BROADCOM SOFTWARE

WHITE PAPER

OPTIMIZE CONTINUOUS DELIVERY OF MICRO-SERVICES APPLICATIONS WITH CONTINUOUS PERFORMANCE TESTING

Shamim Ahmed
CTO for DevOps Solutions, Broadcom Software

WHITE PAPER

BROADCOM SOFTWARE

OPTIMIZE CONTINUOUS DELIVERY OF MICRO-SERVICES APPLICATIONS WITH CONTINUOUS PERFORMANCE TESTING

Shamim Ahmed, CTO for DevOps Solutions, Broadcom Software

TABLE OF CONTENTS

Introduction

What is "Continuous" Performance Testing?

Key Practices for Continuous Performance Testing

Pulling It All together: Continuous Performance Testing Lifecycle

Next Up: Continuous Reliability

INTRODUCTION

I often hear from customers who complain about how "classic" performance testing (i.e., end-to-end testing with high volume of virtual users) of their applications before release slows down the cycle time by several weeks. In addition, the testing significantly consumes both people and infrastructure (hardware and software license) resources.

This is especially true for retailers (and other e-commerce providers) who do this type of testing typically before a major product/service launch, as well as before key shopping seasons. They ask how they can reduce (or even eliminate) the testing bottleneck, and instead be "peak performance ready" all the time, so that they can release software updates without incurring a delay, but simultaneously not risk issues in production.

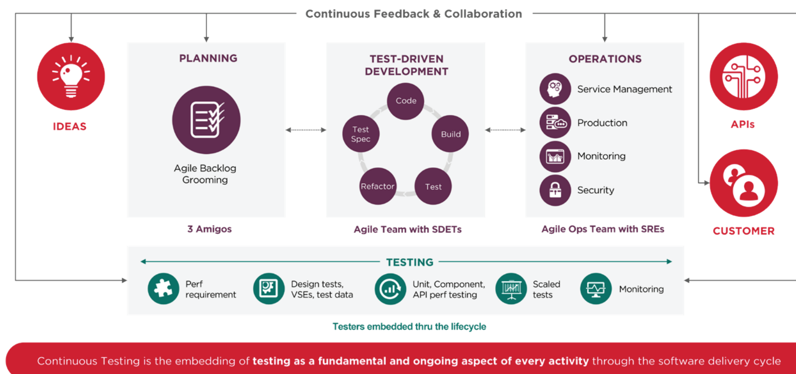
We have seen how Continuous Testing (CT) addresses the bottleneck problem from a functional testing perspective. When applications are architected using a component-based approach (such as using micro-services), it is possible to effectively implement CT for performance testing of such applications. The key to enabling Continuous Performance Testing (CPT) for micro-services-based applications is *being able to test and scale each component in isolation. This is especially applicable for modern cloud-native applications.*

Most literature on micro-services testing (for example, this canonical approach from Martin Fowler) seems to focus primarily on continuous functional testing, and considerably less on continuous performance testing. In this white paper, I will describe the full lifecycle implementing end-to-end continuous performance testing for micro-services.

Note that it is possible to implement similar continuous performance testing for monolithic applications as well, but is less elegant and more onerous.

WHAT IS "CONTINUOUS" PERFORMANCE TESTING?

"Continuous" Performance Testing (CPT) derives from the principle of "Continuous Everything" in DevOps. It is a subset of Continuous Testing (CT), where performance testing needs to happen across the different phases of the Continuous Integration/Delivery (CI/CD) lifecycle as opposed to a single "performance testing phase", see Figure below. CPT is a key enabler for Continuous Delivery (CD), as we will discuss in the next sections.



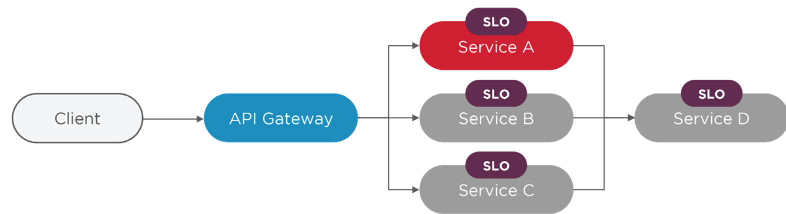
KEY PRACTICES FOR CONTINUOUS PERFORMANCE TESTING

As mentioned before, the key to enabling CPT for micro-services-based applications is: being able to specify performance requirements at the component level, the ability to test and scale each component in isolation. This allows us to run frequent shorter-duration performance tests, as well as test for scalability using smaller scale test profiles – which is what is needed for CPT.

For example, if we have an application that comprises Services A, B, C, and D (see Figure below), each with its own service level objective (or SLO, see the Requirements practice below), we can test each component quickly in isolation, in addition to API-based x-service system tests, and end-user journey tests.

BROADCOM SOFTWARE

KEY PRACTICES FOR CONTINUOUS PERFORMANCE TESTING (CONT'D)



Performance testing components in isolation gives us the ability to test early and often using smaller time slots, without having to rely on long duration traditional performance tests. If the component tests do not pass, the higher order tests will not pass as well, so there is no need to run them. Thus, we save time and resources that are critical from a CD perspective.

