



WHITEPAPER

Performance Testing Best Practices Guide

Performance testing is a software testing practice that determines the responsiveness, reliability, scalability, and resource consumption of a software server application when it's subjected to a stream of concurrent service requests called a workload.

The ultimate goal of performance testing is to predict the behavior of a server application in a production environment. To accomplish this, the server application under test (AUT) is deployed in a simulated production environment and subjected to synthetic workloads generated by a performance testing tool.

Performance testing is often an iterative process of finding and resolving performance bottlenecks, race conditions, and operating system and network configuration issues before reaching a state where the server application meets its performance requirements. In some cases, this process goes a step further to build a performance model of the system under test that can reasonably predict its behavior in various types of loads and deployment environments. In this sense, it's closely related to capacity planning.



WHAT ARE THE DIFFERENT TYPES OF PERFORMANCE TESTS?

When a server application is deployed in a production environment, it can be subjected to various types of workloads.

- » An average anticipated load
- » A load that greatly varies over time
- » A peak load
- » A load that puts the system at a breaking point

A server application must sustain such production workloads for extended periods of time without restart. The goal of performance testing is to emulate such loads to prove, or at least give reasonable assurances, that application performance is production worthy.

PERFORMANCE TEST PARAMETERS

During the process of [performance testing](#), a server application is subjected to various types of simulated workloads, which include the major parameters listed below.

Load intensity is the number of service requests per unit of time, such as hits per second (HPS) or hits per minute (HPM).

Load concurrency is mostly applicable to testing stateful applications. It's equal to the number of concurrent users the application is servicing. Each user is allocated a set of resources to support the stateful nature of the application, so the greater the concurrency of the load, the more system resources the application consumes. Setting proper concurrency is critical and often overlooked in favor of intensity alone. Even a low-intensity load with high concurrency can break an improperly designed or configured server application.

Load level is a combination of load intensity and load concurrency.

Test duration of a performance test is determined by the duration of its three major sections:

- » Ramp up warms up the system. Performance engineers and testers gradually increase the load until it reaches a desired level.
- » Workload is a major part of the test. Teams use it to measure system performance at a desired load level. The duration of it depends on the type of the performance test, described below.
- » Ramp down allows the system to process the remaining requests and release resources before a clean shutdown.

User profile distribution is a request flow pattern for a certain application user category. It's defined by the request types and the timing between requests, called pacing. All but the simplest server applications will have multiple user profiles. Here are examples:

- » Buyer
- » Regular user
- » Seller
- » Advanced user
- » Administrator

Different user profiles will have different impacts on the application. A performance testing workload will emulate groups of users of different profiles. Teams should assign each profile an appropriate share of the total workload to reflect the expected production workload.

PERFORMANCE TEST TYPES

Based on the value and relation between the parameters described above, teams can divide performance tests into the following major types.

Load tests subject the application to an expected average load. The workload section of a load test should have a duration that will provide minimal reasonable assurance that the application under test is production ready and can vary from a few tens of minutes to several hours.

Stress tests put the application through an excessive load level that would cause the system to start failing. The goal of this test is to establish the upper load limit that the system can handle. It also tests whether the system under such load fails gracefully, for instance, by returning well-formatted error responses rather than suffering an unrecoverable crash.

Endurance or soak tests subject the application to a long-lasting average load level. The goal of this test is to check system reliability and stability over a lengthy period of time. During endurance testing, users pay special attention to checking for slow resource leaks, such as memory, network connections, or file handle leaks, which can eventually put the system out of resources. In addition, endurance tests can uncover rare failures, like those caused by race conditions and slow buildup in the system state. These may eventually result in a failure, for instance, roundoff error accumulation. An example of an endurance test would be a 48-hour performance test run over a weekend.

Smoke or baseline tests are relatively short application tests of a low to medium load level to establish the basic ability of the system to handle concurrent requests. Teams may run such tests regularly as a part of the continuous integration (CI) testing process. Because these tests don't put the system under significant load, the results teams can use the results as a baseline for the best system response times under favorable conditions.

