TestRail

Best of TestRail Quality Hub

Performance Testing

TestRail

Software testing encompasses every aspect of an application, including the user interface, functional testing, load testing, security, and usability. But even if an application passes all of these tests successfully, users will be fully satistifed only if the application also delivers the necessary performance. To assist you in your performance testing, this ebook presents the best recent blog articles on some of the most challenging aspects of performance testing from the Gurock Quality Hub.

Contents

Testing All 5 Aspects of a System's Performance	3
Shifting Left: Performance Testing at the Unit Testing Level	10
Performance Testing in an Ephemeral Environment	16
Performance Testing in Edge Computing	21
Performance Testing Asynchronous Applications	27
Performance Testing: Adding the Service Registry and Service Mesh into the Mix	33
How the Service Mesh Fits with Performance Testing	38

Performance Testing

Testing All 5 Aspects of a System's Performance

Bob Reselman, Industry Analyst

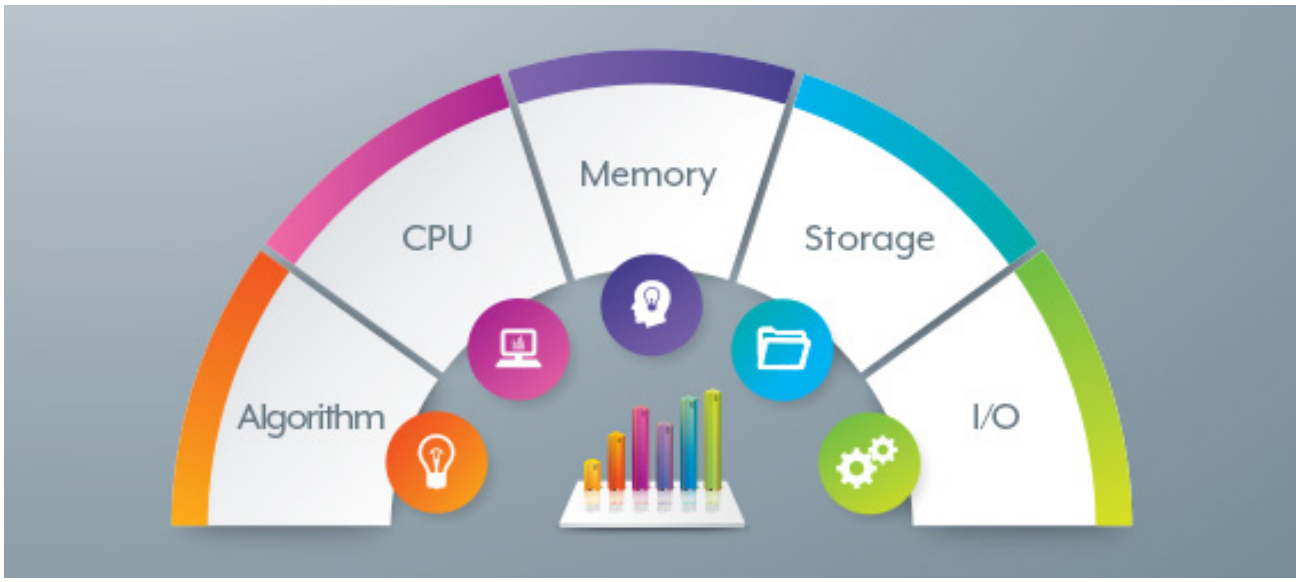
Good performance is a must for any application, whether it's in your cell phone, in a desktop computer or out on the cloud. Thus, before any application can make its way into production, it must be tested to make sure that it meets—or, hopefully, exceeds—the required performance standards.

This is easier said than done because there is no single metric you can use to determine the quality of an application's performance. Rather, performance testing requires that you measure a variety of metrics that relate to the different aspects of an application's execution.

Understanding the different aspects of application performance is critical when planning tests to determine an application's quality. Here, let's delve into these different aspects of computing performance and how to incorporate them into your test planning.

The Different Aspects of Computing Performance

Computing can be divided into five operational aspects: algorithm, CPU, memory, storage and input/output (I/O).



The table on the next page describes how these aspects relate to application performance.

Aspect	Description	Relevant Testing Metrics
Algorithm	The calculations, rules and high-level logic instructions defined in a program to be executed in a computing environment	Rule(s) assertion pass/fail, UI test pass/fail, database query execution speed, programming statement execution speed
CPU	The central processing unit is circuitry within a computing device that carries out programming instructions. CPUs execute low-level arithmetic, logic, control and input/output operations specified by the programming instruction	Clock speed, MIPS (millions of instructions per second), instruction set latency period, CPU utilization
Memory	The computer hardware that stores information for immediate use in a computing environment, typically referred to as Random Access Memory (RAM)	Memory speed per access step size, memory speed per block size
Storage	Computer hardware that stores data for persistent use. The storage hardware can be internal to a given computing device or in a remote facility on a dedicated storage device	Disk capacity utilization, disk read/write rate
I/O	Describes the transfer of data in and among computing devices. Internal I/O describes the transfer of data to the CPU, memory, internal disk storage and to interface to external networks. External I/O describes data transfer to and from different points on a network	Network path latency, available peak bandwidth, loss rate, network jitter profile, packet reorder probability