In addition to the above, we combine the key practice of *change impact testing,* where we test only those components (and higher order transactions and user journeys) that have been impacted by some change. This helps to further reduce the time and effort for performance testing that is key to CD.

We will expand on each step of the CPT lifecycle later in this white paper. But first, let's discuss the key practices required to support CPT.

1) Capture and specify performance requirements

Good testing starts with good requirements. This principle applies to performance testing as well, however, we often see performance requirements that are not as rigorously specified as functional testing requirements. Performance requirements are considered part of non-functional requirements (NFR) of an application/system.

There are different performance testing requirements that can be captured at different levels of system/application granularity. In general, there are three major types of performance requirements:

(a) Those defined at the enterprise level.

Typically these are *performance policies,* often defined by the customer experience team. For example: maximum response time for web applications

(b) Those tied to functional requirements or product features.

These are generally defined by the product manager or product owner. They are typically attached to the functional requirements (e.g., user stories or features, or user interaction scenarios) as *performance constraints* (or sometimes as acceptance criteria).

For example, a user story such as "As a customer, I want to be able to update the search criteria for products, so that I can automatically refine the product selection" could have an NFR such as: "The search results must be updated within one second on screen"

It is important that such performance criteria are considered as part of the "definition of done" for such requirements. Which means that we must develop, store (and execute) tests to validate the performance requirements.

Stories are often thrown away after the end of an agile iteration; however, validation tests will persist, and may be used for regression tests in subsequent iterations.

KEY PRACTICES FOR CONTINUOUS PERFORMANCE TESTING (CONT'D)

(c) Those tied to application system components.

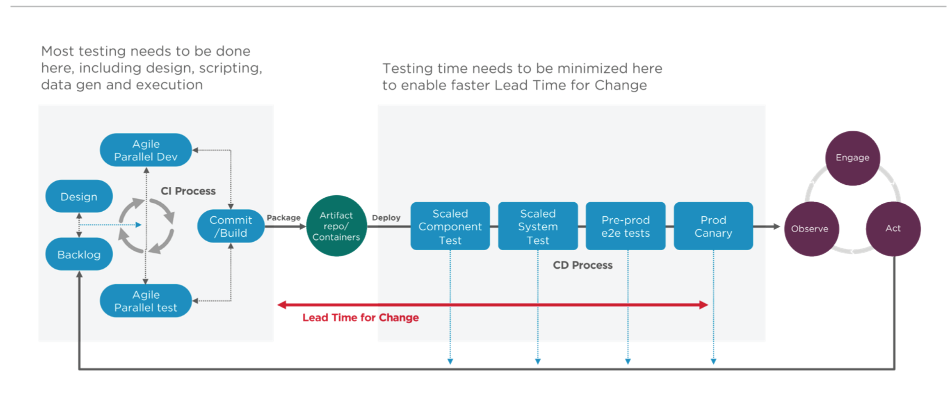
These are generally defined by the product owner, architect, or site reliability engineer, and are tied to different application components. For example, these could be defined for the whole application, or for specific components (like app services or APIs or even specific application functions).

Often *performance requirements attached to key application components (such as services or APIs) are specified in the form of* Service Level Objectives (SLOs) with an accompanying SLA (where applicable). For example, performance SLOs for a Search component could be something as follows: "Latency: 90% of requests take <400ms, 99% of requests take <850ms; Availability > 97%, and Thruput: 24K/sec".

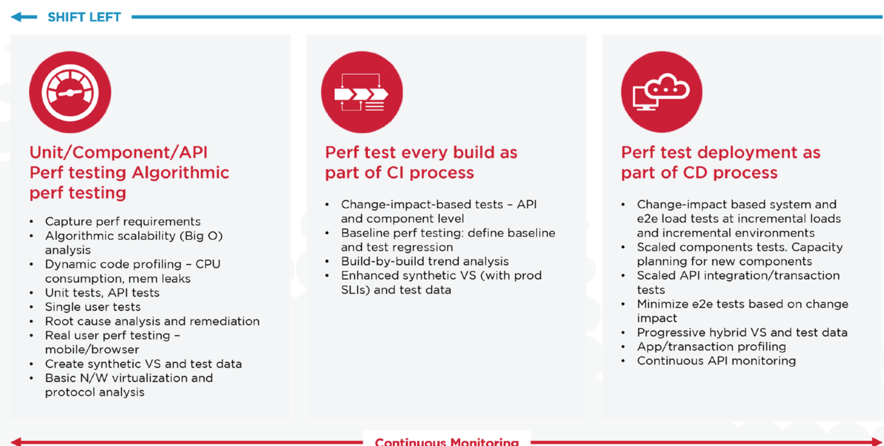
Unlike functional or product features, where performance criteria attached to "logical" stories may be thrown away, SLOs which are described for "physical" components, will persist, and therefore will be available as the basis of performance tests throughout the life of the component. Note, however, that such SLOs may evolve over time depending on business/technical needs.