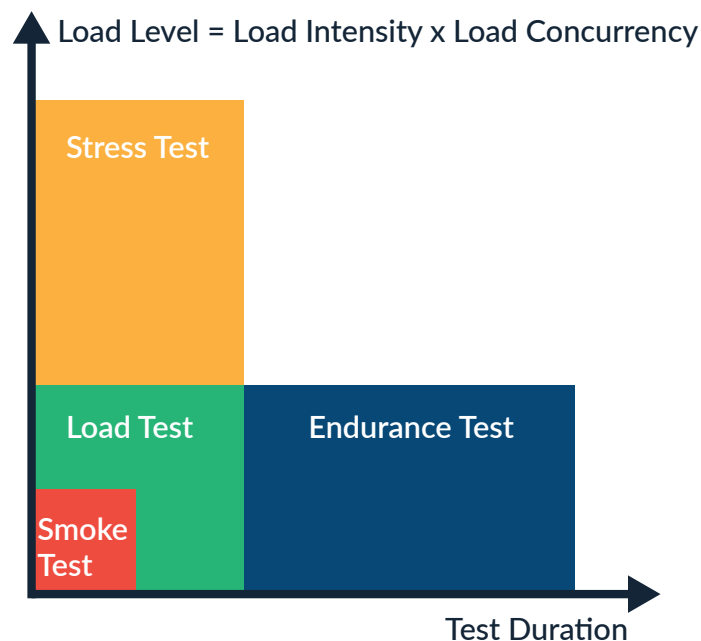


Figure 1:
A visual interpretation of
the load levels and test
durations for the four
major performance test
types.

In addition to the four major performance test types described above, users can apply other tests as needed based on the specific requirements of the application. Here are two examples.

- » **Spike tests** subject the application to high load bursts followed by periods of a normal or low-level load. The goal of this test is to check whether the system can quickly allocate resources to process a sudden workload increase and whether it subsequently releases the extra resources after the load falls back to the off-peak level.
- » **Scalability or capacity tests** establish how well the application can handle an increase in load when it's given a proportionate increase of system resources. For instance, will an application that's approaching a system resource bottleneck, like CPUs, under a certain load be able to handle double that load if it is given twice as many CPUs or an additional machine of the same capacity?

The following table summarizes performance test types and the relationships between their parameters.

Performance Test Type	Concurrency	Intensity	Duration
Load	Average	Average	Average
Stress	High	High	Low-Average
Endurance or Soak	Average	Average	High
Smoke or Baseline	Low	Low	Low
Spike	Low-High-Low	Low-High-Low	Average
Scalability or Capacity	Low to High	Low to High	As Needed

The value of a performance test is directly proportionate to how closely the parameters of the synthetic workload emulate the real one. Most commonly, information about the parameters of the anticipated workload, such as its intensity, concurrency, major user categories, and call patterns, comes from the following sources.

- » Log analysis of production deployments of prior versions of the application.
- » Reasonable predictions based on the nature of the application. For instance, a proper stress test would be critical for a web application in support of a marketing campaign for a product with a fixed announcement date.
- » Business information, such as the total number and types of customers, as well as their distribution across time zones.



CREATING A CONFIGURABLE PERFORMANCE TESTING ENVIRONMENT

Many organizations only perform tests against a subset of the potential conditions that users could experience. Agile teams recognize that they need to run performance tests against a variety of workloads to meet user expectations.

Performance and load testing tools are a check engine light on system performance. They test under regular and extreme loads to find any faults and ensure that your application can take the heat. Test automation makes it easier and faster to run a combination of performance testing steps in parallel.

A configurable testing environment ensures that users can run performance tests under a broad range of operating conditions. Service virtualization can create reusable application test environments to validate that the software can handle the full capacity needed for high-demand response times, such as holiday sales. Often, the cost of maintaining a live performance environment is close to equaling a company's production environment. Service virtualization simulates the dependencies of the APIs in the test environment, reducing the complexity and cost of performance testing.

HOW TO SET UP & RUN PERFORMANCE TESTS

The process of performance testing software server applications consists of the following major stages.

