Smarter Simulations in Performance Testing

By Matt Heusser for Subject7

In my introduction to performance testing I mentioned two problems that stand in tension of each other. On one hand, you might make testing appear too simple. At the extreme end, this is just hitting a static web page frequently, and never actually hitting the parts of the website that require database lookups. The performance test might come back with the all-clear, and yet fail when real users start to access information. On the other side of failure, you have the perfect, overly complex performance test that is uneconomical to create. These sorts of performance tests are expensive to make, but often even harder to maintain. They get used once and thrown away. Six months later, the software has a performance problem due to a change, and the first place it is noticed is in production.

The challenge here is to balance these two; to create simulations that provide the right amount of information for the right price. Today I'll share a few tips on how to get there, starting with how we create simulations.

Simulations and Models

While they may have gone out of fashion, most of us are familiar with model railroads. These increase in size from what is essentially a toy car, to something that runs on wooden blocks, to electric, and even the scale model trains that people can ride on. As you go up in size, the details become more authentic. They start to be self-propelled, with electric slots, then electric motors, then gas and coal. Light-up headlights appear at some size. When you add a human controller, those headlights can be switched on and off by a button on the train. Passenger cars start to be actual hollow shells with space, not just plastic windows with a painted-on background to look like the interior. At full-size, we stop talking about "scale models" and start to talk about "reproductions."

Imagine buying a Thomas the Tank Engine collectible, then trying to use it for real engineering work. It would be laughable.

Yet that is how many performance testing projects start; and I'm going to say something counter-intuitive here: that might be okay. *Just don't stop there.* Today I'll cover what the spectrum looks like, and how to grow that test effort in complexity until you stop at the sweet spot.

Getting Started

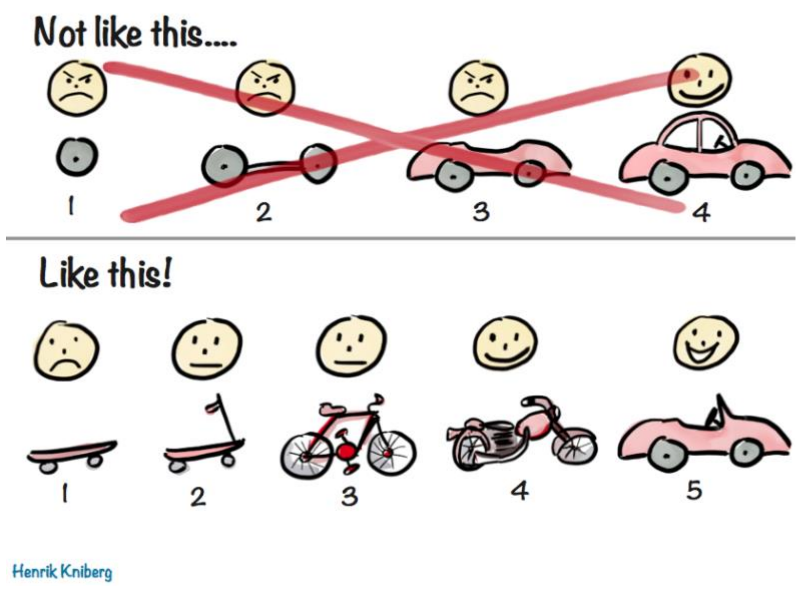
In my world of tradeoffs, over-engineered is often worse than under-engineered, because at least with the lazy approach, you get feedback quickly. With the hard approach, feedback takes time.

As I said in the introduction, there are often problems with even the single use case. Even if there is nothing alarming, there may be problems that are awkward enough to deal with using a laptop and ten simultaneous users. The goal here is to get meaningful feedback to the engineering team quickly. This will keep development making forward progress and help to ensure that the performance test isn't the bottleneck; *fixing* performance is the bottleneck. The interval between may even enable you to block moving the code to production, buying you time to create a real performance test to figure out what is actually going on.