2) Shift-left Performance Testing activities

As with most activities in the CT lifecycle, CPT also requires that most of the performance testing work (such as test specification, design, generation and execution) "shifts left," so it mostly happens during the CI part of the DevOps lifecycle. This minimizes the delay that CPT processes can cause during the CD part of the lifecycle, where minimizing the Lead Time for Change is of utmost importance (see figure below). This is enabled by adherence to the testing pyramid (see next section).



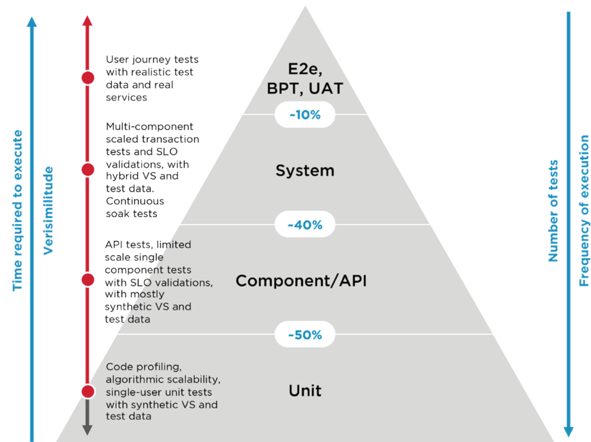
A summary of key performance testing activities that are performed during the CI and CD process are shown in the Figure below. These activities are mapped to the CPT lifecycle phases that we describe later.



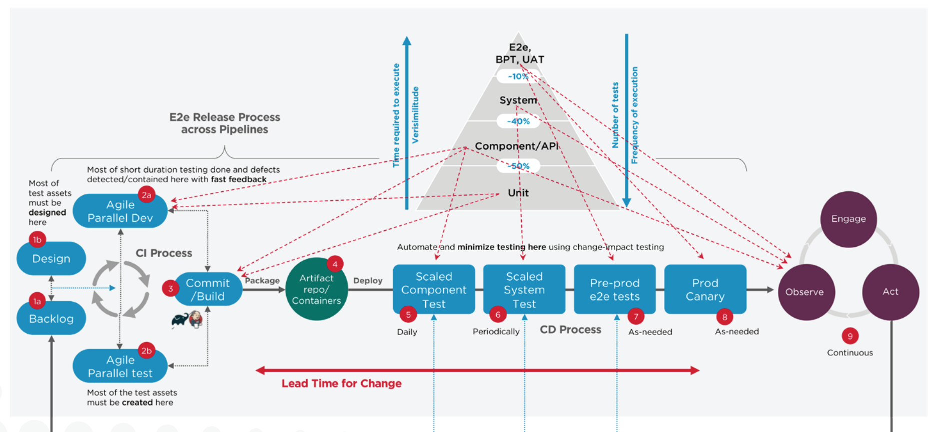
KEY PRACTICES FOR CONTINUOUS PERFORMANCE TESTING (CONT'D)

3) Adhere to the Test Pyramid for Performance Testing

Shift-left of testing is supported with the use of the test pyramid. We generally see the test pyramid used in the context of functional testing, but it applies just as well to performance testing of component-based applications. See a figure of the performance testing pyramid below.



It means that most of the (shorter duration) performance testing needs to be conducted rigorously at the unit/component/API levels (as part of the Continuous Integration process) and fewer longer duration tests executed at the system and e2e levels (as part of the Continuous Delivery process), and that too only as needed. This helps ensure we have lesser slowdown on the "Lead Time for Change" metric, which is indicative of velocity of change deployment. See Figure below. This is quite a departure from the classic performance testing approach, where more of the long duration testing is done (at the system or e2e level) before release.



If it does not scale at the unit level/component level, it won't scale e2e! Shift focus to unit/component tests

KEY PRACTICES FOR CONTINUOUS PERFORMANCE TESTING (CONT'D)

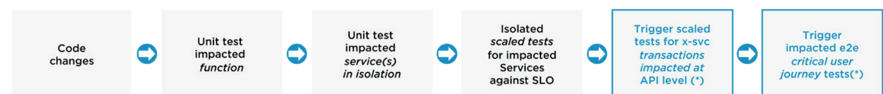
4) Leverage Change Impact Testing

As we have mentioned before, the key to reducing testing elapsed time in the context of the CD process within CPT is to reduce the amount of tests that need to be done. A key way to do this is to leverage change impact testing – i.e., focus testing principally on those parts of the application that have changed. This is an important part of the Design phase of the CI/CD lifecycle.

There are various types of change impact techniques that we may leverage. We will describe two that we recommend here.

(a) Impact analysis based on code change

This is an "inside-out" approach to analyzing impact based on changes in the code of the application components. See Figure below.



In this approach, we *leverage mapping between tests and code executed* to analyze what tests need to run based on specific code changes. Every time a body of code is changed (e.g., as part of development activities and pull requests), the system flags the set of tests that have been impacted by the change. This often helps to cut down testing by over 50%.