- » Set up a performance testing environment.
- » Prepare and execute performance tests.
- » Analyze performance test results.
- » Automate performance test execution and results analysis.

SET UP A PERFORMANCE TESTING ENVIRONMENT

This stage includes getting the server application properly deployed, configured, and ready to process a simulated workload. Here are the major steps to follow to prepare the application under test (AUT) for a performance test.

- 1. Prepare the AUT deployment environment.** Teams can execute a performance test against either a simulated application deployment in a testing or pre-production environment or against a real production deployment. When a performance test runs against a simulated deployment, it's important to minimize differences in hardware, network setup, and software between the test and production environment.

In many cases, it's impractical or impossible to fully match simulated and production deployments. For this reason, teams may run the final performance tests during off-peak hours or against a production environment that hasn't yet been exposed to the outside world.

2. **Use service virtualization to simulate missing application dependencies.** The architectural shift from monolithic server applications towards microservices causes modern applications to have multiple dependencies. Many such applications can only be assembled in full in a production environment. For such applications, performance tests are run against the application subsystems—individual services or collections of services—while virtualizing their dependencies.
3. **Set up load generation infrastructure.** It's common for performance testing software to consume a comparable amount of computing resources to the application under test, especially if the performance testing application does extensive response validation. As with the application under test, teams must monitor the load generating application and infrastructure to make sure they do not run short of resources during a performance test. Resource starvation during performance testing of an application can cause failure to generate the required synthetic workload and inaccuracy of time interval measurements, which will produce distorted results.

PREPARE & EXECUTE PERFORMANCE TESTS

At the core of a performance testing project lies a performance test suite, otherwise called a performance test script. It replays the actions of a real user against the application under test. It's a repurposed functional test tailored to the specific performance testing requirements.

In simple terms, performance testing can be thought of as repeated, concurrent execution of functional tests while monitoring the application under test and the computing environment in which it's deployed. While this statement reflects the essence of performance testing and highlights one of its best practices—the reuse of functional test assets—one should not just take an existing AUT functional test suite and run it in a performance testing application because a purely functional test suite and a functional test suite designed for repeated, concurrent execution in a performance testing tool (a performance test suite) while testing the same application have different goals.

The main goal of a purely functional test suite is to verify as many usage patterns as is practical for a given application. In addition, in some cases, teams can use functional tests together with unit tests to increase the code coverage of the AUT.

A performance test suite, on the other hand, aims to replicate the most common AUT usage patterns. To achieve these different goals, a pure functional test suite will typically include many more tests than a performance test suite. On the other hand, a performance test suite will use parameterization to a much greater degree because during performance testing, functional tests should be invoked with a much wider range of request parameter values to emulate the variety of realworld requests. Failure to do so can result in an unrealistically shallow performance test scenario, which may cause the AUT to serve cached replies in response to a narrow range of repeating requests.

The high parameter variability requirement applies not only to the request parameters related to the AUT business logic but also to the parameters used in the communication protocols on different OSI (open system interconnect) layers, such as using different concurrent user sessions or different client IP addresses.

Once a performance test suite has been created that satisfies the listed above requirements, the following additional steps should be completed before executing a performance project.

1. **Create virtual user profiles.** A virtual user (VU) is a construct within a performance testing application that emulates the behavior of a real application user of a certain type, or more generally, emulates a distinct application usage pattern. A virtual user profile is a template used to create virtual users of the same type. A VU profile typically contains a performance test suite or its subset that represents VU actions and a parameter that regulates the frequency with which these actions are invoked. In different performance testing applications, they might be called "think time," "delay," or "pacing." A typical performance testing configuration will contain many VU profiles.
2. **Configure performance test scenarios.** A performance test scenario is a construct in a performance testing application that represents a test type, such as a stress test, endurance test, and the like. It allows performance engineers and testers to configure the relative parameters, such as duration of the test, load level, and virtual user profile distribution over time. Different performance testing applications may have different names for this construct, but all of them will have controls to configure these major performance test parameters.

