Pre-Production Testing
Best Practices & Guidance
# Revision and Signoff Sheet

## Change Record

<table>
<thead>
<tr>
<th>Date</th>
<th>Author</th>
<th>Version</th>
<th>Change reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>.1</td>
<td>Initial draft for review/discussion</td>
</tr>
</tbody>
</table>

## Reviewers

<table>
<thead>
<tr>
<th>Name</th>
<th>Version approved</th>
<th>Position</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table of Contents

1 Introduction ................................................................................................................................. 1
  1.1 Purpose ................................................................................................................................. 1
  1.2 Scope .................................................................................................................................. 1
  1.3 Overview ............................................................................................................................... 1

2 Testing Environments .................................................................................................................. 2
  2.1 Overview ............................................................................................................................... 2
  2.2 Recommended Configuration for Solutions ........................................................................... 3
  2.3 Pre-Production / Test Environment ..................................................................................... 4
    2.3.1 Description ....................................................................................................................... 4
    2.3.2 Hardware Configuration .................................................................................................... 4
    2.3.3 Software Configuration .................................................................................................... 4
  2.4 Staging Environment ............................................................................................................. 5
    2.4.1 Description ....................................................................................................................... 5
    2.4.2 Hardware .......................................................................................................................... 5
    2.4.3 Software .......................................................................................................................... 5

3 Performance Testing .................................................................................................................. 7
  3.1 Overview ............................................................................................................................... 7
    3.1.1 What is Performance Testing? .......................................................................................... 7
    3.1.2 How to accomplish performance testing? ......................................................................... 7
    3.1.3 When Does It Occur? ....................................................................................................... 7
    3.1.4 Who is Involved? .............................................................................................................. 8
    3.1.5 What Kinds of Performance Tests Are There? ................................................................. 8
    3.1.6 What Can It Do For Me? .................................................................................................. 9
  3.2 Testing Standards ................................................................................................................... 9
    3.2.1 Test Phase Details ........................................................................................................... 11
  3.3 Critical Success Factors ......................................................................................................... 13
  3.4 Performance Profile Test Strategy ....................................................................................... 14
    3.4.1 Do’s & Don’ts ................................................................................................................... 14
    3.4.2 10-step Performance Profile Test guide ......................................................................... 14
  3.5 Load Test Strategy ................................................................................................................ 15
    3.5.1 Do’s & Don’ts ................................................................................................................... 15
    3.5.2 10-step Load Test Guide ............................................................................................... 16
  3.6 Stress Test Strategy ............................................................................................................... 17
    3.6.1 Do’s & Don’ts ................................................................................................................... 17
    3.6.2 5-step Stress Test Guide ................................................................................................. 17
  3.7 Volume Test Strategy ............................................................................................................ 19
    3.7.1 Do’s & Don’ts ................................................................................................................... 19
    3.7.2 5-step Volume Test guide ............................................................................................. 19
3.8 Fault Tolerance Test Strategy .................................................................................................................. 20
3.8.1 Do's & Don'ts .......................................................................................................................................... 20
1 INTRODUCTION

Past experience in XXXXX and the industry as a whole has proven that moving from development environment to production environment or is not a trivial task. Getting a system into production is a multi-dimensional problem that mainly deals with the following aspects

- Setting up Testing Environments
- Performing Functional Tests
- Performing Performance Tests
- Transitioning between Environments (both between the different testing environments and from testing to production)

The key for successful deployment (save writing a bug-less product) is careful planning and execution of the testing while establishing environments and test cases representing the real-world.

1.1 Purpose

The pre-deployment testing guidelines are designed to give the QA team a procedural and tactical vision into the need, planning, execution, and analysis of the different stages on the road to a successful deployment relying on industry and MS tactics and best practices.

1.2 Scope

This document is intended to provide guidance to a QA test team responsible for the creation and execution of automated tests.

The document assumes functional testing is already a working practice within XXXXX, and thus focuses on additional aspects of the stabilization effort namely, setting the current environments, procedures for transition between environments and performance testing.

The guidelines are a set of recognized industry-standard practices and procedures intended to provide project-level guidance to testers.

1.3 Overview

The document is divided into 2 main parts. The first part deals with testing environments (and environment transitions) and the second part deals with the different aspects of performance testing.

The Testing Environments section provides an overview of the different environments for the complete development cycle (as well as recommended setups for different project sizes). It then follows with recommendations for setting testing and staging environments.