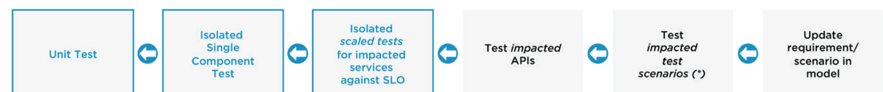
Examples of tools that support this include: TestAdvisor, Microsoft Visual Studio, etc.

This technique is more developer friendly and is more applicable for tests in the lower part of the pyramid (or the more to left of the CI/CD lifecycle), i.e., unit, component and scaled component tests. As these tests need to be run most frequently, even a small savings in the amount of tests run will add up to a significant savings in testing cost and time.

We may in fact extend the benefit of this technique towards the higher order tests (marked with broken outline), however, they are better served by the "outside-in" approach described next.

(b) Impact analysis based on requirements or behavioral change in application

This is an "outside-in" approach to analyzing impact based on changes in the code of the application components. See Figure below.



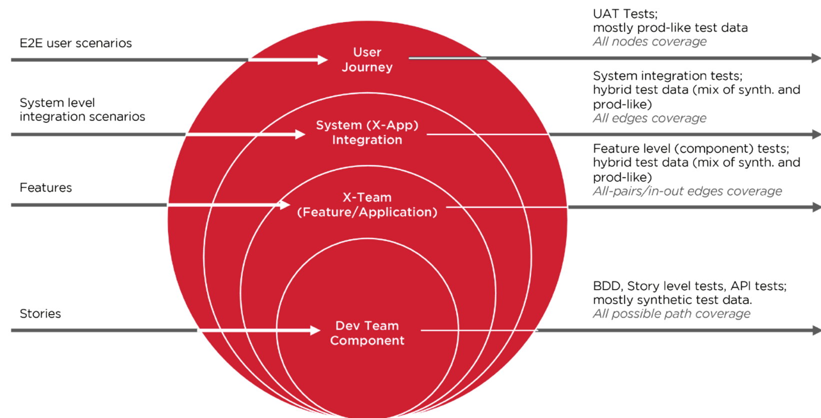
In this approach, we *leverage mapping between tests and requirements (user flows)* to analyze what tests need to run based on specific flow changes. Every time a requirement (or flow) is changed (e.g., as part of backlog grooming), the system flags the set of tests that have been impacted by the change. This often helps to cut down testing by over 70%.

Examples of tools that support this include Agile Requirements Designer (ARD) where requirements are modeled as behavioral flows through the system.

This technique is more tester friendly and is more applicable for tests in the upper part of the pyramid (or the more to right of the CI/CD lifecycle), i.e., journey tests and transaction tests. As these tests are typically resource and time intensive, use of this technique has a significant impact on the testing effort and time during the part of CI/CD lifecycle where Lead Time for Change is key.

KEY PRACTICES FOR CONTINUOUS PERFORMANCE TESTING (CONT'D)

Tools like ARD not only allow the modeling of requirements at different levels of granularity (for change impact analysis at those levels), but provide techniques to significantly optimize the amount of testing based on various optimization techniques, see Figure below. A combination of both change impact analysis and optimization often yields test reduction of greater than 80%.



We may in fact extend the benefit of this technique towards the lower order tests (marked with broken outline), however, they are better served by the "inside-out" approach described earlier.

5) Aggressively leverage Continuous Service Virtualization

Service virtualization (SV) is a key enabler for all types of testing, and especially so for CPT of component-based applications. SV allows us to not only virtualize dependencies between components, but more importantly allow us to correctly emulate the performance of the dependent component thereby allowing proper testing of the component under test. Virtual services also allow us to easily and quickly setup and configure ephemeral test environments that are needed for CPT.

While using SV in the context of CPT, we need to leverage the corresponding techniques around "Continuous" Service Virtualization. Please see my blog on that subject here.

6) Aggressively leverage Continuous Test Data Management

Test Data Management (TDM) is another key enabler for all types of testing, and especially so for CPT. TDM allows us to automate the creation and provision of data – which may sometimes be quite voluminous – for performance testing activities. It is a key capability that allows us to easily setup and configure ephemeral test environments that are needed for CPT.

While using TDM in the context of CPT, we need to leverage the corresponding techniques around "Continuous" Test Data Management. Please see my blog on that subject here.

We recommend use of both Continuous SV and Continuous TDM together to support the needs of CPT, since SV helps to reduce the burden of the more onerous TDM activities.

KEY PRACTICES FOR CONTINUOUS PERFORMANCE TESTING (CONT'D)

7) Leverage Continuous Monitoring and Analytics

Monitoring is a key component of all performance testing, and especially so for CPT. This is not only required for understanding performance bottlenecks (and follow-up tuning activities), but the high frequency and volume of tests (and resulting data) during CPT require the use of data analytics and alerting capabilities. For example, data analytics is required to establish performance baselines, report regressions, and generate alerts for other anomalous behaviors.

For component-based applications, we also need specialized solutions tools for monitoring of containers that deploy application and test assets.