Bear in mind that protocol-level tools are limited to just the packets that go through the internet. Page-rendering problems, where the user interface is complex and slow to appear, cannot be found by tools that only measure server response time. Most of those tools do not look at the web pages for return results (they are performance, not "functional" tools). Instead, they only look at time for the request/response, along with the response codes, such as 200 OK versus 404 NOT FOUND. That means it could be sending back the wrong page, or wrong details, and none of that will show up in the results. The simplest of these tests will use the same user over and over to make a single lookup. Complex systems will keep the answer in memory, or at least readily available, or "cached." That means the "simultaneous user" test will really be the same user many times, looking up the same product. One term for the accuracy of the simulation is the *fidelity.* This approach is low-fidelity, quick and easy, more like a "Thomas the Tank Engine mini" test than an accurate scale model.

Still, it keeps us moving. To scale our tests up, we want to understand how systems tend to fall over as they get more complex and add in users for a multi-user experience. Then we can design the scaling up of the tests to scale up with the system. This is an iterative approach that provides feedback at every step.

In other words, we want a minimum viable product performance test that scales up. Something more like this diagram illustrates. There out to be a progression with your performance testing, not a progression of building it from the ground up, but rather in making it better, more robust, and more sophisticated.



Source: Henrik Kniberg, Making Sense of Minimum Viable Product

How Systems Fall Down

Systems tend to have breakpoints where something stops working. Often, I find this at one user, then again at two or three. Those aren't really multi-user issues as much as poor design. The single-user failure case is a regular old bug. Multi-user error could be threading or a singleton class in the wrong place so users get the results of other users. Simple performance test tools or single user testing will not know this, as most performance test tools focus on monitoring the speed of results or system utilization, not testing is-the-data-correct-on-the-screen. Those are the kind of defects you might find by having a team pizza session at lunch. If you can't talk the team into multi-user testing, at least try with two laptops. Given tablets and cell phones, it can be surprisingly easy to click two buttons at once. Some of this overlaps with functional testing. You might, for example, add a comment to a post on one machine half a second after deleting the post on another. With real humans doing the testing, looking for incorrect data on the screen is easy.

Beyond that, it is time to scale up, using a trick I like to call the *reverse gossamer condor*. The Gossamer Condor and the next version, the Gossamer Albatross, were the first heavier than air, human (pedal powered) aircraft capable of sustained flight and direction. They were essentially bicycles combined with an incredible ultralight aircraft, where the pedals converted energy to turn the rotors. The main problem with the planes was weight. They simply weighed too much. To figure out how to reduce the weight, the creators would just throw the plane off a cliff. Anything that didn't break would be weakened (made lighter), what did break would be reinforced (made heavier). When the Smithsonian wanted to put a reproduction in a museum, they asked the creators for blueprints which did not exist. The entire system was too fluid.

The first performance test can be very similar to that, more learning an exploration than planned-up-front. Here's what that looks like in practice.

Reverse Condor Performance Test Strategy

Monitor the components - CPU, memory, network, disk, databases, and APIs - to see what is performing slowly as the test becomes more realistic. When performance hits a "hockey stick" (becomes terrible and times out) at a certain number of users, one of those components is overloaded. Find the bottleneck, figure out what is taxing it, then either add resources or fix the code to use less of it.

Meanwhile, use the fix time of find/fix/retest to refine the tests. It's okay to start with log-in as one user, find one product, add it to cart, log out. Then talk to the programmers about how the system might fail, increasing the fidelity to address the risks. You might make the tests run through browsers, to simulate the real user experience, as rendering could take two seconds or more. You could search for different products, and put a random number in the cart. You could add randomization between activities, or create a meaningful delay between steps, like real users. If you have functional tests to re-use, you can slowly begin adding a variety of activities. If you've got a lot of well-formed functional tests that can be re-used, you can run them in a semi-random order.

Eventually you'll be able to explain the test strategy and heads will nod that it is "good enough." At that point, it's time to shift to figuring out how to run the tests on an ongoing basis, while keeping them in sync and without creating an undo maintenance burden.

But that's a story for another day.

For now, get meaningful feedback fast and improve the performance testing until it is good enough. That might not be the right prescription for every project, but it's certainly a good place to start.

Ready to learn more about Subject7?
Contact us today to request a free demo.