The Performance Test Guidelines are organized into an overview of the “who, what, when, where, why” questions of performance testing followed by methodologies and considerations to execute the various types of tests. A list of proposed standards, measures, and metrics is included after the test types followed by lists of the do/don't do rules easing a successful execution of performance testing projects.
2 TESTING ENVIRONMENTS

2.1 Overview

The ways in which an application is exercised at the various stages of its life cycle and deployment schedule require several different parallel instantiations of the application. These instantiations or environments go by many names in different organizations, but the following names are commonly used:

- The **development environment** is where unit level development is done. Therefore, software and data structures tend to be volatile in this environment. It is here that the developer is at liberty to modify the application module under development and its test environment to suit unit level development needs. In this environment, developers typically work solely on development workstations where they often have full administrative rights. The development environment is a “sandbox” environment where developers are free to use various application infrastructure elements without the constraints, for instance, of the security that will exist in other environments. This allows developers to focus on building application logic and learning how best to use the various application infrastructure elements available without limitations imposed by the environment.

- The **integration environment** is where application units (software modules, data schemas, and data content) are first assembled and then tested as an integrated suite. This environment is also volatile but is much less so than the development environment. The objective here is to force coherence among the independently developed modules or schemas. This is typically an environment where developers do not have all the permissions that they had in the development environment. As a result, this is often the first time that issues such as installation, configuration, and security requirements for the eventual target infrastructure are addressed.

- The **test environment** is where a “release candidate” grade version of the application is run through testing exercises. It is as tightly controlled as the production environment and also substantially less volatile than integration. The objective here is to assume relative stability of the integrated application and test its stability, correctness, and performance. This environment is usually off limits to developers. Deployment and updates to applications in this environment are controlled by the test team and are often done by a member of the build or infrastructure teams.

- The **staging environment** is where an application resides after it has been fully tested in the test environment. It provides a convenient location from which to deploy to the final production environment. Because the staging environment is often used to perform final tests and checks on application functionality, it should resemble the production environment as closely as possible. For example, the staging environment should not only have the same operating systems and applications installed as the production computers, but it should also have a similar network topology (which the testing environment might not have). Usually, the staging network environment mimics the production environment in all respects except that it is a scaled down version (for example, it may have fewer cluster members or fewer processors than your production servers).

- The **production environment** is where the application is actually used by an organization; it is the least volatile and most tightly controlled.
XXXXX, already has the notion of most if not all of these environments. One point that should be noted is the difference in responsibility between the Testing and Staging environments. The Testing environment is under the responsibility of the QA team, while the Staging environment is under the responsibility of the Infrastructure team (whose responsibility, as mentioned above, is to make this environment as close as possible to the production environment).

### 2.2 Recommended Configuration for Solutions

The following table shows the recommended machine configurations for different sized projects.

<table>
<thead>
<tr>
<th>Project Size</th>
<th>Environments Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>Build Machine / Source Control Server / Release Server (Shared)</td>
</tr>
<tr>
<td></td>
<td>Development Environment / Test Environment (Shared)</td>
</tr>
<tr>
<td></td>
<td>Production Environment</td>
</tr>
<tr>
<td>Medium</td>
<td>Build Machine / Release Server (Shared)</td>
</tr>
<tr>
<td></td>
<td>Source Control Server</td>
</tr>
<tr>
<td></td>
<td>Minimal Environment</td>
</tr>
<tr>
<td></td>
<td>Development Environment / Test Environment (Shared)</td>
</tr>
<tr>
<td></td>
<td>Staging Environment</td>
</tr>
<tr>
<td></td>
<td>Production Environment</td>
</tr>
<tr>
<td>Large</td>
<td>Build Machine</td>
</tr>
<tr>
<td></td>
<td>Release Server</td>
</tr>
<tr>
<td></td>
<td>Source Control Server</td>
</tr>
<tr>
<td></td>
<td>Minimal Environment</td>
</tr>
<tr>
<td></td>
<td>Development Environment</td>
</tr>
<tr>
<td></td>
<td>Test Environment (multiple if very large)</td>
</tr>
<tr>
<td></td>
<td>Staging Environment</td>
</tr>
</tbody>
</table>
2.3 Pre-Production / Test Environment

2.3.1 Description
This is a multi-box configuration that is used by the Test team for running test cases and regressing tests. It should have the same configuration as the Staging Environment, except for the debugging tools and modifications needed on the Staging Environment. This Environment is owned by the Test team and is used for running test cases on a multi-machine environment. It can be used for long duration "soak" testing, performance testing and functional testing.

It is extremely important that this environment will have a configuration as close as possible to the production environment. Having an exact replica of the deployed environment greatly helps ensuring the deployment is correct and repeatable when the system will go live.

Another critical aspect for the effectiveness of this environment is the ability to define the various "user profile" – describing in details areas such as: The number of users in the various groups of user types, the various common activities of a typical user (of each of the groups), etc.