In fact, we recommend the use of Continuous Observability solutions to not only analyze performance data, but also provide proactive insights around problem detection and remediation. This is especially important for debugging issues like tail latencies and other system issues unrelated to application components.

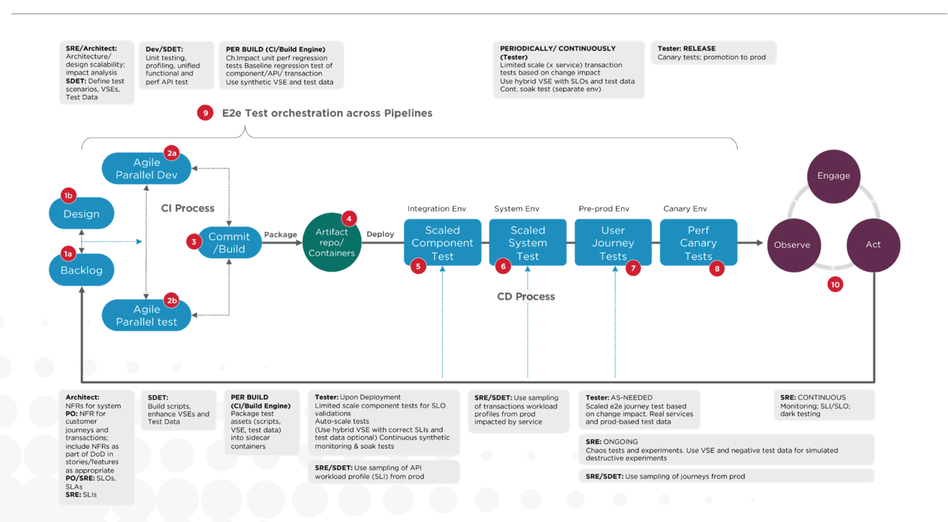
8) Integrate testing processes with CI/CD processes

CPT requires tests to be run frequently in response to change events (e.g., code updates, builds, deployments, etc.) with a variety of accompanying test assets (like test scripts, test configurations, test data, virtual services, etc.) in dedicated test environments. It is practically impossible to manage all of this processing manually in the context of CI/CD lifecycle.

It is therefore key that all CPT activities be integrated with a CI/CD orchestration engine that triggers the provisioning of environments, deployments of application components and test assets, execution of tests, capturing and communication of test results, and cleanup after completion. For component-based applications that are deployed in containers, we may package test assets (including test data) in sidecar deployment containers and deploy them alongside application containers.

PULLING IT ALL TOGETHER: CONTINUOUS PERFORMANCE TESTING LIFECYCLE

The following Figure summarizes the different activities in a typical CPT process across the different stages of the CI/CD pipeline along with typical personas who perform them.



BROADCOM SOFTWARE

PULLING IT ALL TOGETHER: CONTINUOUS PERFORMANCE TESTING LIFECYCLE (CONT'D)

These activities are summarized below:

1) Step 1(a): Requirements and Backlog

CPT starts with well-defined performance requirements. As we discussed earlier, this can either be performance constraints attached to functional requirements such as user stories and features or transactions, or customer journeys (typically defined by Product Owner), or SLOs defined for service components (typically defined by Site Reliability Engineers)

2) Step 1(b): Agile design

The CPT activities at this stage support the needs of Development and subsequent CI/CD phases. This includes:

(a) Impact analysis of requirements changes to identify key user scenarios and flows to be tested (Software Development Engineer in Test or SDET/Tester)

(b) Impact analysis of system configuration changes to identify key system components that need to be tested (SRE or System Architect)

(c) Definition/update of impacted test scripts based on requirements/flow change impact analysis (SDET/Tester)

(d) Definition/update of virtual services and test data needed for the tests (SDET/Tester)

(e) Definition/update of test environment specifications for various CI/CD environments (SDET/SRE)

3) Step 2(a): Agile Parallel Development

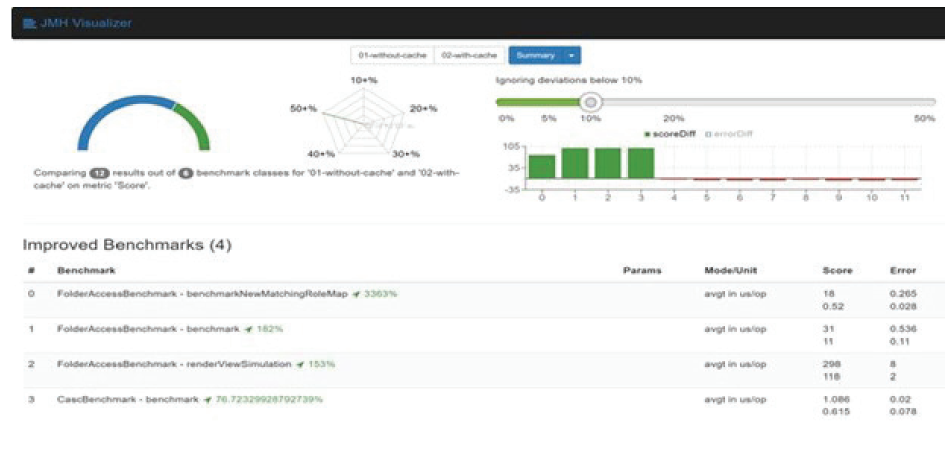
During development, developers/SDETs conduct unit and component level performance testing of the code/component being built (or updated). Some of the key practices include:

(i) Code profiling. This helps us understand performance characteristics of the application at the code level. It helps to identify performance bottlenecks, critical paths, resource usage, memory leaks, threading behavior, and other characteristics. This helps developers detect (and fix) performance issues as early as possible. Code profiling is supported by a variety of tools such as cProfile, jProfiler, Bazel, etc.

