International Software Testing Qualifications Board

Certified Tester

Foundation Level Specialist Syllabus Performance Testing

Version 2018

Provided by

American Software Testing Qualifications Board

and

German Testing Board

Performance Testing Working Group:

Graham Bath
Rex Black
Alexander Podelko
Andrew Pollner
Randy Rice

Revision History

Version	Date	Remarks
Alpha V04	13 December 2016	Version for NYC Meeting
Alpha V05	18 December 2016	After NYC Meeting
Alpha V06	23 December 2016	Restructure Ch 4 Renumbering and adjusting LOs as agreed in NYC
Alpha V07	31 December 2016	Add author comments
Alpha V08	12 February 2017	Pre-Alpha version
Alpha V09	16 April 2017	Pre-Alpha version
Alpha Review V10	28 June 2017	For the Alpha Review
V2017v1	27 November 2017	For Alpha Review
V2017v2	15 December 2017	Alpha updates
V2017v3	15 January 2018	Technical edit
V2017v4	23 January 2018	Glossary review
V2018 b1	1 March 2018	Beta Candidate for ISTQB
V2018 b2	17 May 2018	Beta Release for ISTQB
V2018 b3	25th August 2018	Beta review comments incorporated for release version
Version 2018	9 December 2018	ISTQB GA Release

Table of Contents

Acknowledgements

This document was produced by the American Software Testing Qualifications Board (ASTQB) and the German Testing Board (GTB):

Graham Bath (GTB, Working Group co-chair)
Rex Black
Alexander Podelko (CMG)
Andrew Pollner (ASTQB, Working Group co-chair)
Randy Rice

The core team thanks the review team for their suggestions and input. ASTQB would like to acknowledge and thank the Computer Measurement Group (CMG) for their contributions in the development of this syllabus.

The following persons participated in the reviewing, commenting or balloting of this syllabus or its predecessors:

Dani Almog
Sebastian Chece
Todd DeCapua
Wim Decoutere
Frans Dijkman
Jiangru Fu
Matthias Hamburg
Ágota Horváth
Mieke Jungebload
Beata Karpinska
Gábor Ladányi
Kai Lepler
Ine Lutterman
Marek Majernik
Stefan Massonet
Judy McKay
Gary Mogyorodi
Joern Muenzel
Petr Neugebauer
Ingvar Nordström
Meile Posthuma
Michaël Pilaeten
Filip Rechtoris
Adam Roman
Dirk Schweier
Marcus Seyfert
Péter Sótér
Michael Stahl
Jan Stiller
Federico Toledo
Andrea Szabó
Yaron Tsubery
Stephanie Ulrich
Mohit Verma
Armin Wachter
Huaiwen Yang
Ting Yang

This document was formally released by the ISTQB on December 9th, 2018.

0. Introduction to this Syllabus

0.1 Purpose of this Document

This syllabus forms the basis for the qualification of Performance Testing at the Foundation Level. The ASTQB® and GTB® provide this syllabus as follows:

1. To National Boards, to translate into their local language and to accredit training providers. National Boards may adapt the syllabus to their particular language needs and modify the references to adapt to their local publications.
2. To Exam Boards, to derive examination questions in their local language adapted to the learning objectives for each syllabus.
3. To training providers, to produce courseware and determine appropriate teaching methods.
4. To certification candidates, to prepare for the exam (as part of a training course or independently).
5. To the international software and systems engineering community, to advance the profession of software and systems testing, and as a basis for books and articles.

The ASTQB and GTB may allow other entities to use this syllabus for other purposes, provided they seek and obtain prior written permission.

0.2 The Certified Foundation Level Performance Testing

The Foundation Level qualification is aimed at anyone involved in software testing who wishes to broaden their knowledge of performance testing or anyone who wishes to start a specialist career in performance testing. The qualification is also aimed at anyone involved in performance engineering who wishes to gain a better understanding of performance testing.

The syllabus considers the following principal aspects of performance testing:

- Technical aspects
- Method-based aspects
- Organizational aspects

Information about performance testing described in the ISTQB® Advanced Level Technical Test Analyst syllabus [ISTQB_ALTTA_SYL] is consistent with and is developed by this syllabus.

0.3 Business Outcomes

This section lists the Business Outcomes expected of a candidate who has achieved the Foundation Level Performance Testing certification.

PTFL-1 Understand the basic concepts of performance efficiency and performance testing
PTFL-2 Define performance risks, goals, and requirements to meet stakeholder needs and expectations
PTFL-3 Understand performance metrics and how to collect them
PTFL-4 Develop a performance test plan for achieving stated goals and requirements
PTFL-5 Conceptually design, implement, and execute basic performance tests
PTFL-6 Analyze the results of a performance test and state implications to various stakeholders
PTFL-7 Explain the process, rationale, results, and implications of performance testing to various stakeholders
PTFL-8 Understand categories and uses for performance tools and criteria for their selection
PTFL-9 Determine how performance testing activities align with the software lifecycle

0.4 Examinable Learning Objectives

The Learning Objectives support the Business Outcomes and are used to create the examination for achieving the Foundation Level Performance Testing Certification. Learning objectives are allocated to a Cognitive level of knowledge (K-Level).

A K-level, or Cognitive level, is used to classify learning objectives according to the revised taxonomy from Bloom [Anderson01]. ISTQB® uses this taxonomy to design its syllabi examinations.

This syllabus considers four different K-levels (K1 to K4):

K-Level	Keyword	Description
1	Remember	The candidate should remember or recognize a term or a concept.
2	Understand	The candidate should select an explanation for a statement related to the question topic.
3	Apply	The candidate should select the correct application of a concept or technique and apply it to a given context.
4	Analyze	The candidate can separate information related to a procedure or technique into its constituent parts for better

		understanding and can distinguish between facts and inferences.

In general, all parts of this syllabus are examinable at a K1 level. That is, the candidate will recognize, remember and recall a term or concept. The learning objectives at K2, K3 and K4 levels are shown at the beginning of the pertinent chapter.

0.5 Recommended Training Times

A minimum training time has been defined for each learning objective in this syllabus. The total time for each chapter is indicated in the chapter heading.

Training providers should note that other ISTQB syllabi apply a “standard time” approach which allocates fixed times according to the K-Level. The Performance Testing syllabus does not strictly apply this scheme. As a result, training providers are given a more flexible and realistic indication of minimum training times for each learning objective.

0.6 Entry Requirements

The Foundation Level Core certificate shall be obtained before taking the Foundation Level Performance Testing certification exam.

0.7 Sources of Information

Terms used in the syllabus are defined in the ISTQB’s Glossary of Terms used in Software Testing [ISTQB_GLOSSARY].

Section 6 contains a list of recommended books and articles on performance testing.

1. Basic Concepts – 60 mins.

Keywords

Capacity testing concurrency testing efficiency, endurance testing, load generation, load testing, performance testing, scalability testing, spike testing, stress testing

Learning Objectives for Basic Concepts

1.1 Principles and Concepts
PTFL-1.1.1 (K2) Understand the principles of performance testing

1.2 Types of Performance Testing
PTFL-1.2.1 (K2) Understand the different types of performance testing

1.3 Testing Types in Performance Testing
PTFL-1.3.1 (K1) Recall testing types in performance testing

1.4 The Concept of Load Generation
PTFL-1.4.1 (K2) Understand the concept of load generation

1.5 Common Failures in Performance Testing and Their Causes
PTFL-1.5.1 (K2) Give examples of common failure modes of performance testing and their causes

1.1 Principles of Performance Testing

Performance efficiency (or simply “performance”) is an essential part of providing a “good experience” for users when they use their applications on a variety of fixed and mobile platforms. Performance testing plays a critical role in establishing acceptable quality levels for the end user and is often closely integrated with other disciplines such as usability engineering and performance engineering.

Additionally, evaluation of functional suitability, usability and other quality characteristics under conditions of load, such as during execution of a performance test, may reveal load-specific issues which impact those characteristics.

Performance testing is not limited to the web-based domain where the end user is the focus. It is also relevant to different application domains with a variety of system architectures, such as classic client-server, distributed and embedded.
Technically, performance efficiency is categorized in the ISO 25010 [ISO25000] Product Quality Model as a non-functional quality characteristic with the three sub-characteristics described below. Proper focus and prioritization depends on the risks

assessed and the needs of the various stakeholders. Test results analysis may identify other areas of risk that need to be addressed.

Time Behavior: Generally the evaluation of time behavior is the most common performance testing objective. This aspect of performance testing examines the ability of a component or system to respond to user or system inputs within a specified time and under specified conditions. Measurements of time behavior may vary from the "end-to-end" time taken by the system to responding to user input, to the number of CPU cycles required by a software component to execute a particular task.

Resource Utilization: If the availability of system resources is identified as a risk, the utilization of those resources (e.g., the allocation of limited RAM) may be investigated by conducting specific performance tests.

Capacity: If issues of system behavior at the required capacity limits of the system (e.g., numbers of users or volumes of data) is identified as a risk, performance tests may be conducted to evaluate the suitability of the system architecture.

Performance testing often takes the form of experimentation, which enables measurement and analysis of specific system parameters to take place. These may be conducted iteratively in support of system analysis, design and implementation to enable architectural decisions to be made and to help shape stakeholder expectations.

The following performance testing principles are particularly relevant.

- Tests must be aligned to the defined expectations of different stakeholder groups, in particular users, system designers and operations staff.
- The tests must be reproducible. Statistically identical results (within a specified tolerance) must be obtained by repeating the tests on an unchanged system.
- The tests must yield results that are both understandable and can be readily compared to stakeholder expectations.
- The tests can be conducted, where resources allow, either on complete or partial systems or test environments that are representative of the production system.
- The tests must be practically affordable and executable within the timeframe set by the project.

Books by [Molyneaux09] and [Microsoft07] provide a solid background to the principles and practical aspects of performance testing.

All three of the above quality sub-characteristics will impact the ability of the system under test (SUT) to scale.

1.2 Types of Performance Testing

Different types of performance testing can be defined. Each of these may be applicable to a given project, depending on the objectives of the test.

Performance Testing

Performance testing is an umbrella term including any kind of testing focused on performance (responsiveness) of the system or component under different volumes of load.

Load Testing

Load testing focuses on the ability of a system to handle increasing levels of anticipated realistic loads resulting from transaction requests generated by controlled numbers of concurrent users or processes.

Stress Testing

Stress testing focuses on the ability of a system or component to handle peak loads that are at or beyond the limits of its anticipated or specified workloads. Stress testing is also used to evaluate a system's ability to handle reduced availability of resources such as accessible computing capacity, available bandwidth, and memory.

Scalability Testing

Scalability testing focuses on the ability of a system to meet future efficiency requirements which may be beyond those currently required. The objective of these tests is to determine the system's ability to grow (e.g., with more users, larger amounts of data stored) without violating the currently specified performance requirements or failing. Once the limits of scalability are known, threshold values can be set and monitored in production to provide a warning of problems which may be about to arise.. In addition the production environment may be adjusted with appropriate amounts of hardware.