It is extremely important to have the test tools configured to resemble to users' activities, not only in the aspects of the tested application, but also as a complete user profile (e.g. Performing various Microsoft Office activities while working with the application being tested, etc.).

2.3.2 Hardware Configuration
The Test environment should include the following components:

- **Servers** – There should be (at least) the various types of servers in this environment, each built with the exact hardware (type and size) as in the production environment. This is crucial for testing hardware related components (drivers, etc.) as well as conducting stress tests which could resemble, as close as possible, the various users activities along with the hardware and software response expected in the production environment.

- **Workstations** – There should be the most common types of workstations which are currently working in production. It is extremely important to focus on the "slowest" workstations which could be found in the production environment (as long as their percentage is high enough in relative to the total number of workstations).

  - There should be at least another workstation (preferably one for each type of workstation) which will continue to work with the production version of the code, acting as a baseline for the test environment, and providing a "watchdog" for possible changes in the performance of the environment as a whole, which are not related at all to the changes in the version of the application being tested in this environment.

- **Networking devices** - It is important to use the same networking devices as in the projection system in the test environment, to allow simulating the production environment as closely as possible - for the same reasons as described in the servers' part above.

2.3.3 Software Configuration

- **The Application** (System under test) – installing the application to the test environment should be done using the procedures used to deploy the application to end-users

- **Development / Debug tools** – some of the machines in the testing environment can contain the development environment and / or any other debugging tools used in the project. It is highly recommended though, to have most of the machines in configuration that is as close as possible to the target's configuration. Since the bulk of the functional
testing will occur at the test environment it is also permissible to use remote debugging tools.

- **Monitoring tools** – including tools such as windows Performance Monitoring tool (Perfmon), SNMP monitors etc. should work (if possible) on machines separate from the machines that participate in the test itself.

- **Test management tools** – tools such as Load Runner from Mercury or TestComplete from Automated QA – the management tools should work on separate machines, but it may be needed to install agents on the different machines under test (usually GUI machines) to allow the automated testing.

- **Defect tracking tools** – such as Test Director or ClearQuest should not reside on machines that participate in the testing itself.

## 2.4 Staging Environment

### 2.4.1 Description

The Staging environment is used to test the application in a production like environment with the addition of debugging and testing tools. This environment will mimic production, i.e. it will use the actual hardware that will be used in the production environment and will use the exact same configuration (e.g. clustering, software etc.) as the production environment will have.

The Staging environment is the last stop, or the last chance to verify that the code is indeed production worthy. It is therefore even more important to configure it correctly (compared with the Testing Environment detailed above).

**Important**

Unless noted otherwise, he points in sections 2.3.2 and 2.3.3 regarding the testing environment applies to the staging environment as well. Additional pointers and recommendations as well as the differences between the environments are listed below.

### 2.4.2 Hardware

- **Network** – Not only is it important to use the same equipment that is used in the production environment – it is also important to have the same networking loads and behavior that the production environment has. This can be achieved by use of appliances (such as Shunra's StormAppliance) and traffic generators (such as Ixia's Chariot). Using such tools will allow to simulate loads generated by other applications (that are also deployed on the users machines).

- **Workstations** – It is recommended to have at least one of each type of target workstations. This will allow building a profile of the application usage on the target platforms and understanding the limitations of the system.

### 2.4.3 Software

There are several different software aspects between the staging environment and the testing environment:

- No "Debug" mode code is used

- No Development tools. Additionally use of debugging tools is also limited – see below
Try to install any test supporting tool (i.e. testing related software that is not the application) on separate machines from the machines used to run the tests themselves whenever possible.

Unfortunately, the system under test is most likely not the only application that will run on the end-users machine. It is probably worthwhile to install common software used by end-users and verify the way the system works with the additional load.

2.4.3.1 Production Debugging

The staging environment should be used mainly to conduct performance and load tests. When you encounter a lot of functionality or stability bugs you should consider reverting to the testing or development environments to further investigate these issues, however occasionally there is a need to pinpoint a bug in the staging/production environments. In this cases follow Microsoft’s guidelines for production debugging (which can be found at [http://msdn.microsoft.com/library/en-us/dnbda/html/DBGrm.asp](http://msdn.microsoft.com/library/en-us/dnbda/html/DBGrm.asp)).
3 PERFORMANCE TESTING

3.1 Overview

3.1.1 What is Performance Testing?
The purpose of performance testing a solution is to:

- Optimize the configuration of hardware, software and network environments of a system under anticipated user and/or transaction load simulations.
- Optimize application performance by identifying system bottleneck symptoms and their root causes.
- Reduce the risk of deploying a low-performance system that does not meet user/system requirements.