See Figure below for examples of outputs from code profiling.

PULLING IT ALL TOGETHER: CONTINUOUS PERFORMANCE TESTING LIFECYCLE (CONT'D)

(ii) Unit performance testing at the function/method level. This is typically done when a specific method is identified during profiling as a bottleneck, or is heavily used or involves significant processing. We recommend running these tests when significant changes occur in the function (see later section on code change impact testing), using a controlled environment (such as a dedicated virtual machine). Tools like JUnitPerf can be used for this. See Figure below for examples of outputs from such tests.



4) Step 2(b): Agile Parallel Testing

This is an important stage where SDETs and testers create or update the performance test scenarios (and accompanying test assets such as virtual services and test data) and define/update the test asset packaging and deployment configurations for the rest of the CI/CD lifecycle based on change impact analysis as described earlier.

There are various approaches to defining performance tests for acceptance, integration, system and e2e test scenarios.

For performance requirements attached to functional requirements, we may use Behavior Driven Development (BDD) to define performance acceptance cases in Gherkin format. For example, the baseline acceptance test for an API may be as follows:

Scenario: API Tests with small load
Given API Query http://dbankdemo.com/bank
And 100 concurrent users
And the test executes for 10 minutes
And has ramp time of 5 minutes
When Response time is less than 1ms
Then Response is Good

The Gherkin feature file can then be translated into YAML that may be executed using tools like jMeter.

For System and E2E test scenarios, our recommendation is to define those using model-based testing tools like ARD. This allows us to conduct automated change impact analysis and precise optimization, and generate performance test scripts (and test data) directly from the model that can be executed with tools like jMeter.

PULLING IT ALL TOGETHER: CONTINUOUS PERFORMANCE TESTING LIFECYCLE (CONT'D)

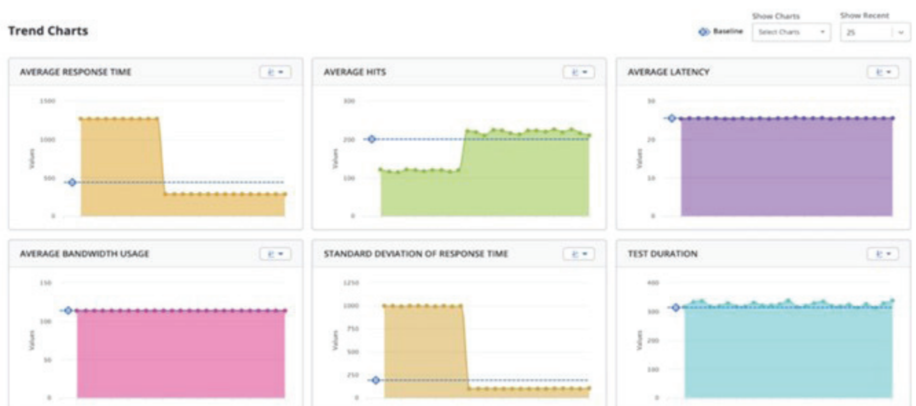
5) Step 3: Testing after each build

As part of every build, we recommend running baseline performance tests on selected components (based on the change impact analysis as described earlier). These are short duration, limited scale (using a small number of virtual users) performance tests on a single instance of the component at the API level, to establish a baseline, *assess build-over-build regression in performance, and provide fast feedback to the development team.* A significant regression may be used as a criteria to fail the build. Tests on multiple impacted APIs can be run in parallel, each in its own dedicated environment. Tools such as jMeter and Blazemeter may be used for such tests.

The profile of such a test would depend on performance requirements of the component. For example, for the Search component that we discussed earlier, we could set the test profile as follows:

Thruput = 24K/sec TPS
Number of virtual users = 100
Wait time = 0.5 sec
Duration: 10 mins
Warmup time: about 15 sec
Key measures: thruput (TPS), Response time (P95/P99), CPU/memory usage, etc.

See Figure below for examples of outputs from such tests that show baselines and trend charts.



If a component has dependencies on other components, we recommend the use of virtual services to stand-in for the dependent components so that these tests can be spun up and executed in a lightweight manner within limited time and environmental resources.