A typical high-level question that a performance test must answer is: Can an application or component under test support X number of users or hits per second while maintaining the pre-defined service level agreements (SLAs) and given specified computing resources?

A performance engineer needs to translate these high-level requirements into performance test acceptance criteria specific to the performance testing application he or she plans to use. Different performance testing tools offer different mechanisms for setting up such performance acceptance criteria. For example, [Parasoft Load Test](#) allows the user to configure quality of service (QoS) metrics that are automatically applied to the performance test results upon the completion of a test to determine performance test success or failure.

QoS metrics are created from QoS metric templates, which represent different AUT key performance indicators (KPI). Below is a table of selected server application key performance indicators and their example acceptance criteria.

Key Performance Indicator (KPI)	Acceptance Criteria
Response time for synchronous applications	Average response time should be less than one second.
Throughput for batch processing applications	Average processing rate should be no less than 1,000 messages per second.
Failure rate	Should be less than 0.01%.
Consumption of system and application resources	Average AUT CPU utilization should be less than 50%.

Once the AUT automated performance acceptance criteria have been configured, they serve as performance regression controls.

- 3. Set up the AUT and system monitors.** Application and system resource monitoring results are used for performance test failure root cause analysis and application scalability evaluation. For example, the root cause of a consistently slow application response time may be high CPU utilization, while occasional response timeouts may be caused by thread synchronization issues inside the AUT. Performance testing tools will typically provide some monitoring. For more detailed monitoring, specialized application performance monitoring (APM) software is used.
- 4. Set up the AUT persistent state.** It is common for a server application to maintain a persistent state, typically in the form of databases, directory services, or file systems. The size and type of data in such persistent data storage can significantly affect application performance. For this reason, prior to the execution of a performance test, the AUT persistent data storage should be populated with data that matches in size and type the data in a production deployment.

After all preparation steps are complete, a performance test is executed and the results are collected for further analysis.



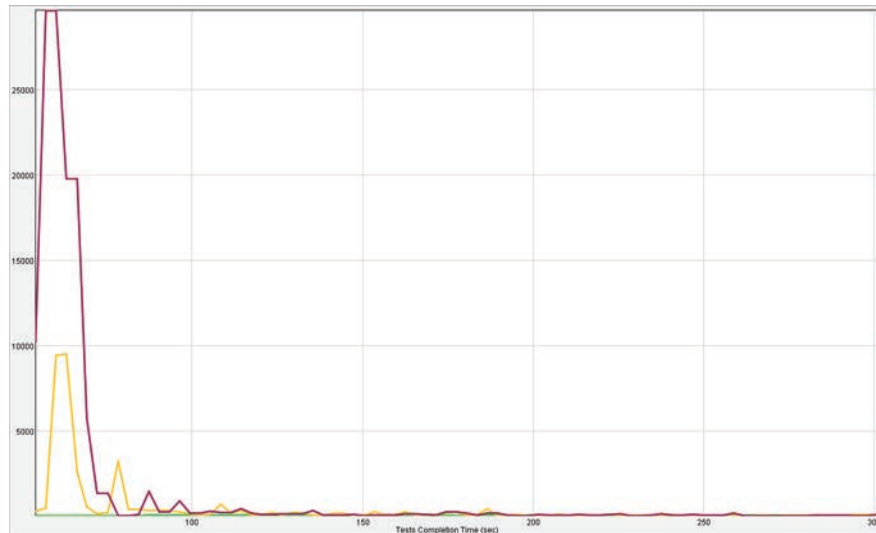
Figure 2:
Progress graphs of a
performance testing tool
executing a performance
test.

ANALYZE PERFORMANCE TEST RESULTS

A performance test results analysis starts with checking if the application performance criteria were met. If these criteria are translated into measurable metrics and performance test regression controls before the test execution as described earlier, then most performance testing tools will provide this answer automatically upon the completion of the test.

Performance engineers or testers should perform a root cause analysis if any of the metrics fail. One of the common methods in such analysis is finding a correlation between a failing performance indicator and one or more of the system or application resource monitors. For example, the following graph demonstrates a correlation between a spike in the AUT response time (yellow line) and the amount of time the AUT threads spent in a blocked state (dark red line). Visual correlation between these lines on the graph suggests thread synchronization issues as a potential cause of the spike in the AUT response time.