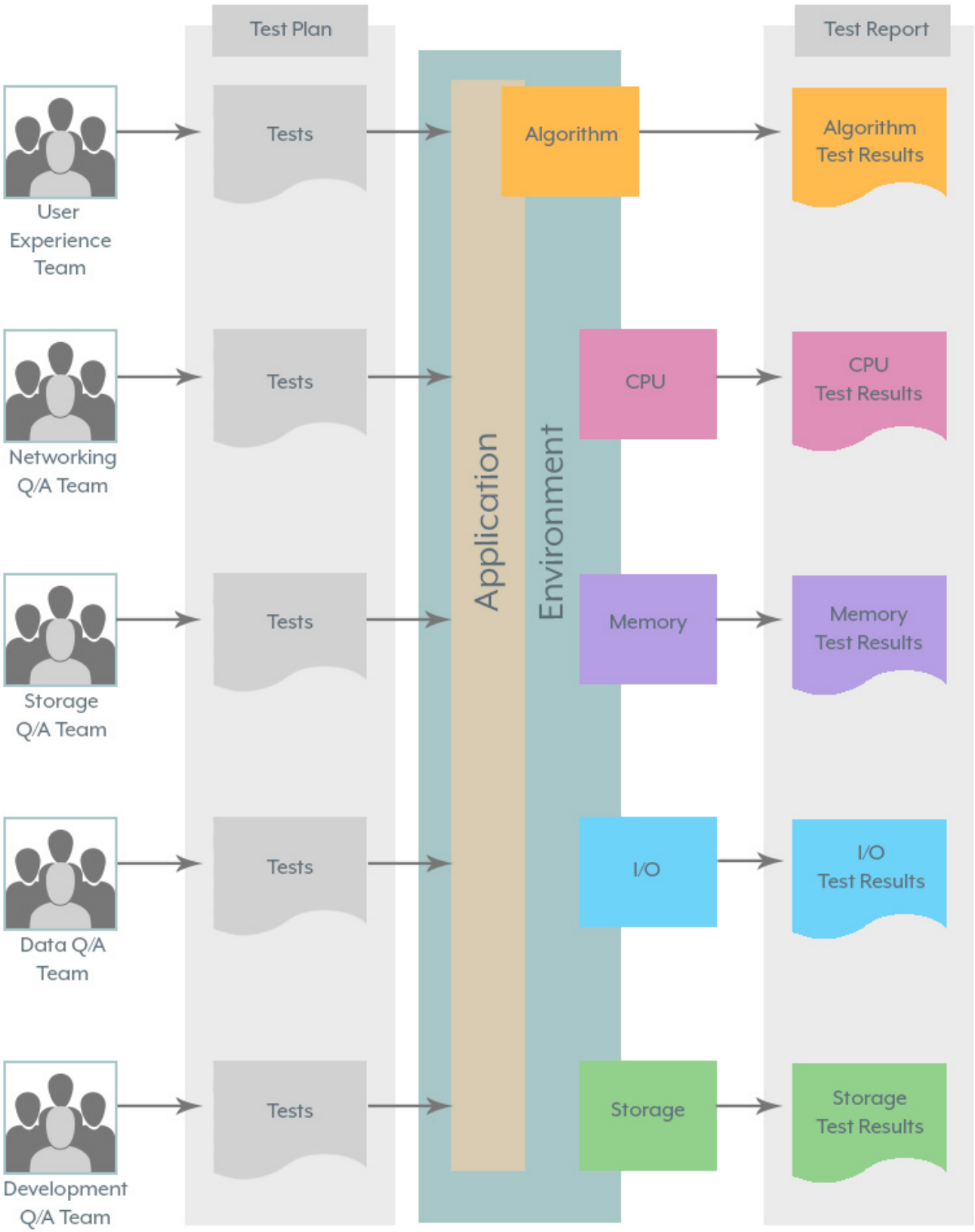
Measuring the performance of an application in terms of each of these aspects gives you the depth of insight required to determine if your application is ready for production use. How and when you'll measure these aspects will vary according to the priorities you set in your test plan.

Establishing Priorities in a Test Plan

No single testing session can cover all aspects of application performance, and it's rare for a single team to have the expertise required to test all aspects. Typically, different teams in the quality assurance landscape will focus on a particular aspect of performance testing. The testing activities of each team are organized according to a general test plan, with each part of the test plan defining a particular set of priorities to be examined.

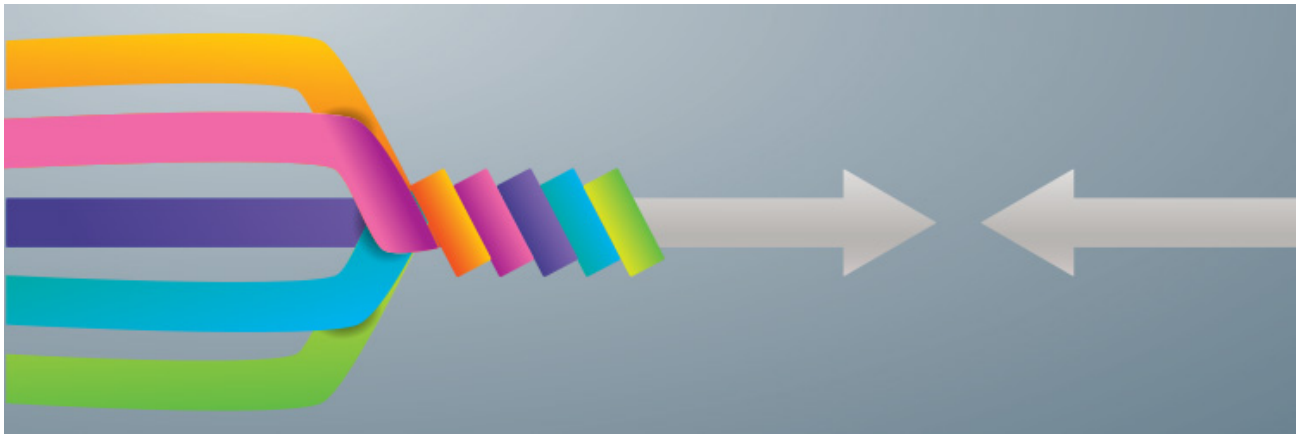
For example, the networking QA team will design and execute networking tests. The customer experience QA team will implement UI functional testing. The data QA team will performance test data access and query execution. And additional teams with expertise in other aspects will participate according to the need at hand.