6) Step 4: Packaging of Test Assets for Deployment

A variety of test assets (such as test scripts, test data, virtual services, and test configurations) are created in the previous steps and need to be packaged for deployment to the appropriate downstream environments for running different types of tests. As mentioned above, for component-based applications — where micro-services are typically deployed as containers — we recommend packaging these test assets as accompanying sidecar containers. The sidecar containers are defined appropriately for the type of tests that need to be run in specific CI/CD environments. This is an important aspect of being able to automate the orchestration of tests in the pipeline.

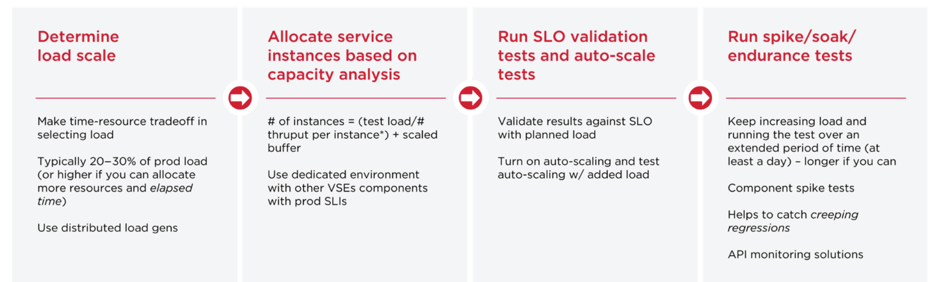
PULLING IT ALL TOGETHER: CONTINUOUS PERFORMANCE TESTING LIFECYCLE (CONT'D)

7) Step 5: Scaled Component Test in Integration Environment

Scaled component tests are conducted on isolated impacted components (based on change impact analysis) to test for SLO conformance and auto-scaling.

This is a single-component isolated performance test that is typically run at 20% to 30% of the target production load in a dedicated test environment. We may run this at higher loads, but that will take a longer time (thereby increasing the Lead Time for Change, and increasing test environment resources), so run at the highest possible load keeping in mind the maximum time allotted to run the test. Typically *scaled component tests should be limited to no more than 30 minutes,* so that delays to the CD pipeline are minimized.

The typical process for running such a test is shown in the Figure below.



After the SLO validation tests are completed (step 3), the results are reported and the CD pipeline is progressed. However, we recommend running spike and soak tests (step 4) over a longer duration of time, often greater than a day, (without holding up the CD pipeline) which often help us catch creeping regressions and other reliability problems that may not be caught by limited duration tests.

Another key item to monitor during these tests are *tail latencies*, which are typically not detected in baseline tests described above. We need to closely monitor the P99 percentile performance.

Scaled component tests should leverage service virtualization to isolate dependencies on dependent components. Such virtual services must be configured with response times that conform to their SLOs. See more on this in the section on the use of service virtualization.

To minimize test data provisioning time, these tests need to use hybrid test data – i.e., a mix of mostly synthetic and some production-like test data (typically 70:30 ratio).

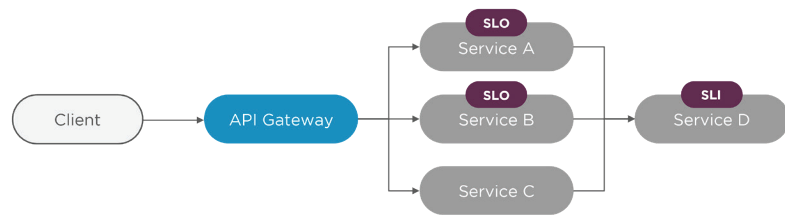
Although distributed load generators can be used to account for network overheads, the use of appropriate network virtualization significantly simplifies the provisioning of environments for such tests.

PULLING IT ALL TOGETHER: CONTINUOUS PERFORMANCE TESTING LIFECYCLE (CONT'D)

8) Step 6: Scaled System Test in System Environment

Scaled system tests are API-level transaction tests (based on change impact analysis) across multiple components to test for transaction SLO conformance and auto-scaling. These tests should be run after functional x-service contract tests have passed.

A transaction involves a sequence of service invocations (using the service APIs) in a chain, see Figure below, where the transaction invokes Service A, followed by B and C, etc. These tests help to expose communication latencies and other performance characteristics over and above individual component performance. Distributed load generators should be used to account for network overheads (or with suitable network virtualization).

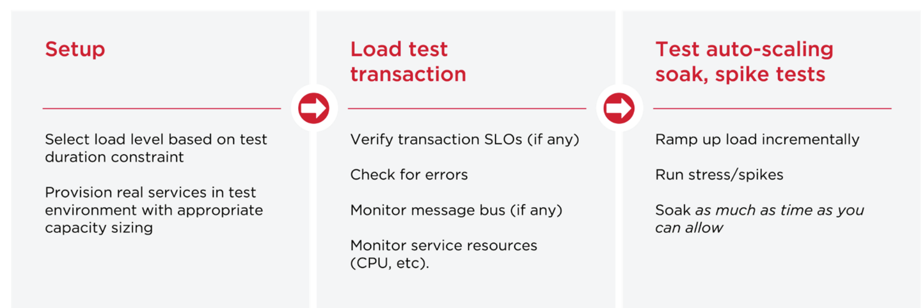


Since such tests take more time and resources, we recommend that such tests be limited to run only periodically and that too for critical transactions that have been impacted by some change.