3.1.2 How to accomplish performance testing?

- Document Clear Requirements and Objectives
  - All project team members should agree to each question performance testing will answer, before the tests can occur (i.e. Non-functional requirements, Solution Requirements Specifications).

- Be Realistic
  - Subject the system under test to conditions that match production as closely as possible (identical hardware and software configurations, network infrastructure with all layers of firewalls, switches, and proxy servers, and user access methods to correlate application issues to user experience).

- Be Aware
  - Monitor every possible component of the system and collect pertinent data.
  - Subject matter experts analyze the collected data for trends and opportunities to optimize.

- Provide Useful Analysis and Recommendations
  - Must improve end user experience (response time, reliability, scalability, etc…)
  - Must be based on value-driven cost-benefit analysis.

3.1.3 When Does It Occur?

- Performance testing is just one component of a fully comprehensive system test plan complementing the Solutions Delivery Life Cycle (SDLC) which includes unit, functional, integration, certification, regression, performance, disaster recovery (fault-tolerance) and user acceptance testing.

- Successful performance testing is dependent on successful completion of unit, functional, integration, and certification testing to create a stable working environment to stress. Once a functional use case (i.e. System Logon) has successfully passed regression testing that use case can be benchmarked and stress tested as a specific test case. (However, the results are to be taken with a grain of salt as stress tests, by definition “induce unreasonable stress on a particular component to find its independent breaking point”. In general, a valuable volume test includes multiple types of transactions and is best executed when at least 80% of the functional release’s code has successfully passed regression testing.)
Volume testing can be conducted in parallel with disaster recovery testing to generate a load on the system while simulating system disasters (unplug server/NIC, kill process, etc.).

User Acceptance testing usually is executed post-performance testing to provide the final signoff.

### 3.1.4 Who is Involved?

- **Business Owner/BSA**
  - Provides use case requirements and non-functional specifications (end-user response, availability requirements, etc) in measurable language.

- **QA/Performance Testing**
  - Configuration Management, Regression testing, Certification testing, Performance testing

- **System Solution Representatives** – who support the infrastructure or function of the solution
  - Subject Matter Experts – define company network utilization and system scalability requirements, as well as perform analysis on collected data.
  - Core Team: Project Managers, Requirements Analysts, Solution Architects, and Developers.
  - Infrastructure Services: Network Engineering, Database Administration, Network Security

- **Vendors**

### 3.1.5 What Kinds of Performance Tests Are There?

The first four types of tests (Benchmark, Contention, Profile, & Load) constitute a basic Performance test that will answer the basic questions set forth in the Test Phase Details Elicit Requirements section (3.2.1.2). The next four tests (Stress, Volume, Burn-in, & Recovery) are “extreme risk” mitigation methods and their necessity should be evaluated on a case by case basis.

- **Benchmark Testing:** Measures the performance of new/unknown target-of-test to a known reference standard such as: existing software or system measurement(s). This would be a test that documents a performance baseline for a single user/transaction executed throughout the end-to-end system.

- **Contention Testing:** Verifies the target-of-test can acceptably handle multiple actor demands on the same resource (data records, memory, I/O, etc.) as a part of Performance Profiling and Load Testing.

- **Performance Profiling:** Verifies the acceptability of the target-of-test's performance behavior under varying system configurations with constant operational conditions. This is used to determine optimal configuration for the components of the solution in a distributed n-tier system.

- **Load Testing:** Verifies the acceptability of the target-of-test’s performance behavior under varying operational conditions (number of users/transactions) with a constant system configuration. As the converse of performance profiling, this is used to determine the end-user experience under varying loads. This is the meat of Performance testing

- **Stress Testing:** Verifies the acceptability of the target-of-test’s performance behavior under abnormal or extreme conditions (diminished resource allocation, high-volume user/transaction conditions). This testing is useful for determining component breaking points outside of normal conditions
• **Volume Testing:** Verifies the acceptability of the target-of-test's performance behavior under varying operational conditions with a constant configuration and large amounts of data to determine if limits are reached that cause the software to fail. Volume testing also identifies the continuous maximum load or volume the target-of-test can handle for a given period. (For example, if the target-of-test is processing a set of database records to generate a report, a Volume Test would use a large test database, and would check that the software behaved normally and produced the correct report in an acceptable amount of time).

• **Burn-in Testing:** Verifies the acceptability of the target-of-test’s performance behavior under varying operational conditions with a constant configuration, but over a long period of time (24+ hours). Specifically used to pinpoint gradual drains on system resources such as memory leaks.