The important thing to understand is that acceptable performance is determined in terms of all aspects of your application's operation, and different teams will have the expertise and test methodologies required to make the determination about a given aspect of performance quality. Therefore, when creating a test plan, you will do well to distribute work according to a team's expertise. The result of the testing done by all the teams gets aggregated into a final test report, which influences the decision to release the software to production, as shown below:



Many teams are required in order to test the performance quality of an application

The Importance of Environment Consistency



Comprehensive testing requires that all aspects of application performance be tested, measured and evaluated. Of equal importance is that the environment in which the application is tested be identical to the production environment into which the application will be deployed. The code in the testing environment probably will be the same code that will be deployed to production, but the code might vary between a test environment and production due to different configuration settings and environment variable values.

Tests that depend on the physical environment are another matter. Fully emulating the production environment in a testing scenario can be an expensive undertaking. Thus, requiring full emulation depends on the aspect of testing in play. When testing for algorithm, in terms of pass/fail of a particular rule or UI response, the speed of execution does not really matter—for example, testing login and authentication works according to credentials submitted. However, when testing that login happens within the time advertised in the application's service level agreement, environment consistency between testing and production environments becomes critical. It's a matter of testing apples to apples.

One of the benefits of automation, virtualization and cloud computing is that production-level test environments can be created on an as-needed basis, in an ephemeral manner. In other words, if your production environment is an industrial-strength, AWS m5.24xlarge environment (96 CPU, 384 GB memory, 10,000 Mbps network bandwidth), you can create an identical environment for testing, keeping it up and running only for

the setup and duration of the test. Once the test is over, you destroy the environment, thus limiting your operational costs. Ephemeral provisioning not only provides the environmental consistency required to conduct accurate testing over all five aspects of performance testing, but also allows you to test in a cost-effective manner.

Conclusion

No single test provides an accurate description of an application's performance, any more than a single touch of an elephant describes the entire animal. A comprehensive test plan will measure system performance in terms of all five aspects. Conducting a thorough examination of an application by evaluating the metrics associated with the five aspects of application performance is critical for ensuring that your code is ready to meet the demands of the users it's intended to serve.

Performance Testing

Shifting Left: Performance Testing at the Unit Testing Level

Bob Reselman, Industry Analyst

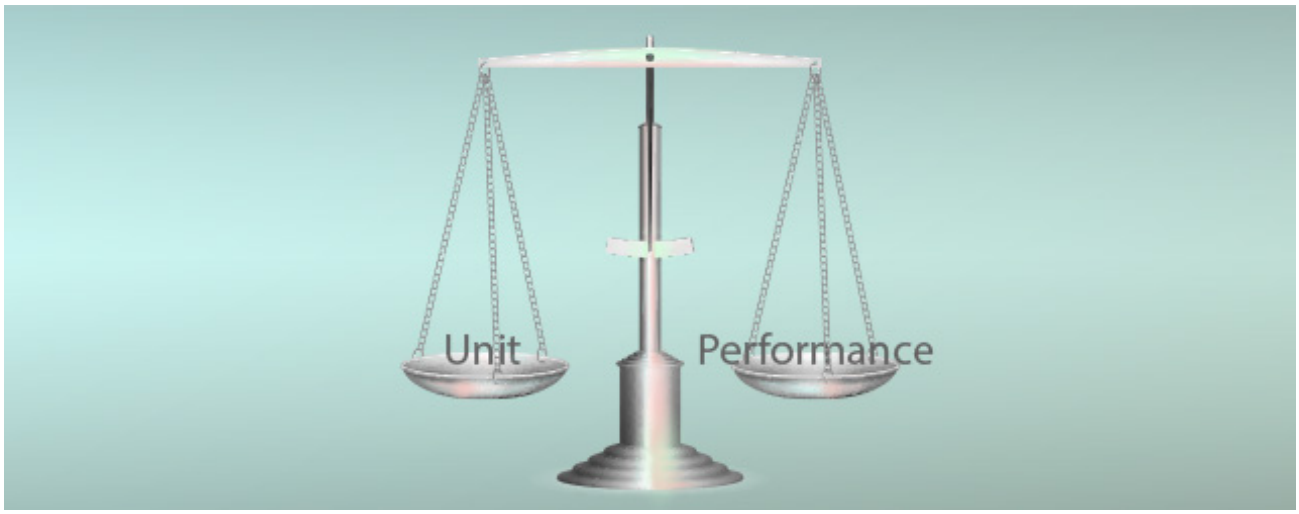
Unit testing is important. Performance testing is important. You'll get little argument from the people with boots on the ground, actually doing software development, that these tests matter a lot.

Unit testing is widely practiced in development circles, particularly by programmers who have embraced test-driven development (TDD). Unit testing, by definition, is conducted as the code is being written. Now a similar trend is emerging with regard to performance testing—to move it as close as possible to when the code is being written, into the hands of the developer. This is known as Shift Left movement. However, there's a problem.

As much as it's a nice idea to have developers conduct performance tests on their code as immediately as they do with unit testing, real-world performance testing is not a simple matter of "write it and run it." Much more is involved, particularly when unit and performance tests are part of a continuous integration and delivery process. Unless proper testing planning is in place, shifting performance testing left will not only hinder the work of the developer, but also produce test results that are of questionable reliability. The devil really is in the details.

The first step is to understand the difference between unit testing and performance testing.

Unit Testing vs. Performance Testing



The purpose of a unit test is to ensure that a unit of code works as expected. The typical unit of code is a function. A unit test submits data to a function and verifies the

accuracy of the result of that function. Unit testing is conducted using a tool such as JUnit (Java), unittest (Python), Mocha (NodeJS), PHPUnit or GoogleTest (C++). There are many others. Using a unit testing tool allows the tests to run automatically within a CI/CD process, under a test management system such as TestRail.

Performance testing, on the other hand, is the process of determining how fast a piece of software executes. Some performance tests are system-wide. Some exercise a part of the system. Some performance tests can be quite granular, to the component or even the function level, as in the case of a microservice.

The types of performance tests vary. Some performance tests focus on database efficiency. They'll execute a set of predefined SQL queries against a database of interest, using a tool such as JMeter. JMeter runs the queries and measures the time it take each query to run.