Such tests also need to be run with real components (that have been impacted by the change), but use virtual services for dependent components that have not.

Also, to minimize test data provisioning time, these tests need to use hybrid test data – i.e. a balanced mix of synthetic and production-like test data (typically 50:50 ratio).

The typical process for running such a test is shown in the Figure below. Depending on cycle time availability in the CD pipeline, we need to limit the amount load level



BROADCOM SOFTWARE

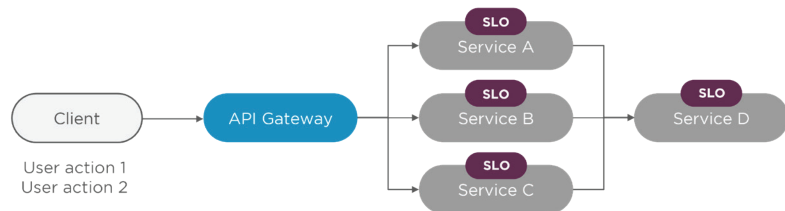
PULLING IT ALL TOGETHER: CONTINUOUS PERFORMANCE TESTING LIFECYCLE (CONT'D)

System tests are probably the most challenging performance tests in the context of CPT, since they cross component boundaries. At this stage of the CD pipeline, we should be confident that individual components that have been impacted are well tested and scale correctly. However, additional latencies may creep from other system components, such as the network, message buses, shared databases, other cloud infrastructure and aggregation of tail latencies across multiple components, which typically occur due to some other system component outside of the application components.

For this reason, we recommend that other system components also be performance tested individually using the CPT methodology described here. It is easier to do so for systems that use infrastructure-as-code, since changes to such systems can be detected (and tested) more easily. Leveraging Site Reliability Engineering techniques are ideal for addressing such problems.

9) Step 7: E2E User Journey Test in Pre-prod Environment

In the pre-prod environments, we recommend running *scaled e2e user journey tests* for selected journeys based on change impact analysis. These tests measure customer experience as perceived by the user. See Figure below.



Since these tests typically take more times and resources to run, *we recommend that these be run sparingly and selectively in the context of CPT,* for example, when multiple critical transactions have been impacted, or major configuration updates have been done.

Such tests are typically run with real service instances (virtual services may be used to stand in for dependent components if they are not part of the critical path), realistic test data, use a realistic mix of user actions (typically derived from production usage), and use real network components with distributed load generators. For example, an e-commerce site has a mix of varied user transactions such as login, search, checkout, etc., each with different loading patterns.

As in System testing described above, it is vital to closely monitor other system components (more so than application components) during these tests.

10) Step 8: Performance Canary Tests in Canary Production Environment

Performance canary tests are similar to functional canary tests, except that they validate system performance using a limited set of users. Such tests are a great way to validate performance scenarios that are difficult or time-consuming to run in pre-production environments, and provide valuable feedback before the application changes are rolled out to a wider body of users.

Canary environments can also be used for chaos tests and destructive experimental testing to understand the impact on application performance. Component-based applications lend themselves very well to controlled chaos experiments, since we can simulate failure at various levels of granularity to enable us to understand and resolve problems faster. Some organizations even use virtual services in chaos environments to easily simulate failure conditions.

PULLING IT ALL TOGETHER: CONTINUOUS PERFORMANCE TESTING LIFECYCLE (CONT'D)

11) Step 9: Continuous Performance Test Orchestration in CI/CD pipelines

As we have mentioned before, one of the key requirements of CPT is being able to orchestrate all of the key testing processes and steps (described above) in an automated manner. For component-based applications, this is a complex problem to manage for thousands of changes occurring across hundreds of components (and their corresponding deployment pipelines).

12) Step 10: Continuous Performance Monitoring in Production

For component-based applications, production monitoring additionally involves tracking the SLIs, SLOs and SLAs at the component, transaction and business services levels.

IN CONCLUSION

This white paper has provided an overall approach for Continuous Performance Testing practices for component-based applications. As you can tell, component-based applications are particularly well suited for this approach.

Testing however is an activity, while *quality is the real outcome that we desire.* The goal of CPT is to help us continuously ensure that quality outcomes such as reliability can be continuously assured. We call this Continuous Reliability (CR).

BROADCOM SOFTWARE

About Broadcom Software

Broadcom Software is a world leader in business critical software that modernizes, optimizes, and protects the world's most complex hybrid environments. With its engineering-centered culture, Broadcom Software has an extensive portfolio of industry-leading infrastructure and security software, including AIOps, Cybersecurity, Value Stream Management, DevOps, Mainframe, and Payment Security. Our software portfolio enables scalability, agility, and security for the largest global companies in the world.

For more information, visit our website at: software.broadcom.com

Copyright © 2022 Broadcom. All Rights Reserved. Broadcom and other trademarks are the property of Broadcom. The term "Broadcom" refers to Broadcom Inc. and its subsidiaries. Other trademarks are the property of their respective owners.