endava

# AGILE PERFORMANCE TESTING

**ALAN GORDON –** HEAD OF PERFORMANCE TESTING
**ALEXANDRU GATU –** SENIOR PERFORMANCE TESTING ENGINEER
**MARK FIRTH –** HEAD OF TESTING SERVICES

# AGILE PERFORMANCE TESTING

**THIS PAPER WILL DISCUSS:**

- How to overcome the key challenges of implementing performance testing in sprints
- The best ways to integrate test tools with the CI process

An increasing number of companies recognise the benefits that agile development and DevOps bring in helping to improve software quality and reduce time to market.

While the importance of testing and test automation as an integral part of agile development is widely recognised, performance testing is often not included in the sprints and is either overlooked entirely or conducted only on the release candidate shortly before deployment to production.

This article outlines some of the tools and techniques you can use to implement performance testing effectively in agile projects, and gives examples of some technical solutions for both Java and .NET platforms.

In waterfall projects, performance testing is usually not conducted until late in the testing phase, which leads to project risk – project management can only hope that either the system will perform as required or that any issues will be addressed through some quick tuning or configuration fixes.  Many project managers face the awkward choice of going live with a poorly performing application, or incurring delays and additional costs to fix issues that were found too late.

With an agile methodology, these problems should be resolved. Performance testing is conducted during sprints, and non-functional acceptance criteria are defined for each story as part of the Definition of Done, so performance is proven to meet requirements before any functionality is agreed to be complete.

# TESTING CHALLENGES

The benefits of agile performance testing seem obvious but there are many challenges you will encounter in implementing in-sprint performance testing:

- How to include performance testing in the continuous integration (CI) process
- How to performance test an unfinished and frequently changing code base
- How to get meaningful performance results in an environment much smaller than production
- Identifying the best approach for performance test tools and/or harnesses

# TEST APPROACH

***Each scrum team should decide what's best for them.***

The proportion of the performance testing that you can conduct in-sprint depends on the similarity of the test environments and data to production and the skills and tools available.

In a perfect world development environments would be an exact replica of production, which would enable all of the performance testing to be conducted in-sprint.

In reality this is very rare, which means that some aspects of performance testing have to be conducted less frequently in more production-like environments.

Other than environments, in-sprint performance testing relies on selecting appropriate tools and using them to implement continuous automated performance regression testing in the continuous integration pipeline.

To achieve this you need experienced performance test engineers who are familiar with the technologies in the development stack and the tools and techniques to integrate them. Not all scrum teams have to have a dedicated performance tester – each team should decide what's best for them.

This could mean engaging resources for specific points in the project such as an expert to build the performance test framework and mentor your team during the early sprints.

*Make results visible immediately and to the whole team.*

# IN-SPRINT TESTING GOOD PRACTICE

Test execution should be fully automated so tests can be scheduled without, for example, manual test data setup slowing things down.

You should make results visible immediately and to the whole team.

This means that your performance test tool must be integrated with the CI tool and results should be available in the dashboard.

There may be a cost to implement this, but the immediate feedback will make it worthwhile.

Run tests as often as possible without holding back the build. Testing nightly is often an appropriate frequency.

Use version control and actively manage the test pack, removing redundant or less important tests to stop it growing too large, as well as adding new ones.

Execute in-sprint and post-sprint testing using the same test scripts and approach – the main difference is that the environment is more like production in specification, configuration, and perhaps in terms of test data composition and volume.

Testing in a small environment will uncover some performance issues but will not predict exactly how the application will perform in production.

Continually comparing results against a baseline identifies any performance degradation.

Scalability testing can also be done in a small environment – testing different configurations (server specification, number of servers) to build a model of application behaviour.

# NON-FUNCTIONAL REQUIREMENTS

Meeting non-functional requirements (NFRs) such as response times and volumes should be included as part of the Definition of Done, providing performance acceptance criteria for new and existing functionality.

- Default NFRs should be defined early in the project, for example the typical response time for any actions that write a new record to the database.  Write specific non-functional acceptance criteria for user stories that override the default requirements.
- NFRs should be clear, testable and feasible – including a tolerance threshold, e.g. 95% of simple page navigations should complete within 3 seconds – express these as constraints.
- Volumes must be scaled down for smaller test environments – there is no formula for this, but you can calibrate the volumes, adjusting these over a series of trial tests.

***Non-functional acceptance criteria should be part of the Definition of Done for each story.***

For all quality attributes, including performance, there is a trade-off between the cost and the level of quality. For example, for a web application to have 99.999% availability would have a high cost and it may not be truly necessary.

# INTEGRATING IN THE CI PIPELINE

The build management system creates the build from the code repository runs the unit tests and deploys to the test environment(s) if all the unit tests pass. Next, the functional tests are executed and then the performance tests are executed only if the functional tests achieve target pass rates.

Test tool selection is heavily influenced by the options for integration into the chosen CI tools stack. You can use a combination of plugins, APIs, test tools libraries and command line arguments to achieve this integration. The aim is to allow results and error reports to be available on the CI dashboard immediately after test execution.

Many CI tools allow direct integration of the performance test results with the dashboard:  however, there are some aspects such as deciding whether a test has failed due to performance degradation and presenting percentile values that require additional processing.

Transactions may also need to be recalibrated as they differ from build to build, and fault tolerance has to be built in before reporting failures. This is achieved through a combination of the functionality of the CI tool (and plugins) and custom scripts to present an accurate reflection of performance against SLAs.

**EXAMPLES:**

- Integrating JMeter with Maven and Jenkins, configured to show performance trends across builds with defined error thresholds
- A .NET project where we integrated LoadRunner with Microsoft Team Foundation Server, starting the tests and analysing the results automatically

*As new features are added, so the individual feature tests grow to become a full end-to-end performance test.*

# TYPES OF PERFORMANCE TEST

Different performance tests can be executed at different stages:

**Performance unit tests** should be included to measure the performance of system components and track the impact of new code or refactoring. They can be run with other unit tests, after each build.

A simple performance measure is to track the execution time of each unit test across builds. At a more advanced level, we can design multi-threaded tests to observe any change in performance, or to test with different data sets.

The test machine used to run unit tests should be identical between test cycles as hardware differences will produce inconsistent results.

**Component tests** can be written to focus on individual components before the whole system is integrated. For example, a script that runs SQL queries directly against a database; or a test harness making API calls. Running component tests in this way gives the team a chance to tune and configure each system layer in isolation, so when we have an end-to-end system we can be confident that individual layers are performing well.

Performance tests can be designed to test individual stories, which are often end-to-end functionality but with a narrow focus. Such feature tests are very useful to provide early feedback to the team on system performance.

A test including only user login is often the first feature test. Other examples could be calling individual web services to request quotes, or creating new customer orders.

As new features are added, so the individual feature tests grow to become a full end-to-end performance test.

# BENEFITS

**YOU WILL BENEFIT BECAUSE:**

Finding issues in the sprint has huge advantages compared to finding them in a separate performance test weeks later, when issues will have to be fixed immediately (disrupting the next sprint) or added to the backlog for future sprints.

It will raise general awareness of performance and lead to better architected, more scalable applications. Your project will be much less likely to experience delays due to poor performance in release testing and hence more likely to be able to go live on time.

We have seen considerable benefits delivered on many projects following the implementation of Agile performance testing techniques.

On one project we found and resolved 25 performance issues during sprints, which would not otherwise have been found until later phases.

We were able to test 78 different performance test scenarios during development – a far greater number than is typically tested in a short, intensive burst of traditional performance testing.

The performance of the application improved by 35% compared to the previous version.