Performance Testing Primer for AIQ

Your quick guide to achieving the best test results without reading entire books on Performance Testing

What is Performance Testing?

We ultimately want to understand the performance of the application or system under various conditions. This requires a number of specialized tests to understand the behavior and guarantee stakeholders that the system will perform well under reasonable circumstances. Performing well includes transaction times as well as not crashing.

Why is it important?

People are used to 300ms performance from Google and sub-1-second performance from Amazon. The world has been conditioned to very fast response times. People begin to get distracted after 1 second, and at 7 seconds 50% have moved on at least mentally to another task. Performance testing at each and every build catch issues that can crop up at any time.

Types of Tests

- **Performance** | What is the transactional performance of my application to the user (through the UX) under a variety of conditions from 1 user to N users?
- **Load** | How do my services perform under load at the API level?
- **Stress** | What happens when my application is under stress for a short while or longer?
- **Scalability** | How does my application scale?

- **Soak** | How does my application perform over time specifically concerning memory leaks or other behavior which changes over time?
- **Spike** | Does my application recover after a large spike in traffic?
- **Database (direct)** | How do direct database interactions impact throughput and performance?

Testing in Today's DevOps World

Performance testing should be a standard part of every build where you build, test, tune, and start again. Performance improvements occur, and catching performance issues before your users do because it happens at each and every build. Correcting issues in build 144 that occurred in build 72 is not fun, nor productive.

Create Repeatable Real-User Simulation

Tests must be **realistic**. That is, closely resembles the way real users will interact with the system as well as repeatable so that you can track progress.

1. Identify the maximum number of users the system will ever see (call that Max Design Capacity)
2. Identify reasonable SLAs (max transaction times) that your users should tolerate
   A modern target would be under 1 second response times through the UX
   A good start would be under 3 seconds for all transactions and work towards 1 second
3. Create a mix of relevant use-cases at the UX level as well as optionally at the API level and database level
4. A good scenario will have many use cases represented which interact with the system in a realistic way and activate all aspects of the system such as the database and other layers
5. Data-drive those transactions for more realistic scenarios
6. Randomize think times and interval times to create more realistic scenarios
   Generally, randomize think times between 1 and 10 seconds
   Generally, randomize interval times between 5 and 20 seconds

Setup and Run Tests

A. Monitor

Place APCMonitor on your web, application, database, and all other servers associated with the app under test.

B. Measure Transaction Times Without Load

Run a single user performance test with one scenario repeated 10x in series. This both ensures that everything is functioning correctly including your application and that the nodes can see the application. This will also provide a baseline for transaction times

1. You should not see any errors with this 1-user test. If you see a consistent error you may have a script or application issue.
2. You may see some steps which take a long time (over ten seconds) in the report. You can validate these manually and see what actually happens. Some steps can have a lot of elements loading besides the visible ones and the browser will still show in an active state, and Appvance will await all elements to come in, even though a real user may have moved on. So you may decide to ignore this step results for the moment as it won't impact a user, but more that there could be better ways to code that Ajax activity in the future. Also, this can be different for each browser.
3. If you do not achieve your SLAs for transaction times at this step, STOP and address the issues before proceeding.

C. Calibrate the Test Nodes

It is critical to calibrate a test node for max capacity for your exact Scenario. Each scenario requires new calibration.

Follow the calibration guidelines which include ramping up users on one test node until that node reaches CPU and memory limits, or other aberrations occur. When other issues occur these could be due to your application being at its limit (and **not** the test node) or could occur because GPU or other non-monitored data are at a limit on the test node. Back down by 20% from any limit and call that the calibrated maximum per node. Enter that number into each test node setting in the scenario going forward and assign enough nodes to get to the totals you need.

D. Run a Scalability Test

Create a scalability test by picking at least 3 levels of load (it can be more) and setting that up to automatically occur within one Scenario.

This can provide performance, load, scalability, and stress all in one thoughtfully designed test. The top-level should be at or above the design maximum for the application. For example, My application is designed for 900 concurrent users. I would set up a scalability test for 100, 300, 600, 900, and 1200 users to run that sequence automatically.

1. Run each level of load for enough time that each use case could be re-run 5-10x. So an average 2-minute use case run 5x would need at least 10 minutes at each load. This ensures more data for average
2. Ignore error rates below 10%. Some transactions may fail for a variety of reasons but since this is not a functional test, we are only interested in the transaction times and scalability
3. Transaction failures may approach 100% when you have gone beyond the design maximum of the application
4. The key chart to review at the end of the test is the Scalability Index. You will want linear scalability. Anything not linear shows your system is not keeping up with requests
5. You will want to investigate the full CSV data file and look for transaction times that exceed the SLA
6. No app-related servers should get above 90% resource usage on any resources such as CPU, memory, HEAP, etc. as reported by APCMonitor. Should any servers see levels this high, additional hardware is required for further scalability.

E. Run a Soak Test

It is important to run a soak test for at least 24 hours to assess any abnormalities which occur over time. Use the same scenarios as above and pick a mid-level of load which had few errors and was within the linearly scalable range of the application. Setup the Scenario to ramp up over an hour and stay at the chosen level of load for 24 hours and then ramp down.

F. Run a Spike Test

A spike test can assess how your systems recover after a fast spike in users. You can set this up using the same scenario as above. Change it to run for 10 minutes at 20% of max design capacity, and then spike for 5 minutes to 150% of design capacity and then drop back to 20% for another 20 minutes. The system should recover properly and meet SLAs after the spike.

Troubleshooting

Possible Database Issues

- Insufficient indexing: Tune database indexing to improve query processing
- Fragmented databases: Place table records on adjacent pages
- Out-of-date statistics: Degrade query optimizer performance
- Faulty application design: Excessive DB calls, excessive data requests
- Many other important database issues can be found here:

  https://dzone.com/articles/reasons-slow-database

Typical Web and App Server Issues

- Poor server design: Inefficient data or page caching
- Memory problems: Physical memory constraints
- High CPU usage: Usage >80% indicates problems
- Poor database tuning: The application server sending too many DB requests
- Poor cache management: Produces high CPU usage, disk access
- Poor session management: Produces high CPU usage, disk access, Time-outs
- Poor security design: Excessive use of HTTPS protocol
- Many other important issues can be found here:

  https://www.serverwatch.com/servers/server-optimization/

Network Issues

- Firewall capability
- Internet access throughput, packet loss, delays, and bandwidth
- Load balancers, gateways, routers
- Many other important issues can be found here:
- http://www.dataexpedition.com/support/notes/tn0009.html

Resolving the Issues

Depending on time, expertise, and accessibility, one might choose different items to address first. Given the very low cost of CPU cores and memory today, it's hard to justify spending manpower on addressing issues before spending $$ on faster hardware.

- Upgrade hardware: RAM, CPU, network bandwidth (RAM and CPU cores are cheap!)

- Improve current application design: Algorithms, caching, DB calls, memory usage
- Upgrade software infrastructure: OS, web server, database
- Upgrade system architecture: Client-server to basic n-tier, basic n-tier to enterprise n-tier, software and hardware changes

---

Last updated: July 03, 2023

If you have any feedback on this topic, please contact us at documentation@appvance.ai

---

Revision #8
Created 2026-07-31 15:09:13 UTC by Admin
Updated 2026-11-17 21:58:26 UTC by AI_Mananger