Benchmarking NGINX: 4 Ways to Improve Accuracy

People frequently ask about ways to accurately benchmark NGINX. The goal, of course, is to determine the maximum level of performance possible—and this is important data to understand. But if you've ever attempted benchmarking a web server before, you know there are times when results can be confusing or misleading.

Unfortunately, little information exists online to clear things up. Most of the published data is overly simplified and lacks critical details regarding test environments or configurations—both of which play a large role in the process as well as the outcome. People end up drawing conclusions based on limited, synthetic tests, and inaccurate—or even spurious—test results. Naturally, this produces conflicting opinions with no real answers.

This paper provides four concrete tips to help you avoid benchmarking confusion, so you can achieve the most accurate and understandable results.

1. Set the stage properly with the right tools & test environment

When you measure web server performance using specialized tools, remember that you will actually be measuring the performance of both the test tools *and* the web server. You'll also be dependent on the environment in which you host the test. Here are a few things to keep in mind:

- **You're working with many variables.** A benchmark environment is a complicated mixture of components, including HTTP-request generators, OS kernels, different hardware pieces, network topologies, and so on. Every component in this mix affects your benchmarking. Even subtle things—like the kernel version or a particular network card or driver—might greatly influence your test results. Always use the latest version of the server software. And remember, of course, that you cannot apply one environment's benchmark results towards a different environment with a different set of components.

- **Different tools do different things.** To test web server behavior, you need one of two things: real clients talking HTTP to your server, or simulated HTTP requests from any of the free or commercial tools available on the market. While tools like ab might do just fine for a very simple test, alternatives like wrk and siege might be better for more serious situations. Different testing tools also create different load patterns that may be a far cry from your real-world load. Know what you're working with.

- **The ideal is not always real.** Ideally, the performance of the request generator should be much higher than the maximum performance of the server being tested. But a typical HTTP client code is, in fact, slower than a well-written HTTP server code—which naturally creates more overhead in terms of calls to the operating system. And this, in turn, decreases performance.

- ✓ **You're testing the whole thing.** For the majority of applications, the role of a modern web server is to orchestrate the OS kernel functions, so that the hardware and OS resources can most efficiently serve users' requests. In this regard, real-world benchmarks are about testing the *entire assembly*, not just the web server code algorithms.

- ✓ **Don't create a competition.** Many people choose to do benchmarks on the same machine, but this requires the client and the server to severely compete for system resources, which in turn can generate vastly different results. Consider investing in additional resources, and build a test environment that best matches your benchmarking goal. You might even consider using several clients in a back-to-back testing, so you can fully saturate the network and the server's resources.

2. Test with a config that mimics your real-world use scenario

Configuration matters. That's why we ship NGINX with a default setup that we believe is the best for performing in most real-world situations. But benchmarking is a synthetic process, and this can lead to some difficulties. Remember these tips:

- ✓ **Stick with the NGINX default.** We've pre-configured NGINX for optimal performance in most real-world environments—so don't change it if you don't need to. (Of course, disable any features that won't be used in production.) Watch out for the default setups of some benchmarking tools, which will cause you to test a completely unrealistic traffic pattern/load.

- ✓ **Requesting clients behave differently.** They not only create and process requests differently, but they also vary in network bandwidth, latency, and packet loss. That means that each of the potentially hundreds of thousands of simultaneous connections processed by NGINX (standard quantities for real-world use) could be handled differently depending on the client. For example, the Google Chrome browser prefers to open quite a few "spare" connections to the server, and the ones not used are left pending. Some connections carry "heavyweight" requests, while others are being utilized for a much "lighter" load. And, of course, some connections just remain idle, eating up server memory.

- ✓ **NGINX Configuration Language (NCL) directives can affect your results.** The usual culprits are in the list below, but we advise you to refer to the documentation for more details:
    - accept_mutex
    - worker_processes

  - worker_connections
  - keepalive_timeout;
  - lingering_close;
  - sendfile
  - keepalive
  - aio
  - open_file_cache

- ✓ **Watch out for SSL.** Increasing the size of the certificate key—from 1024 to 2048, for example—can actually drop performance by five times per second for new, un-cached SSL handshakes. When benchmarking and tuning SSL for maximum performance, your best option is often to employ RSA/AES based cipher suites, and use 1024-bit certificates.

3. Use statistics

Statistics are the essence of benchmarking, so take some time to understand the basics. Use tools like *ministat* to check your results. Here are some additional points to remember:

- ✓ **Always evaluate the number of samples (probes), the standard deviation, and the errors.** Peaks and averages are not really meaningful without fully assessing the distribution. For any given test, check if you have large variation in samples. If that's the case, something is definitely wrong somewhere.

- ✓ **Don't focus on the number of requests per second (RPS).** This rarely reflects real-world use, as browsers typically don't request all of the page resources at once, and actual web page rendering is quite a complicated process. Make sure you're trying to emulate faster, lower latency communications with your web server as seen from a client. The goal with the benchmark is to understand how your web server behaves when faced with a growing and non-uniform load from real users.

4. Take external factors into consideration

It might seem obvious—but remember that you are not conducting your benchmark in a vacuum. Be sure to check the following:

- ✓ **Ambient temperature.** Modern processors can dynamically adjust cycle frequency depending on the load or the current CPU temperature. If you haven't turned off this kind of behavior optimization for your benchmarks, your results might get quite unpredictable.
- ✓ **Scheduled processes.** Review and disable any scheduled processes (think: *crontab*) that might affect your benchmarks in an unexpected way.
- ✓ **System and error logs.** Look out for anomalies, such as:
    - *[..] CPU1: Core temperature above threshold, cpu clock throttled*
    - *[..] possible SYN flooding on port 80. Sending cookies.*
- ✓ **CPU usage.** Check the CPU usage of your clients, and that of your server. Use traffic monitoring tools like *wireshark* to help.

Still stuck with a performance issue? Try to dig out the true source of the problem. Often, it's not related to your web server. Many factors are at play, including client performance, the network bandwidth between the client and server, the performance of your network card (think: packets, excessive interrupts), disks, free RAM, etc. Make sure you monitor *everything* in order to plan carefully for real-world capacity needs.

Summary

As you conduct your benchmarking of NGINX, we hope you will keep these tips in mind. Maintaining and improving NGINX performance from release to release is our priority. That’s why we regularly test new builds for any regression in performance (or other errors). We also frequently check different algorithms and compiler options, and we track details of how the current OS kernels, hardware, and CPUs function. Statistically meaningful results find their way into actual code optimizations.