Spike Testing

Spike testing focuses on the ability of a system to respond correctly to sudden bursts of peak loads and return afterwards to a steady state.

Endurance Testing

Endurance testing focuses on the stability of the system over a time frame specific to the system's operational context. This type of testing verifies that there are no resource capacity problems (e.g., memory leaks, database connections, thread pools) that may eventually degrade performance and/or cause failures at breaking points.

Concurrency Testing

Concurrency testing focuses on the impact of situations where specific actions occur simultaneously (e.g., when large numbers of users log in at the same time). Concurrency issues are notoriously difficult to find and reproduce, particularly when the problem occurs in an environment where testing has little or no control, such as production.

Capacity Testing

Capacity testing determines how many users and/or transactions a given system will support and still meet the stated performance objectives. These objectives may also be stated with regard to the data volumes resulting from the transactions.

1.3 Testing Types in Performance Testing

The principal testing types used in performance testing include static testing and dynamic testing.

1.3.1 Static testing

Static testing activities are often more important for performance testing than for functional suitability testing. This is because so many critical performance defects are introduced in the architecture and design of the system. These defects can be introduced by misunderstandings or a lack of knowledge by the designers and architects. These defects can also be introduced because the requirements did not adequately capture the response time, throughput, or resource utilization targets, the expected load and usage of the system, or the constraints.

Static testing activities for performance can include:

- Reviews of requirements with focus on performance aspects and risks
- Reviews of database schemas, entity-relationship diagrams, metadata, stored procedures and queries
- Reviews of the system and network architecture
- Reviews of critical segments of the system code (e.g., complex algorithms)

1.3.2 Dynamic testing

As the system is built, dynamic performance testing should start as soon as possible. Opportunities for dynamic performance testing include:

- During unit testing, including using profiling information to determine potential bottlenecks and dynamic analysis to evaluate resource utilization
- During component integration testing, across key use cases and workflows, especially when integrating different use case features or integrating with the “backbone” structure of a workflow

- During system testing of overall end-to-end behaviors under various load conditions
- During system integration testing, especially for data flows and workflows across key inter-system interfaces. In system integration testing is not uncommon for the “user” to be another system or machine (e.g. inputs from sensor inputs and other systems)
- During acceptance testing, to build user, customer, and operator confidence in the proper performance of the system and to fine tune the system under real world conditions (but generally not to find performance defects in the system)

In higher test levels such as system testing and system integration testing, the use of realistic environments, data, and loads are critical for accurate results (see Chapter 4). In Agile and other iterative-incremental lifecycles, teams should incorporate static and dynamic performance testing into early iterations rather than waiting for final iterations to address performance risks.

If custom or new hardware is part of the system, early dynamic performance tests can be performed using simulators. However, it is good practice to start testing on the actual hardware as soon as possible, as simulators often do not adequately capture resource constraints and performance-related behaviors.

1.4 The Concept of Load Generation

In order to carry out the various types of performance testing described in Section 1.2, representative system loads must be modeled, generated and submitted to the system under test. Loads are comparable to the data inputs used for functional test cases, but differ in the following principal ways:

- A performance test load must represent many user inputs, not just one
- A performance test load may require dedicated hardware and tools for generation
- Generation of a performance test load is dependent on the absence of any functional defects in the system under test which may impact test execution

The efficient and reliable generation of a specified load is a key success factor when conducting performance tests. There are different options for load generation.

Load Generation via the User Interface
This may be an adequate approach if only a small number of users are to be represented and if the required numbers of software clients are available from which to enter required inputs. This approach may also be used in conjunction with functional test execution tools, but may rapidly become impractical as the numbers of users to be simulated increases. The stability of the user interface (UI) also represents a critical dependency. Frequent changes can impact the repeatability of performance tests and

may significantly affect the maintenance costs. Testing through the UI may be the most representative approach for end-to-end tests.

Load Generation using Crowds
This approach depends on the availability of a large number of testers who will represent real users. In crowd testing, the testers are organized such that the desired load can be generated. This may be a suitable method for testing applications that are reachable from anywhere in the world (e.g., web-based), and may involve the users generating a load from a wide range of different device types and configurations. Although this approach may enable very large numbers of users to be utilized, the load generated will not be as reproducible and precise as other options and is more complex to organize.

Load Generation via the Application Programming Interface (API)
This approach is similar to using the UI for data entry, but uses the application's API instead of the UI to simulate user interaction with the system under test. The approach is therefore less sensitive to changes (e.g., delays) in the UI and allows the transactions to be processed in the same way as they would if entered directly by a user via the UI. Dedicated scripts may be created which repeatedly call specific API routines and enable more users to be simulated compared to using UI inputs.

Load Generation using Captured Communication Protocols
This approach involves capturing user interaction with the system under test at the communications protocol level and then replaying these scripts to simulate potentially very large numbers of users in a repeatable and reliable manner. This tool-based approach is described in Sections 4.2.6 and 4.2.7.

1.5 Common Performance Efficiency Failure Modes and Their Causes

While there certainly are many different performance failure modes that can be found during dynamic testing, the following are some examples of common failures (including system crashes), along with typical causes:

Slow response under all load levels
In some cases, response is unacceptable regardless of load. This may be caused by underlying performance issues, including, but not limited to, bad database design or implementation, network latency, and other background loads. Such issues can be identified during functional and usability testing, not just performance testing, so test analysts should keep an eye open for them and report them.

Slow response under moderate-to-heavy load levels
In some cases, response degrades unacceptably with moderate-to-heavy load, even when such loads are entirely within normal, expected, allowed ranges. Underlying defects include saturation of one or more resources and varying background loads.

Degraded response over time
In some cases, response degrades gradually or severely over time. Underlying causes include memory leaks, disk fragmentation, increasing network load over time, growth of the file repository, and unexpected database growth.

Inadequate or graceless error handling under heavy or over-limit load
In some cases, response time is acceptable but error handling degrades at high and beyond-limit load levels. Underlying defects include insufficient resource pools, undersized queues and stacks, and too rapid time-out settings.

Specific examples of the general types of failures listed above include:

- A web-based application that provides information about a company's services does not respond to user requests within seven seconds (a general industry rule of thumb). The performance efficiency of the system cannot be achieved under specific load conditions.
- A system crashes or is unable to respond to user inputs when subjected to a sudden large number of user requests (e.g., ticket sales for a major sporting event). The capacity of the system to handle this number of users is inadequate.
- System response is significantly degraded when users submit requests for large amounts of data (e.g., a large and important report is posted on a web site for download). The capacity of the system to handle the generated data volumes is insufficient.
- Batch processing is unable to complete before online processing is needed. The execution time of the batch processes is insufficient for the time period allowed.
- A real-time system runs out of RAM when parallel processes generate large demands for dynamic memory which cannot be released in time. The RAM is not dimensioned adequately, or requests for RAM are not adequately prioritized.
- A real-time system component A which supplies inputs to real-time system component B is unable to calculate updates at the required rate. The overall system fails to respond in time and may fail. Code modules in component A must be evaluated and modified ("performance profiling") to ensure that the required update rates can be achieved.

2. Performance Measurement Fundamentals - 55 mins.

Keywords

measurement, metric

Learning Objectives for Performance Measurement Fundamentals

2.1 Typical Metrics Collected in Performance Testing
PTFL-2.1.1 (K2) Understand the typical metrics collected in performance testing

2.2 Aggregating Results from Performance Testing
PTFL-2.2.1 (K2) Explain why results from performance testing are aggregated

2.3 Key **Sources of Performance Metrics**
PTFL-2.3.1 (K2) Understand the key sources of performance metrics

2.4 Typical Results of a Performance Test
PTFL-2.4.1 (K1) Recall the typical results of a performance test

2.1 Typical Metrics Collected in Performance Testing

2.1.1 Why Performance Metrics are Needed

Accurate measurements and the metrics which are derived from those measurements are essential for defining the goals of performance testing and for evaluating the results of performance testing. Performance testing should not be undertaken without first understanding which measurements and metrics are needed. The following project risks apply if this advice is ignored:

- It is unknown if the levels of performance are acceptable to meet operational objectives
- The performance requirements are not defined in measurable terms
- It may not be possible to identify trends that may predict lower levels of performance
- The actual results of a performance test cannot be evaluated by comparing them to a baseline set of performance measures that define acceptable and/or unacceptable performance
- Performance test results are evaluated based on the subjective opinion of one or more people
- The results provided by a performance test tool are not understood

2.1.2 Collecting Performance Measurements and Metrics

As with any form of measurement, it is possible to obtain and express metrics in precise ways. Therefore, any of the metrics and measurements described in this section can and should be defined to be meaningful in a particular context. This is a matter of performing initial tests and learning which metrics need to be further refined and which need to be added.

For example, the metric of response time likely will be in any set of performance metrics. However, to be meaningful and actionable, the response time metric will need to be further defined in terms of time of day, number of concurrent users, the amount of data being processed and so forth.

The metrics collected in a specific performance test will vary based on the

- business context (business processes, customer and user behavior, and stakeholder expectations),
- operational context (technology and how it is used)
- test objectives

For example, the metrics chosen for the performance testing of an international e-commerce website will differ from those chosen for the performance testing of an embedded system used to control medical device functionality.

A common way to categorize performance measurements and metrics is to consider the technical environment, business environment, or operational environment in which the assessment of performance is needed.

The categories of measurements and metrics included below are the ones commonly obtained from performance testing.

Technical Environment
Performance metrics will vary by the type of the technical environment, as shown in the following list:

- Web-based
- Mobile
- Internet-of-Things (IoT)
- Desktop client devices
- Server-side processing
- Mainframe
- Databases
- Networks
- The nature of software running in the environment (e.g., embedded)

The metrics include the following:

- Response time (e.g., per transaction, per concurrent user, page load times)

- Resource utilization (e.g., CPU, memory, network bandwidth, network latency, available disk space, I/O rate, idle and busy threads)
- Throughput rate of key transaction (i.e., the number of transactions that can be processed in a given period of time)
- Batch processing time (e.g., wait times, throughput times, data base response times, completion times)
- Numbers of errors impacting performance
- Completion time (e.g., for creating, reading, updating, and deleting data)
- Background load on shared resources (especially in virtualized environments)
- Software metrics (e.g., code complexity)

Business Environment

From the business or functional perspective, performance metrics may include the following:

- Business process efficiency (e.g., the speed of performing an overall business process including normal, alternate and exceptional use case flows)
- Throughput of data, transactions, and other units of work performed (e.g., orders processed per hour, data rows added per minute)
- Service Level Agreement (SLA) compliance or violation rates (e.g., SLA violations per unit of time)
- Scope of usage (e.g., percentage of global or national users conducting tasks at a given time)
- Concurrency of usage (e.g., the number of users concurrently performing a task)
- Timing of usage (e.g., the number of orders processed during peak load times)

Operational Environment

The operational aspect of performance testing focuses on tasks that are generally not considered to be user-facing in nature. These include the following:

- Operational processes (e.g., the time required for environment start-up, backups, shutdown and resumption times)
- System restoration (e.g., the time required to restore data from a backup)
- Alerts and warnings (e.g., the time needed for the system to issue an alert or warning)

2.1.3 Selecting Performance Metrics

It should be noted that collecting more metrics than required is not necessarily a good thing. Each metric chosen requires a means for consistent collection and reporting. It is important to define an obtainable set of metrics that support the performance test objectives.