Figure 3:
Correlation between
execution time (yellow)
and time spent in blocked
state (dark red).



Adequate monitoring of system and application resources is important for a successful root cause analysis of performance issues. At the same time, it's important to keep in mind that monitors consume system and application resources as well, so using too many monitors or calling them too often can, in some cases, negatively affect the performance of the AUT and distort the performance picture.

AUTOMATE PERFORMANCE TEST EXECUTION

Once a performance testing infrastructure has been set up and performance projects have been created, the next step for a software development organization that follows [CI/CD practices](#) is to integrate performance tests into the CI/CD pipeline. This integration requires the following steps.

- 1. Reduce performance test results to a pass/fail answer.** Performance tests tend to generate a lot of data upon completion. These results typically include statistical tables, multiple graphs, error details, log outputs, and so forth. For an automated performance test, the plethora of this data needs to be reduced to a single pass/fail answer, which is subsequently used to promote or fail an automated build. Teams can use the QoS metrics mechanism described earlier for this purpose.
- 2. Configure automated test execution.** A performance test should fit the automated build process in which it's used. For example, a short and resource-light baseline or smoke performance test is a better fit for frequently run CI build processes. On the other hand, longer-running and more resource-intensive automated performance tests should be executed less frequently and as separate processes. See [When Should Teams Apply Performance Testing?](#) for details.
- 3. Replicate real world behavior.** Service virtualization analyzes the application's actual calls to live services to accurately model requests and responses in the virtual version. Performance engineers and testers can also apply a service definition for better correlation when the required dependent services are unavailable or difficult to connect to in order to replicate realistic interactions with these services.

- 4. Set up historical performance data analysis.** Once development teams integrate performance tests into a CI/CD pipeline, the tests will start generating a lot of results data. Making the most of this data requires a special tool that accumulates historical test results, presents them in a way that gives a high-level view of the application performance, and helps analyze historical performance trends, such as creeping loss in performance or jumps that QoS metrics didn't catch because they were below the metric sensitivity threshold.

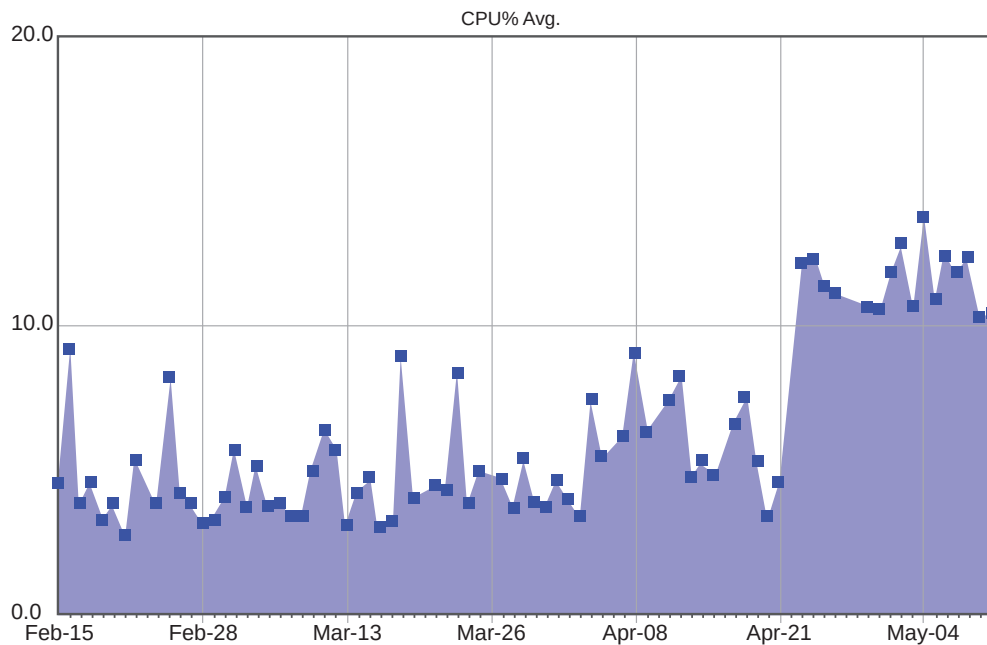


Figure 4:
Key performance
indicator (KPI) history
graph shows a date of a
systemic performance
regression.

