performance requirements (identified during the performance strategy phase) are validated.

REFERENCES

Determining The Test Focus Through Risk Assessment, Ross Collard

ABOUT TRICENTIS

Tricentis is the global leader in enterprise continuous testing, widely credited for reinventing software testing and delivery for DevOps and agile environments. The Tricentis AI-based, continuous testing platform provides automated testing and real-time business risk insight across your DevOps pipeline. This enables enterprises to accelerate their digital transformation by dramatically increasing software release speed, reducing costs, and improving software quality. Tricentis has been widely recognized as the leader by all major industry analysts, including being named the leader in Gartner's Magic Quadrant five years in a row. Tricentis has more than 1,800 customers, including the largest brands in the world, such as Accenture, Coca-Cola, Nationwide Insurance, Allianz, Telstra, Dolby, RBS, and Zappos.

To learn more, visit www.tricentis.com or follow us on LinkedIn, Twitter, and Facebook.

AMERICAS

2570 W El Camino Real,
Suite 540
Mountain View, CA 94040
United States of America
office@tricentis.com
+1-650-383-8329

EMEA

Leonard-Bernstein-Straße 10
1220 Vienna
Austria
office@tricentis.com
+43 1 263 24 09 – 0

APAC

2-12 Foveaux Street
Surry Hills NSW 2010,
Australia
frontdesk.apac@tricentis.com
+61 2 8458 0766

v. 0821