For example, the Goal-Question-Metric (GQM) approach is a helpful way to align metrics with performance goals. The idea is to first establish the goals, then ask questions to know when the goals have been achieved. Metrics are associated with

each question to ensure the answer to the question is measurable. (See Section 4.3 of the Expert Level Syllabus – Improving the Testing Process [ISTQB_ELTM_ITP_SYL] for a more complete description of the GQM approach.) It should be noted that the GQM approach doesn't always fit the performance testing process. For example, some metrics represent a system's health and are not directly linked to goals.

It is important to realize that after the definition and capture of initial measurements further measurements and metrics may be needed to understand true performance levels and to determine where corrective actions may be needed.

2.2 Aggregating Results from Performance Testing

The purpose of aggregating performance metrics is to be able to understand and express them in a way that accurately conveys the total picture of system performance. When performance metrics are viewed at only the detailed level, drawing the right conclusion may be difficult—especially for business stakeholders.

For many stakeholders, the main concern is that the response time of a system, web site, or other test object is within acceptable limits.

Once deeper understanding of the performance metrics has been achieved, the metrics can be aggregated so that:

- Business and project stakeholders can see the "big picture" status of system performance
- Performance trends can be identified
- Performance metrics can be reported in an understandable way

2.3 Key Sources of Performance Metrics

System performance should be no more than minimally impacted by the metrics collection effort (known as the "probe effect"). In addition, the volume, accuracy and speed with which performance metrics must be collected makes tool usage a requirement. While the combined use of tools is not uncommon, it can introduce redundancy in the usage of test tools and other problems (see Section 4.4).

There are three key sources of performance metrics:

Performance Test Tools

All performance test tools provide measurements and metrics as the result of a test. Tools may vary in the number of metrics shown, the way in which the metrics are shown, and the ability for the user to customize the metrics to a particular situation (see also Section 5.1).

Some tools collect and display performance metrics in text format, while more robust tools collect and display performance metrics graphically in a dashboard format. Many tools offer the ability to export metrics to facilitate test evaluation and reporting.

Performance Monitoring Tools

Performance monitoring tools are often employed to supplement the reporting capabilities of performance test tools (see also Section 5.1). In addition, monitoring tools may be used to monitor system performance on an ongoing basis and to alert system administrators to lowered levels of performance and higher levels of system errors and alerts. These tools may also be used to detect and notify in the event of suspicious behavior (such as denial of service attacks and distributed denial of Service attacks).

Log Analysis Tools

There are tools that scan server logs and compile metrics from them. Some of these tools can create charts to provide a graphical view of the data.

Errors, alerts and warnings are normally recorded in server logs. These include:

- High resource usage, such as high CPU utilization, high levels of disk storage consumed, and insufficient bandwidth
- Memory errors and warnings, such as memory exhaustion
- Deadlocks and multi-threading problems, especially when performing database operations
- Database errors, such as SQL exceptions and SQL timeouts

2.4 Typical Results of a Performance Test

In functional testing, particularly when verifying specified functional requirements or functional elements of user stories, the expected results usually can be defined clearly and the test results interpreted to determine if the test passed or failed. For example, a monthly sales report shows either a correct or an incorrect total.

Whereas tests that verify functional suitability often benefit from well-defined test oracles, performance testing often lacks this source of information. Not only are the stakeholders notoriously bad at articulating performance requirements, many business analysts and product owners are bad at eliciting such requirements. Testers often receive limited guidance to define the expected test results.

When evaluating performance test results, it is important to look at the results closely. Initial raw results can be misleading with performance failures being hidden beneath apparently good overall results. For example, resource utilization may be well under 75% for all key potential bottleneck resources, but the throughput or response time of key transactions or use cases are an order-of-magnitude too slow.

The specific results to evaluate vary depending on the tests being run, and often include those discussed in Section 2.1.

3. Performance Testing in the Software Lifecycle – 195 mins.

Keywords

metric, risk, software development lifecycle, test log

Learning Objectives

3.1 Principal Performance Testing Activities
PTFL-3.1.1 (K2) Understand the principal performance testing activities

3.2 Performance Risks for Different Architectures
PTFL-3.2.1 (K2) Explain typical categories of performance risks for different architectures

3.3 Performance Risks Across the Software Development Lifecycle
PTFL-3.3.1 (K4) Analyze performance risks for a given product across the software development lifecycle

3.4 Performance Testing Activities
PTFL-3.4.1 (K4) Analyze a given project to determine the appropriate performance testing activities for each phase of the software development lifecycle

3.1 Principal Performance Testing Activities

Performance testing is iterative in nature. Each test provides valuable insights into application and system performance. The information gathered from one test is used to correct or optimize application and system parameters. The next test iteration will then show the results of modifications, and so on until test objectives are reached.

Performance testing activities align with the ISTQB test process [ISTQB_FL_SYL].

Test Planning
Test planning is particularly important for performance testing due to the need for the allocation of test environments, test data, tools and human resources. In addition, this is the activity in which the scope of performance testing is established.

During test planning, risk identification and risk analysis activities are completed and relevant information is updated in any test planning documentation (e.g., test plan, level test plan). Just as test planning is revisited and modified as needed, so are risks, risk levels and risk status modified to reflect changes in risk conditions.

Test Monitoring and Control

Control measures are defined to provide action plans should issues be encountered which might impact performance efficiency, such as

- increasing the load generation capacity if the infrastructure does not generate the desired loads as planned for particular performance tests
- changed, new or replaced hardware
- changes to network components
- changes to software implementation

The performance test objectives are evaluated to check for exit criteria achievement.

Test Analysis
Effective performance tests are based on an analysis of performance requirements, test objectives, Service Level Agreements (SLA), IT architecture, process models and other items that comprise the test basis. This activity may be supported by modeling and analysis of system resource requirements and/or behavior using spreadsheets or capacity planning tools.

Specific test conditions are identified such as load levels, timing conditions, and transactions to be tested. The required type(s) of performance test (e.g., load, stress, scalability) are then decided.

Test Design
Performance test cases are designed. These are generally created in modular form so that they may be used as the building blocks of larger, more complex performance tests (see section 4.2).

Test Implementation
In the implementation phase, performance test cases are ordered into performance test procedures. These performance test procedures should reflect the steps normally taken by the user and other functional activities that are to be covered during performance testing.

A test implementation activity is establishing and/or resetting the test environment before each test execution. Since performance testing is typically data-driven, a process is needed to establish test data that is representative of actual production data in volume and type so that production use can be simulated.

Test Execution
Test execution occurs when the performance test is conducted, often by using performance test tools. Test results are evaluated to determine if the system's performance meets the requirements and other stated objectives. Any defects are reported.

Test Completion
Performance test results are provided to the stakeholders (e.g., architects, managers, product owners) in a test summary report. The results are expressed through metrics which are often aggregated to simplify the meaning of the test results. Visual means of reporting such as dashboards are often used to express performance test results in ways that are easier to understand than text-based metrics.

Performance testing is often considered to be an ongoing activity in that it is performed at multiple times and at all test levels (component, integration, system, system integration and acceptance testing). At the close of a defined period of performance testing, a point of test closure may be reached where designed tests, test tool assets (test cases and test procedures), test data and other testware are archived or passed on to other testers for later use during system maintenance activities.

3.2 Categories of Performance Risks for Different Architectures

As mentioned previously, application or system performance varies considerably based on the architecture, application and host environment. While it is not possible to provide a complete list of performance risks for all systems, the list below includes some typical types of risks associated with particular architectures:

Single Computer Systems
These are systems or applications that runs entirely on one non-virtualized computer. Performance can degrade due to

- excessive resource consumption including memory leaks, background activities such as security software, slow storage subsystems (e.g., low-speed external devices or disk fragmentation), and operating system mismanagement.
- inefficient implementation of algorithms which do not make use of available resources (e.g., main memory) and as a result execute slower than required.

Multi-tier Systems
These are systems of systems that run on multiple servers, each of which performs a specific set of tasks, such as database server, application server, and presentation server. Each server is, of course, a computer and subject to the risks given earlier. In addition, performance can degrade due to poor or non-scalable database design, network bottlenecks, and inadequate bandwidth or capacity on any single server.

Distributed Systems
These are systems of systems, similar to a multi-tier architecture, but the various servers may change dynamically, such as an e-commerce system that accesses different inventory databases depending on the geographic location of the person placing the order. In addition to the risks associated with multi-tier architectures, this architecture can experience performance problems due to critical workflows or dataflows to, from, or through unreliable or unpredictable remote servers, especially

when such servers suffer periodic connection problems or intermittent periods of intense load.

Virtualized Systems
These are systems where the physical hardware hosts multiple virtual computers. These virtual machines may host single-computer systems and applications as well as servers that are part of a multi-tier or distributed architecture. Performance risks that arise specifically from virtualization include excessive load on the hardware across all the virtual machines or improper configuration of the host virtual machine resulting in inadequate resources.

Dynamic/Cloud-based Systems
These are systems that offer the ability to scale on demand, increasing capacity as the level of load increases. These systems are typically distributed and virtualized multi-tier systems, albeit with self-scaling features designed specifically to mitigate some of the performance risks associated with those architectures. However, there are risks associated with failures to properly configure these features during initial setup or subsequent updates.

Client –Server Systems
These are systems running on a client that communicate via a user interface with a single server, multi-tier server, or distributed server. Since there is code running on the client, the single computer risks apply to that code, while the server-side issues mentioned above apply as well. Further, performance risks exist due to connection speed and reliability issues, network congestion at the client connection point (e.g., public Wi-Fi), and potential problems due to firewalls, packet inspection and server load balancing.

Mobile Applications
This are applications running on a smartphone, tablet, or other mobile device. Such applications are subject to the risks mentioned for client-server and browser-based (web apps) applications. In addition, performance issues can arise due to the limited and variable resources and connectivity available on the mobile device (which can be affected by location, battery life, charge state, available memory on the device and temperature). For those applications that use device sensors or radios such as accelerometers or Bluetooth, slow dataflows from those sources could create problems. Finally, mobile applications often have heavy interactions with other local mobile apps and remote web services, any of which can potentially become a performance efficiency bottleneck.