• **Failover and Recovery Testing:** Ensures that the target-of-test can successfully failover and recover from a variety of hardware, software or network malfunctions without undue loss of data or data integrity. For those systems that must be kept running, failover testing ensures that, when a failover condition occurs, the alternate or backup systems properly “take over” for the failed system without any loss of data or transactions. Recovery testing is an antagonistic test process in which the application or system is exposed to extreme or unlikely conditions, to cause a failure, such as device I/O failures, or invalid database keys. Recovery processes are invoked, and the application or system is monitored and inspected to verify proper application, system, and data recovery has been achieved.

### 3.1.6 What Can It Do For Me?

• Ensures up-front understanding of system/solution performance characteristics.

• Provides an effective mechanism to establish service level agreements (SLA) between the business and IT solution providers, in terms of system or user response times and system workload capabilities.

• Provides benchmarks to the Capacity Planning, QA, and other IT support organizations to monitor the system effectively, post-implementation.

• Reduce the risk of deploying system/solution that cannot meet the needs of the user (in terms of response time or availability).

• Reduce risk of incorrectly sizing hardware for highly scalable solutions, including *over* configuration and *under* configuration. – Both of these errors result in poor use of project budget/estimation.

• Provides Accountability and Requirements Trace-ability by equating specific performance tests to specific non-functional requirements that can result in Service Level Agreements that document expected system behavior under expected system conditions.

• Highly recommended for widely-implemented, high-volume, high-risk, high profile systems/solutions.

### 3.2 Testing Standards
Figure 2: High-Level QA Process Flow Model
3.2.1 Test Phase Details

3.2.1.1 Requirements Gathering

**Kickoff Meeting** - The members from QA, IT Support, Network Engineering, Data Security, Database Administration, Development, Capacity Planning, Architecture, Project Management, and Customer Input that are involved should be assembled to acclimate all parties to the Performance Testing process.

**Elicit Requirements** - During requirements gathering, the performance requirements for the system may not be as explicitly stated as the functional design requirements. They are often gleaned from the performance characteristics/ capabilities of the current system, or if one is not available, an industry standard benchmark may be proposed as an initial goal. Clear Objectives and Definitions must be agreed upon at this time, for project work estimation and cost/benefit ratio.

Standard required information for performance testing includes:

- Anticipated # of total users
- Anticipated # of concurrent users
- Anticipated volume of transactions
- Application, system, & network architecture
- User & Transaction response time requirements
- Test Type Requirements (Load, Stress, Volume, Burn-in, Fail Over, etc)

**Risk Assessment** - At this point, the performance tester can present a list of assumptions and prerequisites for performance testing in a value-adding manner, as well as entry and exit criteria to accurately determine when testing begins and completes.

**System-Under-Test** – (SUT) diagrams should be created that documents the Test Environment including:

- Initial Hardware Recommendation (Hardware to deploy the developed application upon)
- Data Model Flow (Flow of information through system to determine possible bottleneck points)
- Test Execution Hardware (Hardware to drive the test from a separate location)

3.2.1.2 Test Planning

**Test Plan & Test Cases** - Performance Test planning follows the Requirements Gathering stage and feeds from the Functional/Non Functional requirements as well as the Initial Hardware Recommendation documentation. This stage revolves around development of the Test Plan. The Test Plan template provides an ideal structure and source of information regarding the various types of performance tests as well as common objectives, best practices and special considerations for each variety. The Performance tester must identify the functional test cases that will be used to emulate the throughput of the system.

**Proof of Concept** - A prototype application should be available at this time to evaluate and create a proof of concept that validates that performance testing CAN be accomplished on the system with the tools on hand. At this stage there should be a “go/no-go” decision made for the types of performance tests to use. This stage of testing does not require production grade hardware or the complete system as it is for POC.

**Test Strategy** - A test approach should be adopted that strives to simulate realistic throughput in the system as well as test all the objectives that have been set forth. If the system will not access data in a certain manner, there is little value in designing a test for such a scenario. A detailed
testing project plan with specific tasks is needed to organize the work flow process and adjust the master project plan. Performance Testing usually is not afforded limitless time, so prioritize according to the project plan’s needs!

**Identify Resources** – All resources should be identified including, “people-ware” (support & development representatives), software (application code, test software, licenses), and hardware (servers, workstations, network equipment) that are assigned to the Testing effort. Initial performance testing, including profiling and application subsystem testing, is most effectively achieved when executed in close proximity to the development and support organizations that will live with the application post deployment. A cost benefit analysis must be completed to show greater value vs. risk before outsourcing is considered acceptable.

