SUBJECT7
QUALITY IS A CULTURE

Performance Testing in the CI/CD Pipeline

By Matthew Heusser for Subject7

You can skip the cup of coffee; this might be my shortest ever blog. Some have asked what you need to run performance testing in the CI/CD pipeline, and here's my answer. To run performance testing in the CI/CD pipeline, all you need is to "just run the tests!" Well, except for a few minor nuances of course. Like the fact that…

… the tests don't produce output that the continuous integration system can read.

… the tool can't run through the command line, so CI can't "kick it off."

… for the results to be valid, the tool requires a test environment that mirrors production, which is expensive, and you don't have one.

… the data needs to be set up in a way that the performance test can be recognized. The users need to be in the right state. The database might need to be rebuilt on every run.

… all the performance testing needs to complete within a reasonably short CI window, probably measured in minutes.

… as the code changes, the performance tests (specifically, load tests) start to fail. If they run through a browser, they will start to fail when the user interface changes. If tests send pre-recorded packets, these will change as the APIs and web forms change. This creates maintenance similar to GUI test automation -- except the performance tests, once created, tend to look and feel *invisible*. Over time, the performance tests "fork" from the codebase making their results something between flaky and untrustworthy.

These are all reasons to not put performance testing under continuous integration, or perhaps, reasons to give up and turn the load tests off. More than reasons, they are system forces. Without thinking them through, this sort of thing will just … happen. The executive will create the proposal, get the budget, hire the performance test contractor, give high fives and be happy the project worked. Then, six months or a year later, in a discussion in the hallway, the executive brings up the topic with a technical person, who says "oh those? We turned those off six months ago. You'd have to talk to Sydney about getting that restarted."

Today I want to talk about a better way to incorporate continuous integration, and why it's worth the consideration and planning.

Performance in the Pipeline

Getting performance tests in the pipeline isn't that different than functional tests. You need to create a working test system and fill it up with data. For load tests, there might need to be more data and more users. And the test system will need to be more high-powered, nearly as much as production. You can cheat with a system without failover redundancy, or with half the

processing power, and assume performance will be at least that good or better. Yet if you skip components like a load balancer or cache, there is a chance for performance problems caused by the components that you've skipped.

Once a working test system exists, things change. Instead of testing at the end, before release, load testing is now c*ontinuously monitoring how changes in the code impact system performance*. Pause for a moment and think about the kind of insight that provides. Feedback comes in an hour to (at worst) a day, and is tied directly to a specific change. That means the programmer who made the change can be notified and do the fixing if there is an issue. There is no need to argue about what might be broken, to debug to find the problem, to create a small team of a tester, developer, and product owner to make the fixes, test them, and decide if the results are now "good enough." The programmer can do this work as part of the regular workflow. It will be small enough that there is not a need to create a "story" to prioritize; at the most it is a ticket or bug to track.

What this process really opens up is the potential for learning from instant feedback. When the programmers make a change that hurts performance, they know exactly what the change was, and can discuss it during standup or retrospective. When a change improves system performance, that change will be picked up on the monitor. The team can explore and share that too.

System performance testing tends to devolve over time, just like quality. Without someone minding the store, new features make the software slower and slower until it becomes a crisis and morphs to a retest/rescue project. With this approach, performance doesn't need to degrade. Done well, it could even improve performance and quality.

We've been doing this for years through the user interface and API testing -- generalizing a defect into a category, then explaining how that category was created and preventing it from appearing in the future.

Think about what would happen if you did that for performance testing under load.

Divergence and Convergence

Most companies seem to be able to keep their functional tests up and running, as part of the CI pipeline. In the best case, the requirements at the feature level are that the feature is not run until the functional tests run. The programmer does not check-in, or at least does not merge the code into the codebase until the performance tests that touch the specific code have passed. Then CI picks up all the changes and does a full run. If anything breaks, it is checked and fixed the same day. That's better than running regression at the end of a sprint and far better than at the end of a project.

Of course, Subject7 makes a user interface and functional test tool that can turn those functional tests into performance tests – literally on demand. When you change the functional tests because of a feature change, the tool can regenerate the performance test suite and run it

in a browser that will deal with any changes to the user interface that cause the packets to change. The tool can also run on the command line and generate output that is easily adjusted to match your CI tool. This doesn't eliminate the pain of creating the properly scaled test environment or refreshing the database, but it does change the system forces so that performance testing keeps up. That means finding out when a change materially impacts performance within a day, and even less time to make the code change. You can even compare run-times to see how much performance degraded *and* under what conditions it took place. You can take immediate corrective action scoped to that change. With this approach, performance testing makes pinpointing, debugging, experimenting, and fixing issues a straightforward part of the development cycle, not an "optional story" that needs to be "funded" by the product owner. This changes the system forces around load testing itself, making it more likely to stay in the pipeline.

Conclusions

There are significant challenges in getting performance integration to run under CI. The timelines are massively compressed, the tests must set up, execute, and report without any human interaction, and over time the tests become stale. Given the state of testing today, it is understandable why so many teams limit performance testing to a project that runs right before big changes, big shopping seasons, and so on. Getting the tests to run in a tighter window (at least overnight) and providing mechanisms to keep them up to date automatically, makes it at least possible to get performance tests into continuous integration.

Once the work is done to put the performance tests under CI, the value becomes evident almost immediately. Try it and keep us posted on how it turns out…