Embedded Real-time Systems
These are systems that work within or even control everyday things such as cars (e.g., entertainment systems and intelligent braking systems), elevators, traffic signals, Heating, Ventilation and Air Conditioning (HVAC) systems, and more. These systems often have many of the risks of mobile devices, including (increasingly) connectivity-

related issues since these devices are connected to the Internet. However the diminished performance of a mobile video game is usually not a safety hazard for the user, while such slowdowns in a vehicle braking system could prove catastrophic.

Mainframe Applications
These are applications—in many cases decades-old applications—supporting often mission-critical business functions in a data center, sometimes via batch processing. Most are quite predictable and fast when used as originally designed, but many of these are now accessible via APIs, web services, or through their database, which can result in unexpected loads that affect throughput of established applications.

Note that any particular application or system may incorporate two or more of the architectures listed above, which means that all relevant risks will apply to that application or system. In fact, given the Internet of Things and the explosion of mobile applications—two areas where extreme levels of interaction and connection is the rule—it is possible that all architectures are present in some form in an application, and thus all risks can apply.

While architecture is clearly an important technical decision with a profound impact on performance risks, other technical decisions also influence and create risks. For example, memory leaks are more common with languages that allow direct heap memory management, such as C and C++, and performance issues are different for relational versus non-relational databases. Such decisions extend all the way down to the design of individual functions or methods (e.g., the choice of a recursive as opposed to an iterative algorithm). As a tester, the ability to know about or even influence such decisions will vary, depending on the roles and responsibilities of testers within the organization and software development lifecycle.

3.3 Performance Risks Across the Software Development Lifecycle

The process of analyzing risks to the quality of a software product in general is discussed in various ISTQB syllabi (e.g., see [ISTQB_FL_SYL] and [ISTQB_ALTM_SYL]). You can also find discussions of specific risks and considerations associated with particular quality characteristics (e.g., [ISTQB_UT_SYL]), and from a business or technical perspective (e.g., see [ISTQB_ALTA_SYL] and [ISTQB_ALTTA_SYL], respectively). In this section, the focus is on performance-related risks to product quality, including ways that the process, the participants, and the considerations change.

For performance-related risks to the quality of the product, the process is:

1. Identify risks to product quality, focusing on characteristics such as time behavior, resource utilization, and capacity.

2. Assess the identified risks, ensuring that the relevant architecture categories (see Section 3.2) are addressed. Evaluate the overall level of risk for each identified risk in terms of likelihood and impact using clearly defined criteria.
3. Take appropriate risk mitigation actions for each risk item based on the nature of the risk item and the level of risk.
4. Manage risks on an ongoing basis to ensure that the risks are adequately mitigated prior to release.

As with quality risk analysis in general, the participants in this process should include both business and technical stakeholders. For performance-related risk analysis the business stakeholders must include those with a particular awareness of how performance problems in production will actually affect customers, users, the business, and other downstream stakeholders. Business stakeholders must appreciate that intended usage, business-, societal-, or safety-criticality, potential financial and/or reputational damage, civil or criminal legal liability and similar factors affect risk from a business perspective, creating risks and influencing the impact of failures.

Further, the technical stakeholders must include those with a deep understanding of the performance implications of relevant requirements, architecture, design, and implementation decisions. Technical stakeholders must appreciate that architecture, design, and implementation decisions affect performance risks from a technical perspective, creating risks and influencing the likelihood of defects.

The specific risk analysis process chosen should have the appropriate level of formality and rigor. For performance-related risks, it is especially important that the risk analysis process be started early and is repeated regularly. In other words, the tester should avoid relying entirely on performance testing conducted towards the end of the system test level and system integration test level. Many projects, especially larger and more complex systems of systems projects, have met with unfortunate surprises due to the late discovery of performance defects which resulted from requirements, design, architecture, and implementation decisions made early in the project. The emphasis should therefore be on an iterative approach to performance risk identification, assessment, mitigation, and management throughout the software development lifecycle.

For example, if large volumes of data will be handled via a relational database, the slow performance of many-to-many joins due to poor database design may only reveal itself during dynamic testing with large-scale test datasets, such as those used during system test. However, a careful technical review that includes experienced database engineers can predict the problems prior to database implementation. After such a review, in an iterative approach, risks are identified and assessed again.

In addition, risk mitigation and management must span and influence the entire software development process, not just dynamic testing. For example, when critical performance-related decisions such as the expected number of transactions or

simultaneous users cannot be specified early in the project, it is important that design and architecture decisions allow for highly variable scalability (e.g., on-demand cloud-based computing resources). This enables early risk mitigation decisions to be made.

Good performance engineering can help project teams avoid the late discovery of critical performance defects during higher test levels, such as system integration testing or user acceptance testing. Performance defects found at a late stage in the project can be extremely costly and may even lead to the cancellation of entire projects.

As with any type of quality risk, performance-related risks can never be avoided completely, i.e., some risk of performance-related production failure will always exist. Therefore, the risk management process must include providing a realistic and specific evaluation of the residual level of risk to the business and technical stakeholders involved in the process. For example, simply saying, "Yes, it's still possible for customers to experience long delays during check out," is not helpful, as it gives no idea of what amount of risk mitigation has occurred or of the level of risk that remains. Instead, providing clear insight into the percentage of customers likely to experience delays equal to or exceeding certain thresholds will help people understand the status.

3.4 Performance Testing Activities

Performance testing activities will be organized and performed differently, depending on the type of software development lifecycle in use.

Sequential Development Models

The ideal practice of performance testing in sequential development models is to include performance criteria as a part of the acceptance criteria which are defined at the outset of a project. Reinforcing the lifecycle view of testing, performance testing activities should be conducted throughout the software development lifecycle. As the project progresses, each successive performance test activity should be based on items defined in the prior activities as shown below.

- Concept – Verify that system performance goals are defined as acceptance criteria for the project.
- Requirements – Verify that performance requirements are defined and represent stakeholder needs correctly.
- Analysis and Design – Verify that the system design reflects the performance requirements.
- Coding/Implementation – Verify that the code is efficient and reflects the requirements and design in terms of performance.
- Component Testing – Conduct component level performance testing.
- Component Integration Testing – Conduct performance testing at the component integration level.
- System Testing – Conduct performance testing at the system level, which includes hardware, software, procedures and data that are representative of

the production environment. System interfaces may be simulated provided that they give a true representation of performance.
- System Integration Testing– Conduct performance testing with the entire system which is representative of the production environment.
- Acceptance Testing – Validate that system performance meets the originally stated user needs and acceptance criteria.

Iterative and Incremental Development Models
In these development models, such as Agile, performance testing is also seen as an iterative and incremental activity (see [ISTQB_FL_AT]).Performance testing can occur as part of the first iteration, or as an iteration dedicated entirely to performance testing. However, with these lifecycle models, the execution of performance testing may be performed by a separate team tasked with performance testing.

Continuous Integration (CI) is commonly performed in iterative and incremental software development lifecycles, which facilitates a highly automated execution of tests. The most common objective of testing in CI is to perform regression testing and ensure each build is stable. Performance testing can be part of the automated tests performed in CI if the tests are designed in such a way as to be executed at a build level. However, unlike functional automated tests, there are additional concerns such as the following:
- The setup of the performance test environment – This often requires a test environment that is available on demand, such as a cloud-based performance test environment.
- Determining which performance tests to automate in CI – Due to the short timeframe available for CI tests, CI performance tests may be a subset of more extensive performance tests that are conducted by a specialist team at other times during an iteration.
- Creating the performance tests for CI – The main objective of performance tests as part of CI is to ensure a change does not negatively impact performance. Depending on the changes made for any given build, new performance tests may be required.
- Executing performance tests on portions of an application or system – This often requires the tools and test environments to be capable of rapid performance testing including the ability to select subsets of applicable tests.

Performance testing in the iterative and incremental software development lifecycles can also have its own lifecycle activities:
- Release Planning – In this activity, performance testing is considered from the perspective of all iterations in a release, from the first iteration to the final iteration. Performance risks are identified and assessed, and mitigation measures planned. This often includes planning of any final performance testing before the release of the application.

- Iteration Planning – In the context of each iteration, performance testing may be performed within the iteration and as each iteration is completed. Performance risks are assessed in more detail for each user story.
- User Story Creation – User stories often form the basis of performance requirements in Agile methodologies, with the specific performance criteria described in the associated acceptance criteria. These are referred to as "non-functional" user stories.
- Design of performance tests –performance requirements and criteria which are described in particular user stories are used for the design of tests (see section 4.2)
- Coding/Implementation – During coding, performance testing may be performed at a component level. An example of this would be the tuning of algorithms for optimum performance efficiency.
- Testing/Evaluation – While testing is typically performed in close proximity to development activities, performance testing may be performed as a separate activity, depending on the scope and objectives of performance testing during the iteration. For example, if the goal of performance testing is to test the performance of the iteration as a completed set of user stories, a wider scope of performance testing will be needed than that seen in performance testing a single user story. This may be scheduled in a dedicated iteration for performance testing.
- Delivery – Since delivery will introduce the application to the production environment, performance will need to be monitored to determine if the application achieves the desired levels of performance in actual usage.

Commercial Off-the-Shelf (COTS) and other Supplier/Acquirer Models

Many organizations do not develop applications and systems themselves, but instead are in the position of acquiring software from vendor sources or from open-source projects. In such supplier/acquirer models, performance is an important consideration that requires testing from both the supplier (vendor/developer) and acquirer (customer) perspectives.

Regardless of the source of the application, it is often the responsibility of the customer to validate that the performance meets their requirements. In the case of customized vendor-developed software, performance requirements and associated acceptance criteria which should be specified as part of the contract between the vendor and customer. In the case of COTS applications, the customer has sole responsibility to test the performance of the product in a realistic test environment prior to deployment.

4. Performance Testing Tasks– 475 mins.

Keywords

concurrency, load profile, load generation, operational profile, ramp-down, ramp-up, system of systems, system throughput, test plan, think time, virtual user

Learning Objectives

4.1 Planning

PTFL-4.1.1 (K4) Derive performance test objectives from relevant information
PTFL-4.1.2 (K4) Outline a performance test plan which considers the performance objectives for a given project
PTFL-4.1.3 (K4) Create a presentation that enables various stakeholders to understand the rationale behind the planned performance testing

4.2 Analysis, Design and Implementation

PTFL-4.2.1 (K2) Give examples of typical protocols encountered in performance testing
PTFL-4.2.2 (K2) Understand the concept of transactions in performance testing
PTFL-4.2.3 (K4) Analyze operational profiles for system usage
PTFL-4.2.4 (K4) Create load profiles derived from operational profiles for given performance objectives
PTFL-4.2.5 (K4) Analyze throughput and concurrency when developing performance tests
PTFL-4.2.6 (K2) Understand the basic structure of a performance test script
PTFL-4.2.7 (K3) Implement performance test scripts consistent with the plan and load profiles
PTFL-4.2.8 (K2) Understand the activities involved in preparing for performance test execution

4.3 Execution

PTFL-4.3.1 (K2) Understand the principal activities in running performance test scripts

4.4 Analyzing Results and Reporting

PTFL-4.4.1 (K4) Analyze and report performance test results and implications

4.1 Planning

4.1.1 Deriving Performance Test Objectives

Stakeholders may include users and people with a business or technical background. They may have different objectives relating to performance testing. Stakeholders set

the objectives, the terminology to be used and the criteria for determining whether the objective has been achieved

Objectives for performance tests relate back to these different types of stakeholders. It is a good practice to distinguish between user-based and technical objectives. User-based objectives focus primarily on end-user satisfaction and business goals. Generally, users are less concerned about feature types or how a product gets delivered. They just want to be able to do what they need to do.