WHEN SHOULD TEAMS APPLY PERFORMANCE TESTING?

Modern software engineering practices require building performance into the application from the start rather than an afterthought in the software development life cycle (SDLC). Discovering and fixing performance issues early avoids costly redesigns at later stages of the SDLC. It's also important to automate performance tests and run them often and continuously to support Agile development practices that require frequent release cycles.

The requirements discussed in this whitepaper extend the shift-left testing principles into the area of performance testing and referred to as shift-left performance testing (SLPT).

Except for simple applications, it's not always possible or practical to run all types of performance tests as part of the CI build process that may be triggered multiple times a day. A regular testing approach where the frequency of performance test execution depends on how much time and resources are required for the test is more practical. This means that some performance tests should be run as part of the CI build process, while more time-consuming and resource-intensive tests should be run on a regular, but less frequent basis. The table below shows some examples.

Test Frequency	Test Type
Every CI build	Smoke, unit performance
Daily/nightly	Load
Once a week	Endurance, stress

A shift-left performance testing process implemented in such a manner provides a balance between the speed with which performance regressions are caught and the number of resources required to run such processes.

ACCELERATING PERFORMANCE TESTING WITH SERVICE VIRTUALIZATION

Service virtualization simulates services, which might be APIs, systems owned by other teams, or systems under development that must be available for testing but are often tied up with production use. To simplify the process of creating the virtual assets that power service virtualization, tools such as [Parasoft Virtualize](#) create proxies.

Proxies are listeners that stand between the application and the backend systems. They monitor communication between the application and its dependencies (which are lacking any visibility), so the tool can automatically analyze the traffic files to understand patterns, link requests to responses, interpret schemas, abstract data, and so on. Applying a service definition in the virtualized environment enables the virtual asset to be created more quickly and accurately.

With this automation, it's fast and easy to build and configure a virtual service to support the needs of most test cases. It's easy to deploy these virtual services within the test environment manager with the ability to switch between real and virtual services.

Service virtualization reduces the complexity and cost of establishing a performance testbed for applications. In addition to virtualizing dependencies, here are more benefits to tools like Parasoft Virtualize.

- » **Realistic simulations.** Build environments behave—and fail—like the real thing. Teams can simulate services, model test data, and stabilize dependencies by capturing live system behavior in a virtualized service environment.

- » **Test environment management.** Manage the virtual assets associated with service virtualization so users can control every aspect of their test environment from configuring different response types and behavior to controlling performance characteristics.
- » **Stable test environment.** The proxies that sit in between the application and its dependencies can dynamically switch into failover mode if the real system ever goes offline. In failover mode, all service traffic is routed to a virtual service to keep your environment stable.
- » **Test data management.** To build virtual services on the fly with meaningful business logic and test data is a critical aspect of service virtualization and producing realistic test scenarios. This is possible through the automatic generation of data models from interactions already happening in the production system. Tools, such as Virtualize, can automatically infer information about the data to make it easier for nontechnical users to get the test data they need and easily create their own scenarios.
- » **Reporting and analytics.** Reporting is about more than test results, especially when it comes to service virtualization. Information on the health of the test environments and associated services is critical. In addition, it's important to demonstrate the ROI of virtualization with improved results and to reduce complexity and overhead.