JMeter also can be used to measure the performance of different URLs in an API. A test engineer configures JMeter to make HTTP calls against the URLs of interest. JMeter then executes an HTTP request against the endpoint and measures the response time. This process is very similar to performance testing a database.

Web pages can be subject to performance testing. GUI tests can be implemented using a tool such as JMeter or Selenium to exercise various web pages according to a pre-recorded execution script. Timestamps are registered as the script runs, and these timestamps are then used to measure the execution timespan.

Networks are also subject to performance tests. It's becoming more common to run jitter tests that measure the variation in latency over a network as companies such as Facebook, Slack and Atlassian build more video conferencing capabilities into their products.

The important thing to understand is that unit testing is about ensuring a unit of code produces the logical results as expected. Performance testing is about ensuring the code executes in the time expected. The difference may seem obvious, but there are definite implications when performance testing shifts left, moving it closer to the beginning of the software development lifecycle and near, if not into, the hands of the developer.

There is a fundamental dichotomy in play. Unit tests are intended to run fast so as not to slow down the work of the developer. A developer might write and run dozens of unit tests a day. One slow unit test is a productivity nightmare. Performance tests can require a good amount of time to set up, run and measure accurately. If initializing large datasets or provisioning multiple virtual machine instances is required, a performance test can take minutes or even hours to set up.

The rule of thumb is that unit tests need speed and performance tests need time. Therefore, unless they're planned properly, one can get in the way of the other and impede the effectiveness and efficiency of the entire testing process in the CI/CD pipeline.

The Need for Speed vs. the Need for Time

As mentioned above, unit tests are intended to be run fast. A unit test that runs more than a few seconds is a rarity. Performance tests can take time to set up and run, particularly when the test executes a number of tasks during a testing session—for example, performance testing a large number of pages on a website.

The conflict between a need for speed and the need for time is a definite challenge as testers try to move performance testing closer to the unit test paradigm. The further testing shifts left toward the beginning of the test cycle, the faster the testing needs to execute. A long-running performance test will actually create a performance bottleneck in the test process.

Thus, as you consider shifting performance testing left, discretion is the better part of valor. Not every performance test is well suited for execution early in the testing process. The need for speed trumps the need for time. Short-running performance tests, such as those measuring the time it takes for a single query to run or a series of nested forEach loops to execute, are appropriate for shifting left, provided they can run in under a second. Performance tests that take longer to run need to run later in the test plan.

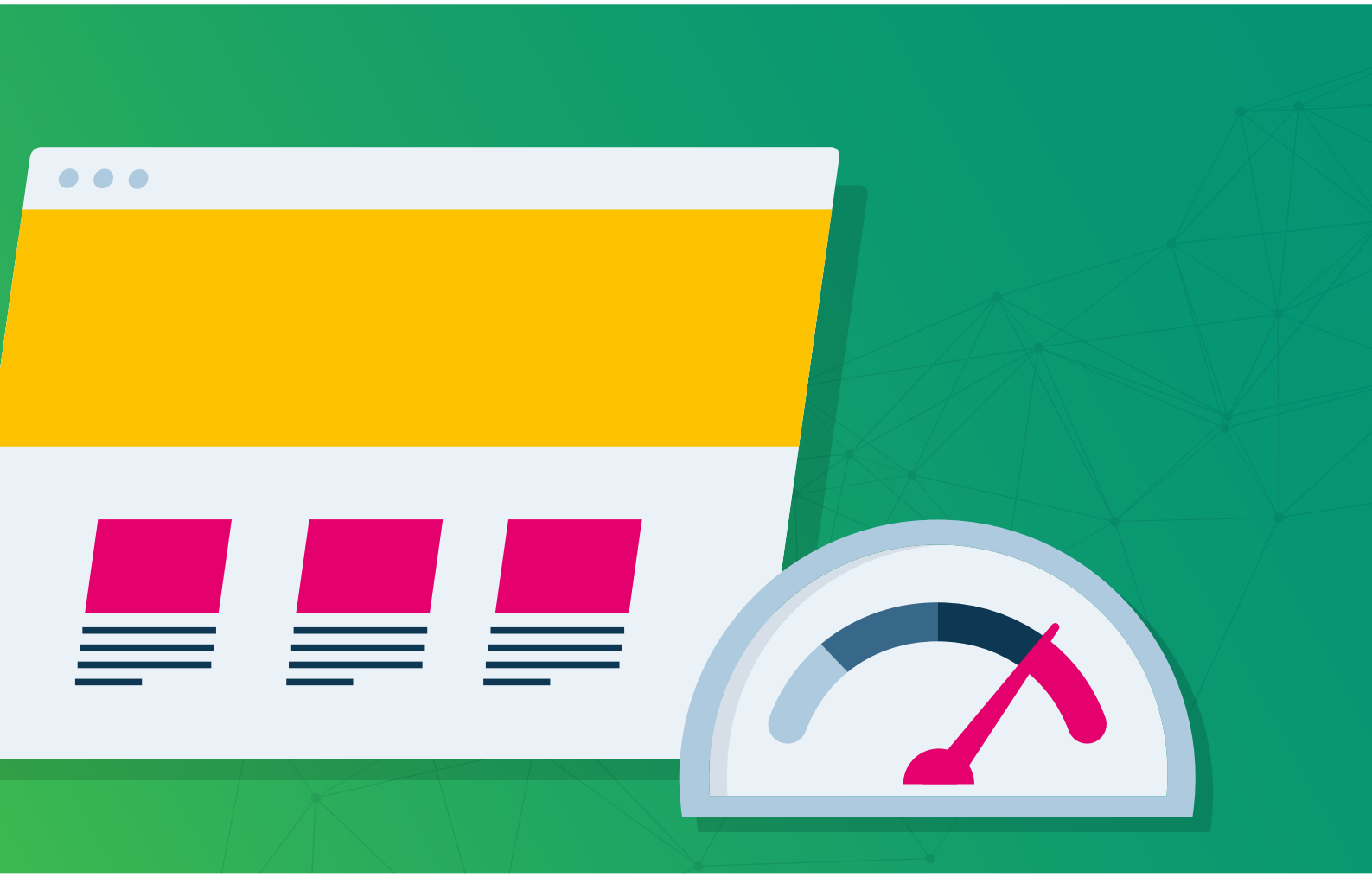
Performance Testing Is All About Apples to Apples

When considering making some aspects of performance testing the responsibility of the developer, another thing that needs to be taken into account is to make sure the physical test environments in which the performance tests run are consistent. It's an apples-to-apples thing. A performance test running on a developer laptop that has a 4 core CPU is going to behave much differently than the same test running on a VM equipped with 32 CPUs. Physical environment does matter. Thus, in order for performance testing to be accurate and reliable, the physical test environment must be real-world, beyond the typical physical configuration of a developer's workstation.