Technical objectives, on the other hand, focus on operational aspects and providing answers to questions regarding a system's ability to scale, or under what conditions degraded performance may become apparent.

Key objectives of performance testing include identifying potential risks, finding opportunities for improvement, and identifying necessary changes.

When gathering information from the various stakeholders, the following questions should be answered:

- What transactions will be executed in the performance test and what average response time is expected?
- What system metrics are to be captured (e.g., memory usage, network throughput) and what values are expected?
- What performance improvements are expected from these tests compared to previous test cycles?

4.1.2 The Performance Test Plan

The Performance Test Plan (PTP) is a document created prior to any performance testing occurring. The PTP should be referred to by the Test Plan (see [ISTQB_FL_SYL]) which also includes relevant scheduling information. It continues to be updated once performance testing begins.

The following information should be supplied in a PTP:

Objective
The PTP objective describes the goals, strategies and methods for the performance test. It enables a quantifiable answer to the central question of the adequacy and the readiness of the system to perform under load.

Test Objectives
Overall test objectives for performance efficiency to be achieved by the System Under Test (SUT) are listed for each type of stakeholder (see Section 4.1.1)

System Overview
A brief description of the SUT will provide the context for the measurement of the performance test parameters. The overview should include a high-level description of the functionality being tested under load.

Types of Performance Testing to be Conducted
The types of performance testing to be conducted are listed (see Section 1.2) along with a description of the purpose of each type.

Acceptance Criteria
Performance testing is intended to determine the responsiveness, throughput, reliability and/or scalability of the system under a given workload. In general, response time is a user concern, throughput is a business concern, and resource utilization is a system concern. Acceptance criteria should be set for all relevant measures and related back to the following as applicable:

- Overall performance test objectives
- Service Level Agreements (SLAs)
- Baseline values – A baseline is a set of metrics used to compare current and previously achieved performance measurements. This enables particular performance improvements to be demonstrated and/or the achievement of test acceptance criteria to be confirmed. It may be necessary to first create the baseline using sanitized data from a database, where possible.

Test Data
Test data includes a broad range of data that needs to be specified for a performance test. This data can include the following:

- User account data (e.g., user accounts available for simultaneous log in)
- User input data (e.g., the data a user would enter into the application in order to perform a business process)
- Database (e.g., the pre-populated database that is populated with data for use in testing)

The test data creation process should address the following aspects:

- data extraction from production data
- importing data into the SUT
- creation of new data
- creation of backups that can be used to restore the data when new cycles of testing are performed
- data masking or anonymizing. This practice is used on production data that contains personally identifiable information, and is mandatory under General Data Protection Regulations (GDPR). However, in performance testing, data masking adds risk to the performance tests as it may not have the same data characteristics as seen in real-world use.

System Configuration
The system configuration section of the PTP includes the following technical information:

- A description of the specific system architecture, including servers (e.g., web, database, load balancer)
- Definition of multiple tiers
- Specific details of computing hardware (e.g., CPU cores, RAM, Solid State Disks (SSD), Hard Drive Disks (HDD)) including versions
- Specific details of software (e.g., applications, operating systems, databases, services used to support the enterprise) including versions
- External systems that operates with the SUT and their configuration and version (e.g., Ecommerce system with integration to NetSuite)
- SUT build / version identifier

Test Environment
The test environment is often a separate environment that mimics production, but at a smaller scale. This section of the PTP should include how the results from the performance testing will be extrapolated to apply to the larger production environment. With some systems, the production environment becomes the only viable option for testing, but in this case the specific risks of this type of testing must be discussed.

Testing tools sometimes reside outside the test environment itself and may require special access rights in order to interact with the system components. This is a consideration for the test environment and configuration.

Performance tests may also be conducted with a component part of the system that is capable of operating without other components. This is often cheaper than testing with the whole system and can be conducted as soon as the component is developed.

Test Tools
This section includes a description of which test tools (and versions) will be used in scripting, executing and monitoring the performance tests (see Chapter 5). This list normally includes:

- Tool(s) used to simulate user transactions
- Tools to provide load from multiple points within the system architecture (points of presence)
- Tools to monitor system performance, including those described above under system configuration

Profiles
Operational profiles provide a repeatable step-by-step flow through the application for a particular usage of the system. Aggregating these operational profiles results in a load profile (commonly referred to as a scenario). See Section 4.2.3 for more information on profiles.

Relevant Metrics

A large number of measurements and metrics can be collected during a performance test execution (see Chapter 2). However, taking too many measurements can make analysis difficult as well as negatively impact the application's actual performance. For these reasons, it is important to identify the measurements and metrics that are most relevant to accomplish the objectives of the performance test.

The following table, explained in more detail in Section 4.4, shows a typical set of metrics for performance testing and monitoring. Test objectives for performance should be defined for these metrics, where required, for the project:

Performance Metrics	
Type	**Metric**
Virtual User Status	# Passed # Failed
Transaction Response Time	Minimum Maximum Average 90% Percentile
Transactions Per Second	# Passed / second # Failed / second # Total / second
Hits (e.g., on database or web server)	Hits / second ▪ Minimum ▪ Maximum ▪ Average ▪ Total
Throughput	Bits / second ▪ Minimum ▪ Maximum ▪ Average ▪ Total
HTTP Responses Per Second	Responses / second ▪ Minimum ▪ Maximum ▪ Average ▪ Total Response by HTTP response codes

Performance Monitoring	
Type	**Metric**
CPU usage	% of available CPU used
Memory usage	% of available memory used

Risks
Risks can include areas not measured as part of the performance testing as well as limitations to the performance testing (e.g., external interfaces that cannot be simulated, insufficient load, inability to monitor servers). Limitations of the test environment may also produce risks (e.g., insufficient data, scaled down environment). See Sections 3.2 and 3.3 for more risk types.

4.1.3 Communicating about Performance Testing

The tester must be capable of communicating to all stakeholders the rationale behind the performance testing approach and the activities to be undertaken (as detailed in the Performance Test Plan). The subjects to be addressed in this communication may vary considerably between stakeholders depending on whether they have a “business / user-facing” interest or a more “technology / operations-facing” focus.

Stakeholders with a Business Focus
The following factors should be considered when communicating with stakeholders with a business focus:

- Stakeholders with a business focus are less interested in the distinctions between functional and non-functional quality characteristics.
- Technical issues concerning tooling, scripting and load generation are generally of secondary interest.
- The connection between product risks and performance test objectives must be clearly stated.
- Stakeholders must be made aware of the balance between the cost of planned performance tests and how representative the performance testing results will be, compared to production conditions.
- The repeatability of planned performance tests must be communicated. Will the test be difficult to repeat or can it be repeated with a minimum of effort?
- Project risks must be communicated. These include constraints and dependencies concerning the setup of the tests, infrastructure requirements (e.g., hardware, tools, data, bandwidth, test environment, resources) and dependencies on key staff.
- The high-level activities must be communicated (see Sections 4.2 and 4.3) together with a broad plan containing costs, time schedule and milestones.

Stakeholders with a Technology Focus
The following factors must be considered when communicating with stakeholders with a technology focus:

- The planned approach to generating required load profiles must be explained and the expected involvement of technical stakeholders made clear.
- Detailed steps in the setup and execution of the performance tests must be explained to show the relation of the testing to the architectural risks.
- Steps required to make performance tests repeatable must be communicated. These may include organizational aspects (e.g., participation of key staff) as well as technical issues.
- Where test environments are to be shared, the scheduling of performance tests must be communicated to ensure the test results will not be adversely impacted.
- Mitigations of the potential impact on actual users if performance testing needs to be executed in the production environment must be communicated and accepted.
- Technical stakeholders must be clear about their tasks and when they are scheduled.

4.2 Analysis, Design and Implementation

4.2.1 Typical Communication Protocols

Communication protocols define a set of communications rules between computers and systems. Designing tests properly to target specific parts of the system requires understanding protocols.

Communication protocols are often described by the Open Systems Interconnection (OSI) model layers (see ISO/IEC 7498-1), although some protocols may fall outside of this model. For performance testing, protocols from Layer 5 (Session Layer) to Layer 7 (Application Layer) are most commonly used for performance testing. Common protocols include:

- Database - ODBC, JDBC, other vendor-specific protocols
- Web - HTTP, HTTPS, HTML
- Web Service - SOAP, REST

Generally speaking, the level of the OSI layer which is most in focus in performance testing relates to the level of the architecture being tested. When testing some low-level, embedded architecture for example, the lower numbered layers of the OSI model will be mostly in focus.

Additional protocols used in performance testing include:

- Network - DNS, FTP, IMAP, LDAP, POP3, SMTP, Windows Sockets, CORBA
- Mobile - TruClient, SMP, MMS

- Remote Access - Citrix ICA, RTE
- SOA - MQSeries, JSON, WSCL

It is important to understand the overall system architecture because performance tests can be executed on an individual system component (e.g., web server, database server) or on a whole system via end-to-end testing. Traditional 2-tier applications built with a client-server model specify the “client” as the GUI and primary user interface, and the “server” as the backend database. These applications require the use of a protocol such as ODBC to access the database. With the evolution of web-based applications and multi-tiered architectures, many servers are involved in processing information that is ultimately rendered to the user’s browser.

Depending on the part of the system that is targeted for testing, an understanding is required of the appropriate protocol to be used. Therefore, if the need is to perform end-to-end testing emulating user activity from the browser, a web protocol such as HTTP/HTTPS will be employed. In this way, interaction with the GUI can be bypassed and the tests can focus on the communication and activities of the backend servers.

4.2.2 Transactions

Transactions describe the set of activities performed by a system from the point of initiation to when one or more processes (requests, operations, or operational processes) have been completed. The response time of transactions can be measured for the purpose of evaluating system performance. During a performance test these measurements are used to identify any components that require correction or optimization.

Simulated transactions can include think time to better reflect the timing of a real user taking an action (e.g., pressing the “SEND” button). The transaction response time plus the think time equals the elapsed time for that transaction.

The transaction response times collected during the performance test show how this measurement changes under different loads imposed on the system. Analysis may show no degradation under load while other measurements may show severe degradation. By ramping up load and measuring the underlying transaction times, it is possible to correlate the cause of degradation with the response times of one or more transactions.