3.2.1.3 **Design & Develop**

**Application-Under-Test** – (AUT) code must be developed with interfaces for each functional/performance test case that is required. Manual use cases must be executed on the AUT to ensure that functional bugs have been mitigated through fixes or workarounds. A Graphical User Interface (GUI) is the most realistic method to access the AUT and must be developed before use cases can be executed. Data Access Layer, EAI, and other subsystem components can be tested without the existence of a GUI providing, development resources create executables or custom web pages that can access their code via a standard interface (HTML, XML).

**System Access** – All necessary security access points must be established prior to script recording as this is a manual process to begin with. This includes NT/UNIX/Mainframe system access, as well as application user accounts numbered as high as the anticipated total # of system users. (100 users == 100 test user ID’s)

**Data Generation** – To make the test realistic, valid production data formats should be provided by the Data Administration team to properly verify subsystem functionality. The data doesn’t need to be actual production data, but it must follow all the same conventions & constraints as will the production system.

**Record Test Scripts** – Test scripts are initially created through a manual recording of an AUT functional test case on a client of the SUT. There are three types of virtual users that are valid in most testing:

1. **GUI Virtual Client** - Represents a single user on a single desktop workstation executing real-time client actions using object and window recording to capture accurate end-to-end response time.

2. **Protocol Virtual Client** - Simulates multiple users on a single workstation/server from the client environment communicating with the system-under-test via a standard application layer protocol (HTTP, FTP). This client can reliably scale the # of users without requiring many client workstations.

3. **Transaction Virtual Client** - Simulates multiple transactions on a single workstation/server from the client environment communicating with the system-under-test via standard application transactions (SQL). This client should only be used for benchmarking data subsystems with an artificial load.

**Create Test Scenario** – The scenario should reflect a realistic simulation of production user traffic travelling through the system increasing in load until the system shows symptoms of failure (incomplete or failed web pages, logon failures, SQL request failures, application errors, unacceptable response times, etc).

**Resource Monitoring** – Integrated performance monitors can quickly isolate system bottlenecks with minimal impact to the system. Monitors for the network, application servers, Web servers and database servers are best designed to accurately measure the performance of every single tier, server and component of the system during the load test. By correlating this performance data with end-user loads and response times, IT groups can easily determine the source of bottlenecks.
Optimize – The test scripts will have the most realistic and accurate effect if the data contained within is parameterized. This allows for inputting multiple values for variables in the test case. This method is mandatory for executing Contention Testing, to locate shared access errors and, verify database security.

Setup Test Environment – The Test Environment must be completed prior to executing any performance tests. This includes having the necessary hardware and software implemented in a simulation of what the production system will entail. Production grade hardware should be used to accurately size the application.

3.2.1.4 Test Execution

Execute Tests – Functional Regression tests must be executed on the system prior to any performance testing. The functional tests verify that the application is in a stable, working condition. Functional errors in the application will lead to inaccurate results during performance testing. Once a functionally tested component of the AUT is ready (see Entry/Exit Criteria), performance tests are executed in a prioritized order that includes capturing a performance baseline of a single user through the system, followed by Contention testing of multiple users in the form of Load and Stress Testing. For proper execution of performance tests, the scenario must include a minimum of two of the three types of virtual users and including a GUI Virtual Client to capture accurate end-user response times.

Compile and Analyze Data – The data gathered from the tiers of the application should be collated after each test execution to a central repository integrated into the Change Management Process. Some data may need to be gathered real-time and coordination of the support groups involved should be organized by QA. The optimal way to analyze data gathered from a test run is to monitor as many tiers of the system architecture as possible and correlate them with end-user response times and system transaction times. This three-way analysis will allow for cross-referencing of user response time issues with recognized system bottlenecks and transaction times reflecting where the bottleneck may reside.

Tune and Retest – Planned adjustments to the hardware or software architecture, including application and system configuration can be tested after capturing Performance Baselines.

3.2.1.5 Analyze & Recommend

If/When bottlenecks are found, the data from the results should be thoroughly reviewed and analyzed by various support and development groups to recommend useful suggestions to the project team in regards to application reliability, scalability, and performance. This data can be used by Management to negotiate Service Level Agreements between the Customer and IS Support that results in increased productivity and planned stability with little to no downtime for the Application.

3.3 Critical Success Factors

- There must be general project buy-in on the need of Performance Testing and the overall project mission.
- If the system doesn’t function properly or reliably, Performance Testing will neither go smoothly nor achieve accurate results; therefore, tests must be executed in a relatively bug-free environment.
- Technical subject matter experts should be on hand during test execution to assist in resolution of system, application, and configuration issues in a timely manner.
- Testing Hardware and Software infrastructure must be implemented and installed (OS, Servers, Clients).
- The network and system architecture are implemented and available (Routers, switches, WAN/LAN).
The application under test is implemented & stable within the test environment (Code, databases, Security).