Making part of performance testing the responsibility of developers requires developers to change the way they work. The CI/CD process also needs to be altered somewhat. Again, the test environment needs to be consistent and reliable.

One way to ensure consistency is to have developers execute performance tests by moving code from their local machines over to a standardized performance testing environment before the code is committed to a repository. Modern development shops configure their CI/CD pipeline so that code enters an automated test and escalation process once it's committed. It's up to the developer to decide when the code is performant enough to commit to the repo. If the code performs to expectation, the developer makes the commit.

Testing code in a dedicated test environment before committing is a different way of working for many developers, but it's necessary when shifting part of performance testing left.



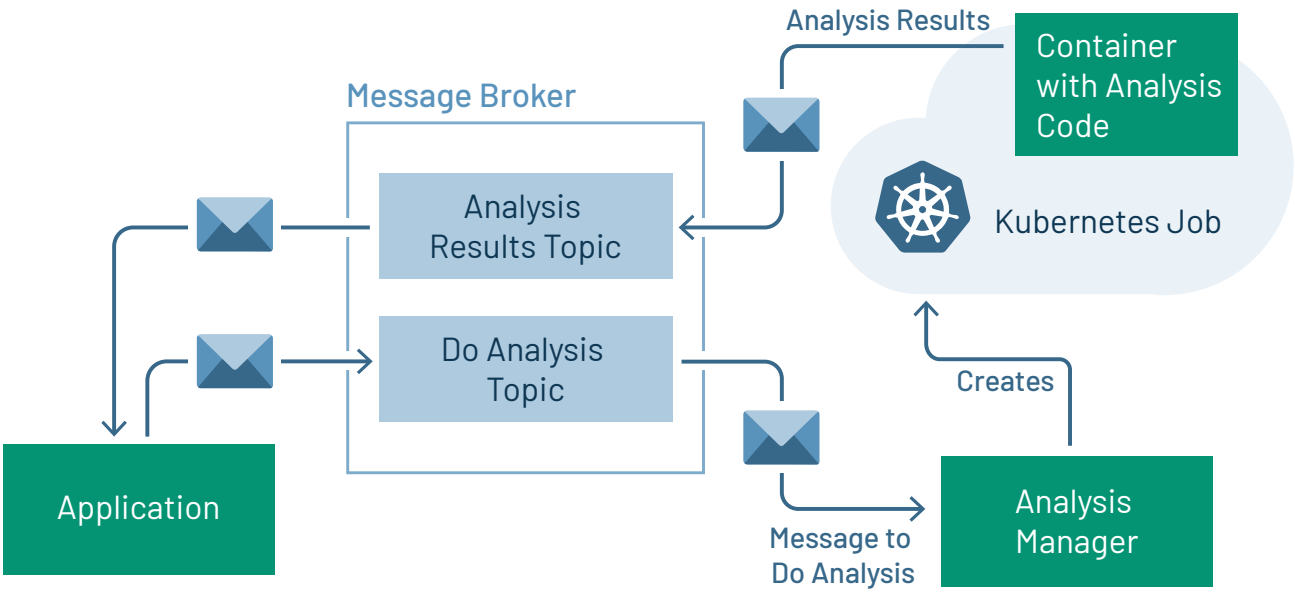
Performance Testing

Performance Testing in an Ephemeral Environment

Bob Reselman, Industry Analyst

A while back I had an interesting problem to solve. I was involved with refactoring a cloud-based application that did intensive data analysis on very large data sets. The entire application —business logic, analysis algorithm and database — lived in a single virtual machine. Every time the analysis algorithm ran, it maxed out CPU capacity and brought the application to a grinding halt. Performance testing the application was easy: We spun up the app, invoked the analysis algorithm and watched performance degrade to failure. The good news is that we had nowhere to go but up.

The way I solved the problem was to use ephemeral computing. I implemented a messaging architecture to decouple analysis from general operation. When analysis is needed, the app sends a message to a topic. An independent analysis manager takes the message off the topic and responds by spinning up a Docker container within a Kubernetes job. The Docker container running in the Kubernetes job has the analysis code. The Kubernetes node has 32 cores, which allows the code to run fast. Then, when the job is done, the Kubernetes job is destroyed, as shown below. We end up paying only for the considerable computing resources we need, when we need them.



Using message-driven architecture in combination with automation to provision environments ephemerally has become a typical pattern, so I wasn't really breaking any new ground. But I did encounter a whole new set of problems in terms of testing — particularly performance testing.

Implementing a Performance Test

The way we ran performance tests was standard. We spun up some virtual users that interacted with the application to instigate the analysis jobs. We started small and worked our way up, from 10 virtual users to 100. We didn't go beyond that count because the service-level agreement we needed to support called for 10 jobs running simultaneously, but we went up to 100 to be on the safe side.

Defining Test Metrics

An overlooked and underused tool in testing is the developer console, available in all web browsers. For testers who are developing UI automation tests, this tool is necessary to identify DOM elements. Sometimes, building a CSS expression for difficult-to-locate elements is necessary and developer tools make this possible. By investigating logs and adjusting JavaScript, testers can investigate bugs and provide detailed descriptions of the problems they see. There are many tutorials available that help beginners learn to use the console methods that are useful to conduct a deep investigation of a website.