Transactions can also be nested so that individual and aggregate activities can be measured. This can be helpful, for example, when understanding the performance efficiency of an online ordering system. The tester may want to measure the discrete steps in the order process (e.g., search for item, add item to cart, pay for item, confirm order) as well as the order process as a whole. By nesting transactions, both sets of information can be gathered in one test.

4.2.3 Identifying Operational Profiles

Operational profiles specify distinct patterns of interaction with an application such as from users or other system components. Multiple operational profiles may be specified for a given application. They may be combined to create a desired load profile for achieving particular performance test objectives (see Section 4.2.4).

The following principal steps for identifying operational profiles are described in this section:

1. Identify the data to be gathered
2. Gather the data using one or more sources
3. Evaluate the data to construct the operational profiles

Identify Data

Where users interact with the system under test the following data is gathered or estimated in order to model their operational profiles (i.e., how they interact with the system):

- Different types of user personas and their roles (e.g., standard user, registered member, administrator, user groups with specific privileges).
- Different generic tasks performed by those users/roles (e.g., browsing a web site for information, searching a web site for a particular product, performing role-specific activities). Note that these tasks are generally best modeled at a high level of abstraction (e.g., at the level of business processes or major user stories).
- Estimated numbers of users for each role/task per unit of time over a given time period. This information will also be useful for subsequently building load profiles (see Section 4.2.4).

Gather Data

The data mentioned above can be gathered from a number of different sources:

- Conducting interviews or workshops with stakeholders, such as product owners, sales managers and (potential) end users. These discussions often reveal the principal operational profiles of users and provide answers to the fundamental question “Who is this application intended for”.
- Functional specifications and requirements (where available) are a valuable source of information about intended usage patterns which can also help identify user types and their operational profiles. Where functional specifications are expressed as user stories, the standard format directly enables types of users to be identified (i.e., As a *<type of user>*, I want *<some capability>* so that *<some benefit>).* Similarly, UML Use Case diagrams and descriptions identify the “actor” for the use case.
- Evaluating usage data and metrics gained from similar applications may support identification of user types and provide some initial indications of the expected numbers of users. Access to automatically monitored data (e.g., from a web master’s administration tool) is recommended. This will include monitoring logs

and data taken from usage of the current operational system where an update to that system is planned
- Monitoring the behavior of users when performing predefined tasks with the application may give insights into the types of operational profiles to be modeled for performance testing. Coordinating this task with any planned usability tests (especially if a usability lab is available) is recommended.

Construct Operational Profiles
The following steps are followed for identifying and constructing operational profiles for users:
- A top-down approach is taken. Relatively simple broad operational profiles are initially created and only broken down further if this is needed to achieve performance test objectives (see Section 4.1.1)
- Particular user profiles may be singled out as relevant for performance testing if they involve tasks which are executed frequently, require critical (high risk) or frequent transactions between different system components, or potentially demand large volumes of data to be transferred.
- Operational profiles are reviewed and refined with the principal stakeholders before being used for the creation of load profiles (see Section 4.2.4).

The system under test is not always subjected to loads imposed by the user. Operational profiles may also be required for performance testing of the following types of system (please note this list is not exhaustive):

Off-line Batch Processing Systems
The focus here lies principally on the throughput of the batch processing system (see Section 4.2.5) and its ability to complete within a given time period. Operational profiles focus on the types of processing which are demanded of the batch processes. For example, the operational profiles for a stock trading system (which typically includes online and batch-based transaction processing) may include those relating to payment transactions, verifying credentials, and checking compliance of legal conditions for particular types of stock transactions. Each of these operational profiles would result in different paths being taken through the overall batch process for a stock. The steps outlined above for identifying the operational profiles of online user-based systems can also be applied in the batch processing context.

Systems of Systems
Components within a multi-system (which may also be embedded) environment respond to different types of input from other systems or components. Depending on the nature of the system under test, this may require modeling of several different operational profiles to effectively represent the types of input provided by those supplier systems. This may involve detailed analysis (e.g., of buffers and queues) together with the system architects and based on system and interface specifications.

4.2.4 Creating Load Profiles

A load profile specifies the activity which a component or system being tested may experience in production. It consists of a designated number of instances that will perform the actions of predefined operational profiles over a specified time period. Where the instances are users, the term “virtual users” is commonly applied.

The principal information required to create a realistic and repeatable load profile is:

- The performance testing objective (e.g., to evaluate system behavior under stress loads)
- Operational profiles which accurately represent individual usage patterns (see Section 4.2.3)
- Known throughput and concurrency issues (see Section 4.2.5)
- The quantity and time distribution with which the operational profiles are to be executed such that the SUT experiences the desired load. Typical examples are:
    - Ramp-ups: Steadily increasing load (e.g., add one virtual user per minute)
    - Ramp-downs: Steadily decreasing load
    - Steps: Instantaneous changes in load (e.g., add 100 virtual users every five minutes)
    - Predefined distributions (e.g., volume mimics daily or seasonal business cycles)

The following example shows the construction of a load profile with the objective of generating stress conditions (at or above the expected maximum for a system to handle) for the system under test.

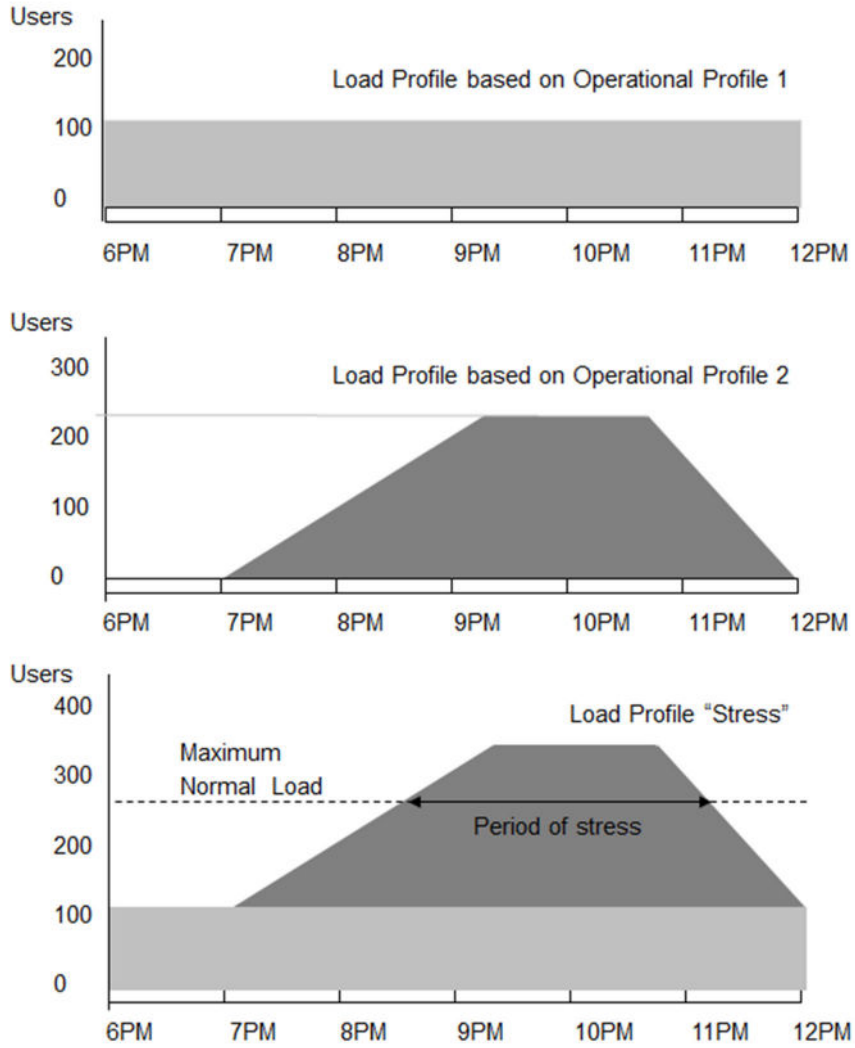

Diagram 1: Example of constructing a "stress" load profile

At the top of the diagram a load profile is shown which consists of a step input of 100 virtual users. These users perform the activities defined by Operation Profile 1 over the entire duration of the test. This is typical of many performance load profiles that represent a background load.

The middle diagram shows a load profile that consists of a ramp-up to 220 virtual users that is maintained for two hours before ramping down. Each virtual user performs activities defined in Operational Profile 2.

The lower diagram shows the load profile that results from the combination of the two described above. The system under test is subjected to a three-hour period of stress. For further examples, refer to [Bath14].

4.2.5 Analyzing Throughput and Concurrency

It is important to understand different aspects of workload: throughput and concurrency. To model operational and load profiles properly, both aspects should be taken into consideration.

System Throughput

System throughput is a measure of the number of transactions of a given type that the system processes in a unit of time. For example, the number of orders per hour or the number of HTTP requests per second. System throughput should be distinguished from network throughput, which is the amount of data moved over the network (Section 2.1).

System throughput defines load on the system. Unfortunately, quite often the number of concurrent users is used to define the load for interactive systems instead of throughput. This is partially true because that number is often easier to find, and partially because it is the way load testing tools define load. Without defining operational profiles – what each user is doing and how intensely (which also is throughput for one user) – the number of users is not a good measure of load. For example, if there are 500 users running short queries each minute, we have a throughput of 30,000 queries per hour. If the same 500 users are running the same queries, but one per hour, the throughput is 500 queries per hour. So there are the same 500 users, but a 60x difference between loads and at least a 60x difference in the hardware requirements for the system.

Workload modeling is usually done by considering the number of virtual users (execution threads) and the think time (delays between user actions). However, system throughput is also defined by processing time, and that time may increase as load increases.

System throughput = [number of virtual users] / ([processing time] + [think time])

So when the processing time increases, throughput may significantly decrease even if everything else stays the same.

System throughput is an important aspect when testing batch processing systems. In this case, the throughput is typically measured according to the number of transactions that can be accomplished within a given time frame (e.g., a nightly batch processing window).

Concurrency

Concurrency is a measure of the number of simultaneous / parallel threads of execution. For interactive systems, it may be a number of simultaneous / parallel users. Concurrency is usually modeled in load testing tools by setting the number of virtual users.

Concurrency is an important measure. It represents the number of parallel sessions, each of which may use its own resources. Even if throughput is the same, the amount of resources used may differ depending on concurrency. Typical test setups are closed systems (from the queuing theory point of view), where the number of users in the system is set (fixed population). If all users are waiting for the system's response in a closed system, no new users can arrive. Many public systems are open systems – new users are arriving all the time even if all the current users are waiting for the system's response.

4.2.6 Basic Structure of a Performance Test Script

A performance test script should simulate a user or component activity that contributes to the load on the system under test (which may be the whole system or one of its components). It initiates requests to the server in a proper order and at a given pace.