- Performance test acceptance criteria must be quantifiable, clear, concise, and valid.
- Primary measures in Performance testing refer to the objectives originally agreed upon at inception.

3.4 Performance Profile Test Strategy

3.4.1 Do’s & Don’ts

3.4.1.1 Do’s

1. Do test only stable (free of critical, non-workable bugs) applications and environments.
2. Do use an independent Test System (TS) to simulate users (1 GUI user per machine) on the system under test.
3. Do use the use cases identified by functional QA to determine the test cases for performance tests.
4. Do monitor all servers involved in the test, including the Test System generating the load.
5. Do require performance tests to relate to a specific objective (i.e. end-user response during a trade).
6. Do capture performance test objectives from the Design Packets and Hardware Recommendations.
7. Do capture transaction times to measure functional business component performance.
8. Do execute a performance profile test on the entire regression test library during each release.

3.4.1.2 Don’ts

1. Don’t use more than one (1) client to access the system during the initial performance profile test.
2. Don’t execute the test from the same hardware that the Application resides.
3. Don’t assume performance requirements are stated explicitly in the Design Packs or anywhere else.
4. Don’t performance-test code that has not passed regression testing.
5. Don’t outsource performance profiling if any hardware (production level) is available.

3.4.2 10-step Performance Profile Test guide

A performance profile test will be built from the Regression Test that is executed by the Functional QA. Use a QA Master Test Plan Template to document the test proceedings and results. Below is a 10 - step process to follow when executing a performance profile test:

1. Receive unit-tested code from development manager with release notes indicating functional coverage.
2. Manually execute the use-cases covered by the release to discover functional bugs/defects in the code.
3. Open a defect change request in Clear Quest to initiate bug tracking and assign to development lead.
4. Record automated test scripts on code that has no functional defects, first.
5. Leave flexibility for workflow changes in script (programming logic to allow for loops and decision trees), room to add new data values (data-driven tests), and use a GUI error capturing function.

6. Add transaction timers in Robot scripts to measure user performance.

7. Set up one GUI client with resource monitors on all available application under test system servers and execute single user/use case to capture a performance profile of the transaction.

8. Archive results of the performance profile test for later trending data.

9. Only execute a performance profile test on use-cases that pass regression testing defect-free.

The performance profile test eventually WILL be run on ALL use-cases, as this is the performance benchmarking methodology to capture initial performance characteristics and to guarantee that no bug-ridden code reaches higher level performance tests, such as load, stress, volume, and fault tolerance.

3.5 Load Test Strategy

3.5.1 Do’s & Don’ts

3.5.1.1 Do’s

1. Do load test AFTER 80% of functional use cases for the release have passed regression testing.

2. Do use one (1) GUI client to capture end-user response time while a load is generated on the system.

3. Do use a Protocol Client to simulate many users on the system concurrently.

4. Do add users to the system incrementally with a standard buffer time (i.e. 5 seconds) between them.

5. Do keep the users on the system simultaneously to verify no system contention issues.

6. Do parameterize the data being used by the Clients to create a more realistic load simulation.

7. Do monitor all servers involved in the test, including the Test System generating the load.

8. Do monitor all network appliances involved in the test to measure network throughput.

9. Do have Subject Matter Experts on hand to assist in resolving application and hardware issues.

10. Do prioritize load test candidates according to critical functionality and high-volume transactions.

3.5.1.2 Don’ts

1. Don’t execute a load test on a functional piece until the performance profile test has been completed.

2. Don’t allow the Test System resources to reach 85% CPU/MEM utilization, as it will skew results.

3. Don’t execute load tests on everything!

4. Don’t execute load tests in live production or environments that have other network traffic.

5. Don’t try to break the system in a load test.
3.5.2 10-step Load Test Guide

A Load test scenario will be developed from the Performance Profile scripts and new recordings of protocol or transactional users.

1. Record new test scripts with a test automation tool by playing successful GUI regression test scripts or performance profile scripts while recording the protocol and server communication at the same time.

2. Parameterize all hard-coded data, URLs, IP addresses, user names, passwords, counter-parties, etc.

3. Correlate all session ids, database calls, certificate keys, and other dynamic user/system specific data.

4. For a functional release (R1), wait until 80% of the functionality has passed regression testing (week 5) and performance profiling before designing & executing the scenario, so that you have a valuable spread of functionality to generate production-diverse load levels.