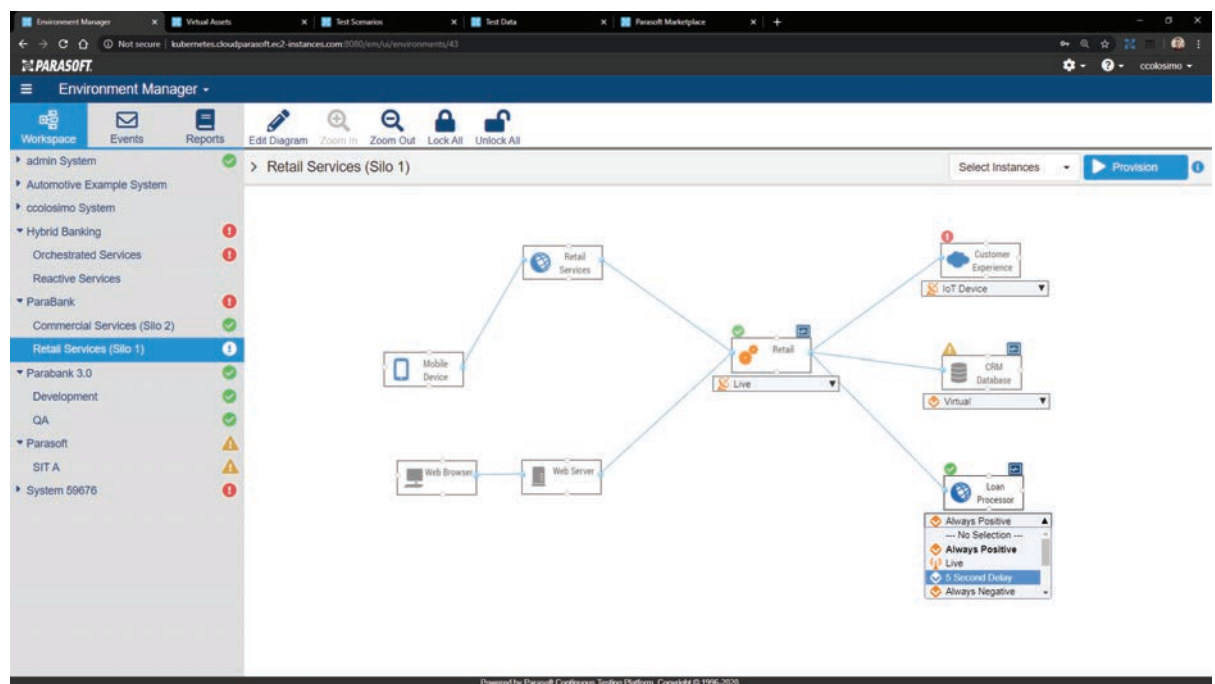


Figure 5:
The Parasoft Test Environment Manager provides a web-based interface where teams can configure and deploy services for a performance test bed.

SUMMARY

Performance testing is an essential part of quality assurance for software server applications and systems. The main goal of performance testing is to provide an answer to the question of whether the application or system can handle a projected workload with a required quality of service.

As a part of the CI/CD pipeline, automated performance testing ensures that changes to application code or system configuration that cause performance regressions are caught early and reliably. In addition, teams can use performance testing as an exploratory method to evaluate server application scalability and help make software or system design and architecture decisions.

Service virtualization simplifies and accelerates performance testing by reducing the complexity and overhead associated with traditional realworld performance test environments. Leveraging new automation techniques means fewer operating expenses and a reduction in lab costs. Virtualization also enables development teams to test earlier and more thoroughly.

TAKE THE NEXT STEP

[Request a demo](#) to see how automated and continuous testing can save your team time creating and maintaining test suites and provide realistic user scenarios to ensure applications perform as expected under diverse operating conditions.

ABOUT PARASOFT

[Parasoft](#) helps organizations continuously deliver quality software with its market-proven, integrated suite of automated software testing tools. Supporting the embedded, enterprise, and IoT markets, Parasoft's technologies reduce the time, effort, and cost of delivering secure, reliable, and compliant software by integrating everything from deep code analysis and unit testing to web UI and API testing, plus service virtualization and complete code coverage, into the delivery pipeline. Bringing all this together, Parasoft's award-winning reporting and analytics dashboard delivers a centralized view of quality enabling organizations to deliver with confidence and succeed in today's most strategic ecosystems and development initiatives—security, safety-critical, Agile, DevOps, and continuous testing.