The best way to create performance test scripts depends on the load generation approach used (Section 4.1).

- The traditional way is to record communication between the client and the system or component on the protocol level and then play it back after the script has been parameterized and documented. The parameterization results in a scalable and maintainable script, but the task of parameterization may be time consuming.
- Recording at the GUI level typically involves capturing GUI actions of a single client with a test execution tool and running that script with the load generation tool to represent multiple clients.
- Programming may be done using protocol requests (e.g., HTTP requests), GUI actions, or API calls. In the case of programming scripts, the exact sequence of requests sent to and received from the real system must be determined, which may be not trivial.

Usually a script is one or several sections of code (written in a generic programming language with some extensions or in a specialized language) or an object, which may be presented to a user by the tool in a GUI. In both cases the script will include server requests creating load (e.g., HTTP requests) and some programming logic around them specifying how exactly these requests would be invoked (e.g., in what order, at what moment, with what parameters, what should be checked). The more sophisticated the logic, the more need for using powerful programming languages.

Overall Structure

Often the script has an initialization section (where everything gets prepared for the main part), main sections that may be executed multiple times, and a clean-up section (where necessary steps are taken to finish the test properly).

Data Collection
To collect response times, timers should be added to the script to measure how long a request or a combination of requests takes. The timed requests should match a meaningful unit of logical work–for example, a business transaction for adding an item to an order or submitting an order.

It is important to understand what exactly is measured: in the case of protocol-level scripts it is server and network response time only, while GUI scripts measure end-to-end time (although what exactly is measured depends on the technology used).

Result Verification and Error Handling
An important part of the script is result verification and error handling. Even in the best load testing tools, default error handling tends to be minimal (such as checking the HTTP request return code), so it is recommended to add additional checks to verify what the requests actually return. Also if any cleanup is required in case of an error, it likely will need to be implemented manually. A good practice is to verify that the script is doing what it is supposed to do using indirect methods–for example, checking the database to verify that the proper information was added.

Scripts may include other logic specifying rules concerning when and how server requests will be made. One example is setting synchronization points, which is done by specifying that the script should wait for an event at that point before proceeding. The synchronization points may be used to ensure that a specific action is invoked concurrently or to coordinate work between several scripts.

Performance testing scripts are software, so creating a performance testing script is a software development activity. It should include quality assurance and tests to verify that the script works as expected with the whole range of input data.

4.2.7 Implementing Performance Test Scripts

Performance test scripts are implemented based on the PTP and the load profiles. While technical details of implementation will differ depending on the approach and tool(s) used, the overall process remains the same. A performance script is created using an Integrated Development Environment (IDE) or script editor, to simulate a user or component behavior. Usually the script is created to simulate a specific operational profile (although it is often possible to combine several operational profiles in one script with conditional statements).

As the sequence of requests is determined, the script may be recorded or programmed depending on the approach. Recording usually ensures that it exactly simulates the real system, while programming relies on knowledge of the proper request sequence.

If recording on the protocol level is used, an essential step after recording in most cases is replacing all recorded internal identifiers that define context. These identifiers must be made into variables that can be changed between runs with appropriate values

that are extracted from the request responses (e.g., a user identifier that is acquired at login and must be supplied for all subsequent transactions). This is a part of script parameterization, sometimes referred to as ‘correlation’. In that context the word correlation has a different meaning than when used in statistics (where it means relationship between two or more things). Advanced load testing tools may do some correlation automatically, so it may be transparent in some cases—but in more complex cases, manual correlation or adding new correlation rules may be required. Incorrect correlation or lack of correlation is the main reason why recorded scripts fail to playback.

Running multiple virtual users with the same user name and accessing the same set of data (as usually happens during playback of a recorded script without any further modification beyond necessary correlation) is an easy way to get misleading results. The data could be completely cached (copied from disk to memory for faster access) and results would be much better than in production (where such data may be read from a disk). Using the same users and/or data can also cause concurrency issues (e.g., if data is locked when a user is updating it) and results would be much worse than in production as the software would wait for the lock to free before the next user could lock the data for update.

So scripts and test harnesses should be parameterized (i.e., fixed or recorded data should be replaced with values from a list of possible choices), so that each virtual user uses a proper set of data. The term “proper” here means different enough to avoid problems with caching and concurrency, which is specific for the system, data, and test requirements. This further parameterization depends on the data in the system and the way the system works with this data, so it usually is done manually, although many tools provide assistance here.

There are cases where some data must be parameterized for the test to work more than once–for example, when an order is created and the order name must be unique. Unless the order’s name is parameterized, the test will fail as soon as it tries to create an order with an existing (recorded) name.

To match operational profiles, think times should be inserted and/or adjusted (if recorded) to generate a proper number of requests / throughput as discussed in Section 4.2.5.

When scripts for separate operational profiles are created, they are combined into a scenario implementing the whole load profile. The load profile controls how many virtual users are started using each script, when, and with what parameters. The exact implementation details depend on the specific load testing tool or harness.

4.2.8 Preparing for Performance Test Execution

The main activities for preparing to execute the performance tests include:

- Setting up the system under test
- Deploying the environment
- Setting up the load generation and monitoring tools and making sure that all the necessary information will be collected

It is important to ensure the test environment is as close to the production environment as possible. If this is not possible, then there must be a clear understanding of the differences and how the test results will be projected on the production environment. Ideally, the true production environment and data would be used, but testing in a scaled-down environment still may help mitigate a number of performance risks.

It is important to remember that performance is a non-linear function of the environment, so the further the environment is from production standard, the more difficult it becomes to make accurate projections for production performance. The lack of reliability of the projections and the increased risk level grow as the test system looks less like production.

The most important parts of the test environment are data, hardware and software configuration, and network configuration. The size and structure of the data could affect load test results dramatically. Using a small sample set of data or a sample set with a different data complexity for performance tests can give misleading results, particularly when the production system will use a large set of data. It is difficult to predict how much the data size affects performance before real testing is performed. The closer the test data is to the production data in size and structure, the more reliable the test results will be.

If data is generated or altered during the test, it may be necessary to restore the original data before the next test cycle to ensure that the system is in the proper state.

If some parts of the system or some of the data is unavailable for performance tests for whatever reason, a workaround should be implemented. For example, a stub may be implemented to replace and emulate a third party component responsible for credit card processing. That process is often referred to as “service virtualization” and there are special tools available to assist with that process. The use of such tools are highly recommended to isolate the system under test.

There are many ways to deploy environments. For example, options may include using any of the following:

- Traditional internal (and external) test labs
- Cloud as an environment using Infrastructure as a Service (IaaS), when some parts of the system or all of the system is deployed to the cloud
- Cloud as an environment using Software as a Service (SaaS), when vendors provide the load testing service

Depending on the specific goals and the systems to test, one test environment may be preferred over another. For example,

- To test the effect of a performance improvement (performance optimization), using an isolated lab environment may be a better option to see even small variations introduced by the change.
- To load test the whole production environment end-to-end to make sure the system will handle the load without any major issues, testing from the cloud or a service may be more appropriate. (Note that this only works for SUTs that can be reached from a cloud).
- To minimize costs when performance testing is limited in time, creating a test environment in the cloud may be a more economical solution.

Whatever approach to deployment is used, both hardware and software should be configured to meet the test objective and plan. If the environment matches production, it should be configured in the same way. However, if there are differences, the configuration may have to be adjusted to accommodate these differences. For example, if test machines have less physical memory than the production machines, software memory parameters (such as Java heap size) may need to be adjusted to avoid memory paging.

Proper configuration / emulation of the network is important for global and mobile systems. For global systems (i.e., one which has users or processing distributed world-wide) one of approaches may be to deploy load generators in places where users are located. For mobile systems network emulation remains the most viable option due to the variances in the network types that can be used. Some load testing tools have built-in network emulation tools and there are standalone tools for network emulation.

The load generation tools should be properly deployed and the monitoring tools should be configured to collect all necessary metrics for the test. The list of metrics depends on the test objectives, but it is recommended to collect at least basic metrics for all tests (see Section 2.1.2).

Depending on the load, specific tool / load generation approach, and machine configuration, more than one load generation machine may be needed. To verify the setup, machines involved in load generation should be monitored too. This will help avoid a situation where the load is not maintained properly because one of the load generators is running slowly.

Depending on the setup and tools used, load testing tools need to be configured to create the appropriate load. For example, specific browser emulation parameters may be set or IP spoofing (simulating that each virtual user has a different IP address) may be used.

Before tests are executed, the environment and setup must be validated. This is usually done by conducting a controlled set of tests and verifying the outcome of the

tests as well as checking that the monitoring tools are tracking the important information.

To verify that the test works as designed, a variety of techniques may be used, including log analysis and verifying database content. Preparing for the test includes checking that required information gets logged, the system is in the proper state, etc. For example, if the test changes the state of the system significantly (add / change information in database), it may be necessary to return the system to the original state before repeating the test.

4.3 Execution

Performance test execution involves generation of a load against the SUT according to a load profile (usually implemented by performance testing scripts invoked according to a given scenario), monitoring all parts of the environment, and collecting and keeping all results and information related to the test. Usually advanced load testing tools / harnesses perform these tasks automatically (after, of course, proper configuration). They generally provide a console to enable performance data to be monitored during the test and permit necessary adjustments to be made (see Section 5.1). However, depending on the tool used, the SUT, and the specific tests being executed, some manual steps may be needed.

Performance tests are usually focused on a steady state of the system, i.e., when the system's behavior is stable. For example, when all simulated users / threads are initiated and are performing work as designed. When the load is changing (for example, when new users are added), the system's behavior is changing and it becomes more difficult to monitor and analyze test results. The stage of getting to the steady state is often referred to as the ramp-up, and the stage of finishing the test is often referred to as the ramp-down.

It is sometimes important to test transient states, when the system's behavior is changing. This may apply, for example, to the concurrent logging of a large number of users or spike tests. When testing transient states it is important to understand the need for careful monitoring and analysis of the results, as some standard approaches—such as monitoring averages—may be very misleading.

During the ramp-up it is advisable to implement incremental load states to monitor the impact of the steadily increasing load on the system's response. This ensures that sufficient time is allocated for the ramp-up and that the system is able to handle the load. Once the steady state has been reached, it is a good practice to monitor that both the load and the system's responses are stable and that random variations (which always exist) are not substantial.

It is important to specify how failures should be handled to make sure that no system issues are introduced. For example, it may be important for the user to logout when a failure occurs to ensure that all resources associated with that user are released.

If monitoring is built into the load testing tool and it is properly configured, it usually starts at the same time as the test execution. However, if stand-alone monitoring tools are used, monitoring should be started separately and the necessary information collected such that subsequent analysis can be carried out together with the test results. The same is true for log analysis. It is essential to time-synchronize all tools used, so that all information related to a specific test execution cycle can be located.