Taking the DevOps Approach

Given that the system we created was message-based, we couldn't just measure timespan, as is typical of an HTTP request/response interaction. Rather, we needed to trace activity around a given message. Thus, we took the DevOps approach.

I got together with the developer creating the message emission and consumption code. Together we came to an agreement that information about the message would be logged each time a message was sent or received. Each log entry had a timestamp that we would use later on. We also decided that each log entry would have a correlation ID, a unique identifier similar to a transaction ID.

The developer implementing messaging agreed to create a correlation ID and attach it to the "do analysis" message sent to the message broker. Any activity using the message logged that correlation ID in addition to the runtime information of the moment. For example, the application logged information before and after the message was sent. The Analysis Manager that picked up the message logged receipt data that included this unique correlation ID. Message receiving and forwarding was logged throughout the system as each process acted upon the message, so the correlation ID tied together all the hops the message made, from application to analysis to results.

We also met with the developer implementing the Kubernetes job to have the correlation ID included when logging the job creation and subsequent analysis activity, where applicable.

Getting the Results

Once we had support for correlation IDs in place, we wrote a script that extracted log entries from the log storage mechanism and copied them to a database for review later on. Then, we ran the performance test. Each virtual user fired off a request for analysis. That request had the correlation ID that was used throughout the interaction. We ran the tests, increasing the number of virtual users from 10 to 100.

One of the results the test revealed was that the messages were taking a long time to get to topic and onto the Analysis Manager subscriber. We checked with the developer who created the messaging code, and he reported that everything was working as expected on his end. We were perplexed.

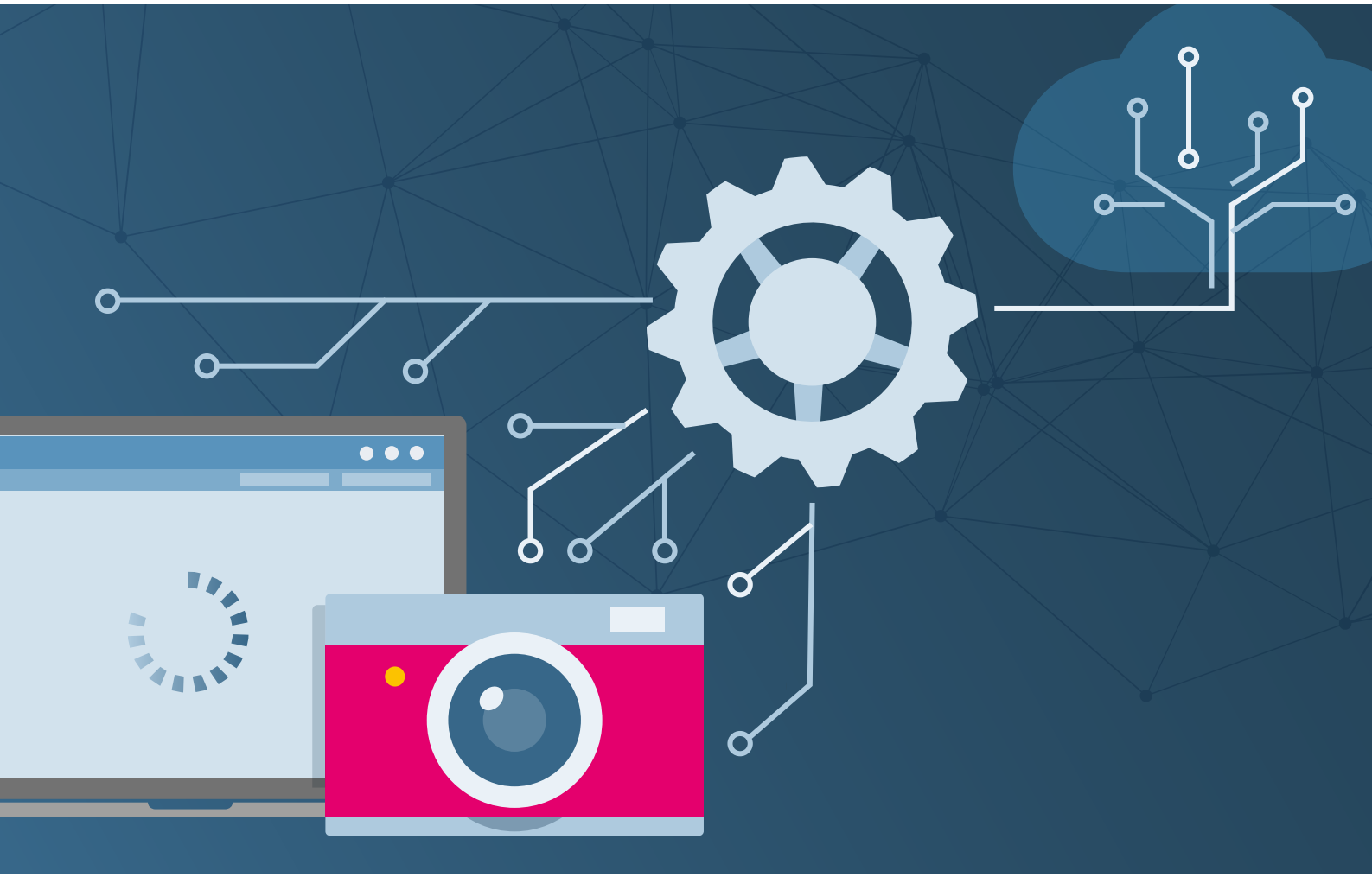
Gotcha: Everything Is Not the Same Everywhere

Our performance issue is not at all atypical in software development — things work well in the development environment but behave poorly under independent, formal testing. So we did a testing post-mortem.

When we compared notes with the development team, we uncovered an interesting fact: Developers were using one region provided by our cloud service, but we were testing in another region. So we adjusted our testing process to have the Kubernetes job run in the same region as the the one used by the developer. The result? Performance improved.