5. Execute Load test in isolated environment while monitoring the resources of all involved devices.

6. Execute at least one (1) GUI user during a load test to capture end-user response time under load.

7. Initiate users to the system 1 to 5 at a time and keep all users on the system until test completion.

8. Increase users until 125% of the anticipated maximum to verify application utilization clearance.


10. Repeat as necessary.
3.6 Stress Test Strategy

3.6.1 Do’s & Don’ts

3.6.1.1 Do’s

1. Do stress test in an isolated environment, ALWAYS.
2. Do stress test critical components of the system to assess their independent thresholds.
3. Do use a Protocol Client to simulate multiple users executing the same task.
4. Do limit resources during a stress test to simulate stressful production levels.
5. Do use the test cases from Performance and Load Testing to build the test scenario.
6. Do try to find the system’s breaking points, realistically.
7. Do use Rendezvous points in tests to create stressful situations (100 users logging in simultaneously).
8. Do correlate/parameterize session ID’s and SQL calls to alleviate data & database contention issues.

3.6.1.2 Don’ts

1. Don’t stress test in production..
2. Don’t stress test code that is not successfully regression tested and complete (no workarounds).
3. Don’t use stress test results to make financial decisions without the input of Load & Volume testing.
4. Don’t stress test until Performance Profiling is complete
5. Don’t stress test just to “stress test”.
7. Don’t stress test unrealistic scenarios.
8. Don’t stress test in production, EVER.

3.6.2 5-step Stress Test Guide

Stress test scenario is used for determining individual breaking points for applications, hardware, and network implements. Stress testing can be executed on functionally complete and regression tested code. Performance profiling should be done to document initial characteristics, and then a great number of users/transactions will be driven through the system via a transactional client (LoadRunner) to stress it.

1. Receive Requirements for stress testing. (These must be provided by system/application architects)
2. Identify use cases from Functional QA that will represent stress test situations.
3. Identify the scripts from Load testing that will be used for stress testing.
4. Create a rendezvous point(s) to synchronize the virtual users (All users wait at a point in the use-case until every user is complete with the previous process and then simultaneously begin the next process).
5. Initiate users on the system rapidly, concurrently, and incrementally until the application starts to fail.
3.7 Volume Test Strategy

3.7.1 Do’s & Don’ts

3.7.1.1 Do’s

1. Do use GUI clients to capture end-user response time while a load is generated on the system.
2. Do use a Protocol Client to simulate many users doing multiple tasks concurrently.
3. Do run volume tests for an extended period of time (over an hour).
4. Do run volume tests with a large transactional mix, multiple users, and production size databases.
5. Do volume test by increasing load gradually and executing background transactions, simultaneously.
6. Do schedule volume tests for off-peak time execution (nighttime).
7. Do volume test when 100% successful functionality and performance profiling has been accomplished.

3.7.1.2 Don’ts

1. Don’t run volume tests in a production environment.
2. Don’t volume-test code that is not successfully regression tested and complete (no workarounds).
3. Don’t volume-test until all performance profiling is complete.
4. Don’t volume-test until all load & stress testing is complete.
5. Don’t volume-test without production grade system hardware available and implemented.

3.7.2 5–step Volume Test guide

1. Receive 100% complete functional area application release (R1, R2…) with bug fixes included.
2. Execute the entire Performance Profile library with at least one (1) GUI user.
3. Execute the entire Load Test library with the fully anticipated volume of users/transactions.
4. Set up resource monitors on all AUT system servers and execute all GUI and Protocol scripts.
5. Present Recommendations for volume trends and resource utilization.
3.8 Fault Tolerance Test Strategy

3.8.1 Do's & Don’ts

3.8.1.1 Do’s

1. Do fault tolerance testing for systems that require 99% uptime and higher.
2. Do use GUI clients to capture end-user response time and functional errors from the system.
3. Do use a Protocol Client to simulate many users doing multiple tasks concurrently.
4. Do simulate application, power, and network catastrophes to induce service level challenges.
5. Do unplug application, web, database, and SAN network connectivity and verify no user issues.
6. Do kill application process on application, web, and database servers and verify no user issues.
7. Do execute recovery procedures while the test continues and verify no user issues.
8. Do execute recovery procedures prior to releasing to production to verify that they are timely.

3.8.1.2 Don’ts

1. Don’t fault-tolerance-test in production, EVER.
2. Don’t stop the test to recover the system (end-users shouldn’t be affected, record entire experience).
3. Don’t fault-tolerance-test code that is not successfully regression tested and complete.
4. Don’t fault-tolerance-test until performance profiling & load testing is complete.
5. Don’t fault-tolerance-test until production hardware & network environment is available & complete.