Test execution is often monitored using the performance test tool's console and real-time log analysis to check for issues and errors in both the test and the SUT. This helps to avoid needlessly continuing with running large-scale tests, which might even impact other systems if things go wrong (e.g., if failure occur, components fail or the generated loads are too low or high). These tests can be expensive to run and it may be necessary to stop the test or make some on-the-fly adjustments to the performance test or the system configuration if the test deviates from the expected behavior.

One technique for verifying load tests which are communicating directly on the protocol level is to run several GUI-level (functional) scripts or even to execute similar operational profiles manually in parallel to the running load test. This checks that response times reported during the test only differ from the response times measured manually at the GUI level by the time spent on the client side.

In some cases when running performance testing in an automated way (for example, as a part of Continuous Integration, as discussed in Section 3.4) checks must be done automatically, since manual monitoring and intervention may not be possible. In this case, the test set up should be able to recognize any deviations or problems and issue an alert (usually while properly completing the test). This approach is easier to implement for regression performance tests when the system's behavior is generally known, but may be more difficult with exploratory performance tests or large-scale expensive performance tests that may need adjustments to be made dynamically during the test.

4.4 Analyzing Results and Reporting

Section 4.1.2 discussed the various metrics in a performance test plan. Defining these up front determines what must be measured for each test run. After completion of a test cycle, data should be collected for the defined metrics.

When analyzing the data it is first compared to the performance test objective. Once the behavior is understood, conclusions can be drawn which provide a meaningful summary report that includes recommended actions. These actions may include

changing physical components (e.g., hardware, routers), changing software (e.g., optimizing applications and database calls), and altering the network (e.g., load balancing, routing).

The following data is typically analyzed:

- **Status of simulated (e.g., virtual) users**. This needs to be examined first. It is normally expected that all simulated users have been able to accomplish the tasks specified in the operational profile. Any interruption to this activity would mimic what an actual user may experience. This makes it very important to first see that all user activity is completed since any errors encountered may influence the other performance data.
- **Transaction response time**. This can be measured in multiple ways, including minimum, maximum, average, and a percentile (e.g., 90th). The minimum and maximum readings show the extremes of the system performance. The average performance is not necessarily indicative of anything other than the mathematical average and can often be skewed by outliers. The 90th percentile is often used as a goal since it represents the majority of users attaining a specific performance threshold. It is not recommended to require 100% compliance with the performance objectives as the resources required may be too large and the net effect to the users will often be minor.
- **Transactions per second**. This provides information on how much work was done by the system (system throughput).
- **Transaction failures**. This data is used when analyzing transactions per second. Failures indicate the expected event or process did not complete, or did not execute. Any failures encountered are a cause for concern and the root cause must be investigated. Failed transactions may also result in invalid transactions per second data since a failed transaction will take far less time than a completed one.
- **Hits (or requests) per second**. This provides a sense of the number of hits to a server by the simulated users during each second of the test.
- **Network throughput.** This is usually measured in bits by time interval, as in bits per second. This represents the amount of data the simulated users receive from the server every second. (see Section 4.2.5)
- **HTTP responses.** These are measured per second and include possible response codes such as: 200, 302, 304, 404, the latter indicating that a page is not found.

Although much of this information can be presented in tables, graphical representations make it easier to view the data and identify trends.

Techniques used in analyzing data can include:

- Comparing results to stated requirements
- Observing trends in results
- Statistical quality control techniques
- Identifying errors
- Comparing expected and actual results
- Comparing the results to prior test results
- Verifying proper functioning of components (e.g., servers, networks)

Identifying correlation between metrics can help us understand at what point system performance begins to degrade. For example, what number of transactions per second were processed when the CPU reached 90% capacity and the system slowed?

Analysis can help identify the root cause of the performance degradation or failure, which in turn will facilitate correction. Confirmation testing will help determine if the corrective action addressed the root cause.

Reporting

Analysis results are consolidated and compared against the objectives stated in the performance test plan. These may be reported in the overall test status report together with other test results, or included in a dedicated report for performance testing. The level of detail reported should match the needs of the stakeholders. The recommendations based on these results typically address software release criteria (including target environment) or required performance improvements.

A typical performance testing report may include:

Executive Summary

This section is completed once all performance testing has been done and all results have been analyzed and understood. The goal is to present concise and understandable conclusions, findings, and recommendations for management with the goal of an actionable outcome.

Test Results

Test results may include some or all of the following information:

- A summary providing an explanation and elaboration of the results.
- Results of a baseline test that serves as “snapshot” of system performance at a given time and forms the basis of comparison with subsequent tests. The results should include the date/time the test started, the concurrent user goal, the throughput measured, and key findings. Key findings may include overall error rate measured, response time and average throughput.
- A high-level diagram showing any architectural components that could (or did) impact test objectives.

- A detailed analysis (tables and charts) of the test results showing response times, transaction rates, error rates and performance analysis. The analysis also includes a description of what was observed, such as at what point a stable application became unstable and the source of failures (e.g., web server, database server).

Test Logs/Information Recorded

A log of each test run should be recorded. The log typically includes the following:

- Date/time of test start
- Test duration
- Scripts used for test (including script mix if multiple scripts are used) and relevant script configuration data
- Test data file(s) used by the test
- Name and location of data/log files created during test
- HW/SW configuration tested (especially any changes between runs)
- Average and peak CPU and RAM utilization on web and database servers
- Notes on achieved performance
- Defects identified

Recommendations

Recommendations resulting from the tests may include the following:

- Technical changes recommended, such as reconfiguring hardware or software or network infrastructure
- Areas identified for further analysis (e.g., analysis of web server logs to help identify root causes of issues and/or errors)
- Additional monitoring required of gateways, servers, and networks so that more detailed data can be obtained for measuring performance characteristics and trends (e.g., degradation)

5. Tools – 90 mins.

Keywords

load generator, load management, monitoring tool, performance testing tool

Learning Objectives

5.1 Tool Support
PTFL-5.1.1 (K2) Understand how tools support performance testing

5.2 Tool Suitability
PTFL-5.2.1 (K4) Evaluate the suitability of performance testing tools in a given project scenario

5.1 Tool Support

Performance testing tools include the following types of tool to support performance testing.

Load Generators
The generator, through an IDE, script editor or tool suite, is able to create and execute multiple client instances that simulate user behavior according to a defined operational profile. Creating multiple instances in short periods of time will cause load on a system under test. The generator creates the load and also collects metrics for later reporting.

When executing performance tests the objective of the load generator is to mimic the real world as much as is practical. This often means that user requests coming from various locations are needed, not just from the testing location. Environments that are set up with multiple points of presence will distribute where the load is originating from so that it is not all coming from a single network. This provides realism to the test, though it can sometimes skew results if intermediate network hops create delays.

Load Management Console
The load management console provides the control to start and stop the load generator(s). The console also aggregates metrics from the various transactions that are defined within the load instances used by the generator. The console enables reports and graphs from the test executions to be viewed and supports results analysis.

Monitoring Tool
Monitoring tools run concurrently with the component or system under test and supervise, record and/or analyze the behavior of the component or system. Typical components which are monitored include web server queues, system memory and disk space. Monitoring tools can effectively support the root cause analysis of performance

degradation in a system under test and may also be used to monitor a production environment when the product is released. During performance test execution monitors may also be used on the load generator itself.

License models for performance test tools include the traditional seat/site based license with full ownership, a cloud-based pay-as-you-go license model, and open source licenses which are free to use in a defined environment or through cloud-based offerings. Each model implies a different cost structure and may include ongoing maintenance. What is clear is that for any tool selected, understanding how that tool works (through training and/or self-study) will require time and budget.

5.2 Tool Suitability

The following factors should be considered when selecting a performance testing tool:

Compatibility
In general a tool is selected for the organization and not only for a project. This means considering the following factors in the organization:

- Protocols: As described in Section 4.2.1, protocols are a very important aspect to performance tool selection. Understanding which protocols a system uses and which of these will be tested will provide necessary information in order to evaluate the appropriate test tool.
- Interfaces to external components: Interfaces to software components or other tools may need to be considered as part of the complete integration requirements to meet process or other inter-operability requirements (e.g., integration in the CI process).
- Platforms: Compatibility with the platforms (and their versions) within an organization is essential. This applies to the platforms used to host the tools and the platforms with which the tools interact for monitoring and/or load generation.

Scalability
Another factor to consider is the total number of concurrent user simulations the tool can handle. This will include several factors:

- Maximum number of licenses required
- Load generation workstation/server configuration requirements
- Ability to generate load from multiple points of presence (e.g., distributed servers)

Understandability
Another factor to consider is the level of technical knowledge needed to use the tool. This is often overlooked and can lead to unskilled testers incorrectly configuring tests, which in turn provide inaccurate results. For testing requiring complex scenarios and a high level of programmability and customization, teams should ensure that the tester has the necessary skills, background, and training.

Monitoring
Is the monitoring provided by the tool sufficient? Are there other monitoring tools available in the environment that can be used to supplement the monitoring by the tool? Can the monitoring be correlated to the defined transactions? All of these questions must be answered to determine if the tool will provide the monitoring required by the project.

When monitoring is a separate program/tools/whole stack then it can be used to monitor production environment when the product is released.

6. References

6.1 Standards

- [ISO25000] ISO/IEC 25000:2005, Software Engineering - Software Product Quality Requirements and Evaluation (SQuaRE)

6.2 ISTQB Documents

- [ISTQB_UT_SYL] ISTQB Foundation Level Usability Testing Syllabus, Version 2018
- [ISTQB_ALTA_SYL] ISTQB Advanced Level Test Analyst Syllabus, Version 2012
- [ISTQB_ALTTA_SYL] ISTQB Advanced Level Technical Test Analyst Syllabus, Version 2012
- [ISTQB_ALTM_SYL] ISTQB Advanced Level Test Manager Syllabus, Version 2012
- [ISTQB_FL_SYL] ISTQB Foundation Level (Core) Syllabus, Version 2018
- [ISTQB_FL_AT] ISTQB Foundation Level Agile Tester Syllabus, Version 2014
- [ISTQB_GLOSSARY] ISTQB Glossary of Terms used in Software Testing, http://glossary.istqb.org

6.3 Books

[Anderson01] Lorin W. Anderson, David R. Krathwohl (eds.) “A Taxonomy for Learning, Teaching and Assessing: A Revision of Bloom’s Taxonomy of Educational Objectives”, Allyn & Bacon, 2001, ISBN 978-0801319037

[Bath14] Graham Bath, Judy McKay, “The Software Test Engineer’s Handbook”, Rocky Nook, 2014, ISBN 978-1-933952-24-6

[Molyneaux09] Ian Molyneaux, “The Art of Application Performance Testing: From Strategy to Tools”, O’Reilly, 2009, ISBN: 9780596520663

[Microsoft07] Microsoft Corporation, “Performance Testing Guidance for Web Applications”, Microsoft, 2007, ISBN: 9780735625709

7. Index