We learned a valuable lesson. When implementing performance testing on a cloud-based application, do not confine your testing activity to one region. Test among a variety of regions. No matter how much we want to believe that services such as AWS, Azure and Google Cloud have abstracted away the details of hardware from the computing landscape, there is a data center down in that stack filled with racks of computers that are doing the actual computing. Some of the data centers have state-of-the-art hardware., but others have boxes that are older. There can be a performance difference.

In order to get an accurate sense of performance, it's best to test among a variety of regions. When it comes to provisioning ephemeral environments, everything is not the same everywhere.



Performance Testing

Performance Testing in Edge Computing

Bob Reselman, Industry Analyst

Edge computing is about moving computational intelligence as close to the edge of a network as possible. Essentially, in the past, "frontline" devices gathered data passively and sent the information onto a target endpoint for processing. Under edge computing, some processing is done within the frontline device itself, and the result of the computation is moved onto other processing targets within the network.

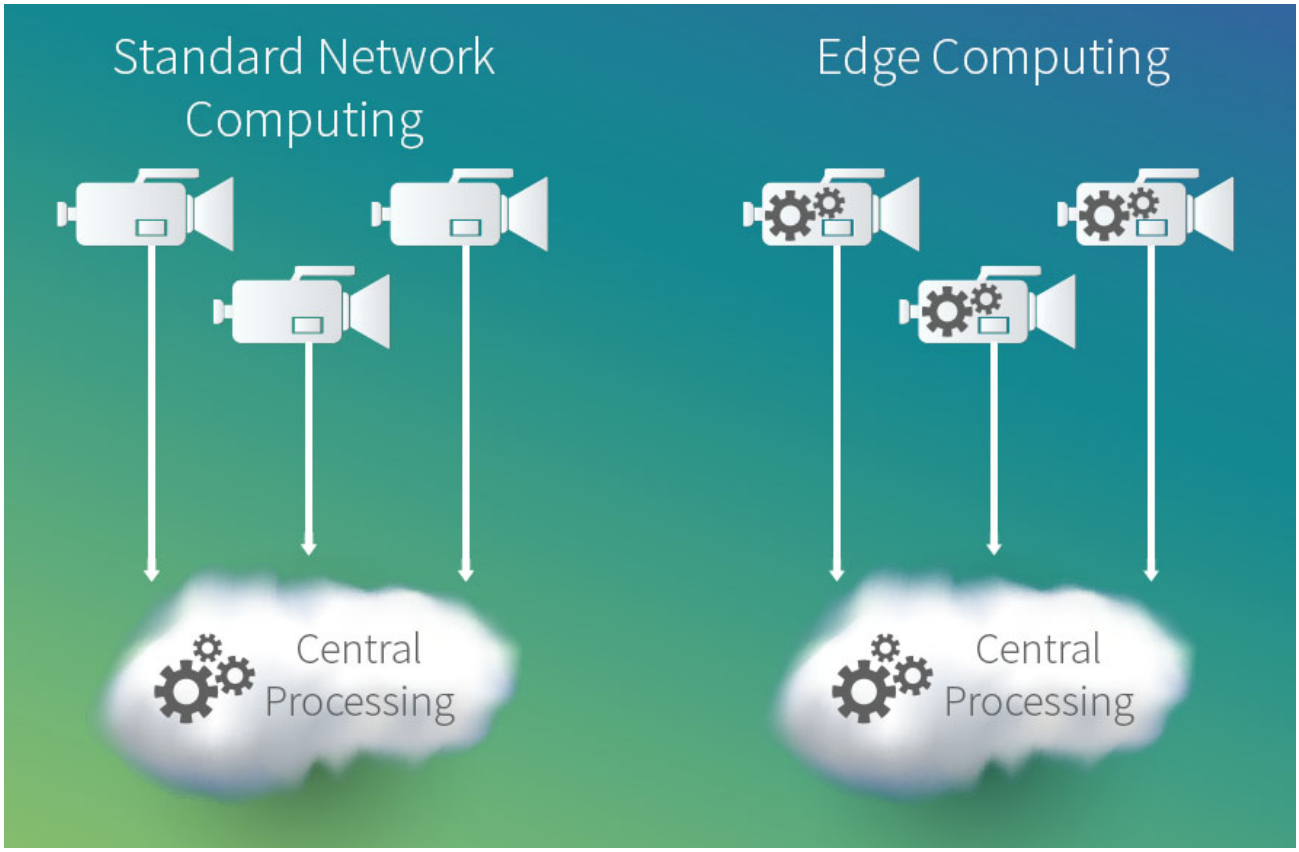
Edge computing is a transformational approach to how distributed applications works on the internet. But, as with any new technique, there are challenges. This is particularly true when it comes to performance testing within an edge computing ecosystem.

Understanding these challenges is useful, particularly now that edge computing is becoming a more prevalent part of the internet. With this new paradigm, we will need to devise new ways of approaching performance testing.

Understanding Edge Computing

One of the easiest ways to understand edge computing is to consider a video camera connected to the internet. Before edge computing came along, the video camera simply captured all visual activity occurring before its lens. Each frame of video was converted into a chunk of data that was sent over the network in a stream. The camera was on all the time, streaming data back to a server, regardless of whether there was any motion occuring. Clearly, a good deal of data having marginal use was being passed over the network, eating away at bandwidth.

In an edge computing paradigm, intelligence is put into the video camera so it can identify motion and then stream data back to the server only when motion occurs, as shown below. Such an approach is highly efficient because only valuable data gets back to server for processing.

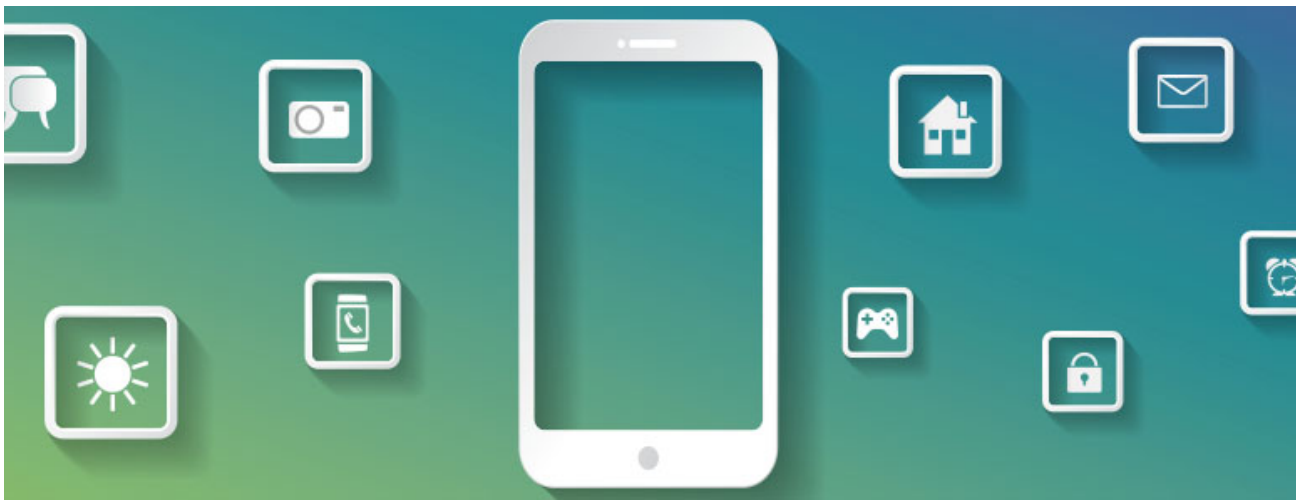


This sort of computing is now becoming the norm. All the security video cameras in the building where I live are motion-sensitive: The video camera is smart enough to detect motion and save only the video stream for the given time-span of motion. This means that the only time that data is saved is when there is activity around the building. The camera also timestamps the video when saving the stream, so we don't have to go through hours of viewing to find an event of interest. We can just browse through recordings of time stamped video segments. A process that used to take hours is now done in minutes.

Now apply the concepts behind edge computing to other types of devices connected to the internet of things. Take a fitness device, for example. A fitness tracker has the ability to monitor the heartbeat of the person wearing it. Applying the principles of edge computing, we can make it so that when the wearer's heartbeat stops, the device

automatically contacts an EMT service for medical assistance directly. There is no delay incurred due to the latency created, should a backend server be the sole point of computation intelligence. Rather, the edge device is making the call for help immediately, saving time in a life-or-death situation. Such an example is extreme, but it does provide profound insight about the power and benefit of edge computing.

Edge computing is an important technology. If nothing else, it will cut down on the amount of useless data that gets passed around the internet. But, as with any technology, in order to be viable, it must be testable. And this is where the challenges of edge computing scenarios emerge.



Performance Testing at the Edge

Computing requires both software and hardware. Over the years hardware has become so abstract that we now treat a lot of it as software — thus, infrastructure as code. As such, a good deal of our performance testing treats the underlying hardware as a virtualized layer that is consistent and predictable. For the most part, it is, provided you're dealing with the types of hardware that live in a data center. But things get difficult when we move outside the data center.

Take mobile phones, for example. Most mobile phone manufacturers provide software that emulates the physical device. Software developers use these emulators to write and test code. It's a lot easier for a developer to write code using an emulator than to have to be constantly deploying code to a physical device every time a line of code is changed.

Using emulators speeds up the process of writing and testing applications that run mobile phones. But at some point, someone, somewhere is going to need to test that software on a real mobile phone. To address this issue, companies set up mobile phone testing farms: A company will buy every mobile phone known to man and put them in a lab. The lab is configured to allow a tester to declare the physical phone that the given software under test will be installed and run on. (Some companies have automated the process, and others even provide mobile phone farms as a service.) In this case, the challenge is conducting large-scale performance testing on disparate physical devices.

The same cannot be said about IoT devices such as fitness trackers, household appliances and driverless vehicles. These devices are positioned to be an important part of the edge computing ecosystem, but presently, the emulation capabilities required to conduct adequate performance testing of software intended to run on these devices are limited — or, in some cases, nonexistent. And there is still the problem that at some point somebody is going to have to load the software onto a piece of the physical hardware in order to run the performance tests required.

Given that most modern performance testing is a continuous, automated undertaking, when it comes to performance testing in an edge computing environment, it seems as as if we've gone back to the Stone Age.

How to Move Forward

Traditionally, companies buy hardware in order to run software, so most performance testing today is primarily focused on testing software. However, in the world of edge computing, companies buy the appliance, which is a combination of both hardware and

software. In this case, performance testing is focused on the appliance. It's a different approach to testing — one that still requires a good deal of time and labor to execute.

If the history of software development has shown us anything, it's that at some point, automation in general — and test automation in particular — must prevail. But in order for automation to prevail in the edge computing ecosystem, a higher degree of standardization must be achieved among IoT devices on an industry-wide basis.

In other words, the way your tests interface with a driverless vehicle should not be that different from the way the tests interact with an internet-aware refrigerator. Yes, you might be testing different features and capabilities, but the means of integration must be similar. Think of it this way: Monitoring the cooling system of a driverless vehicle should not be that different from monitoring the cooling system of an internet-aware refrigerator.

Solving the problem of working disparate devices in a common way is nothing new. At one time the way a printer connected to a computer was completely different from the way you connected a keyboard or mouse. Now, USBs provide the common standard. The same thing needs to happen with edge computing devices.

Creating a universal standard for emulating IoT devices, as well as for physically accessing any device via automation, are the hidden challenges in edge computing when it comes to performance testing. But meeting these challenges is not insurmountable; unifying device standardization is a challenge we've met before and one we're sure to meet again. The company that meets the challenge this time may very well set the standard for performance testing in the edge computing ecosystem for a long time to come.

Performance Testing

Performance Testing Asynchronous Applications

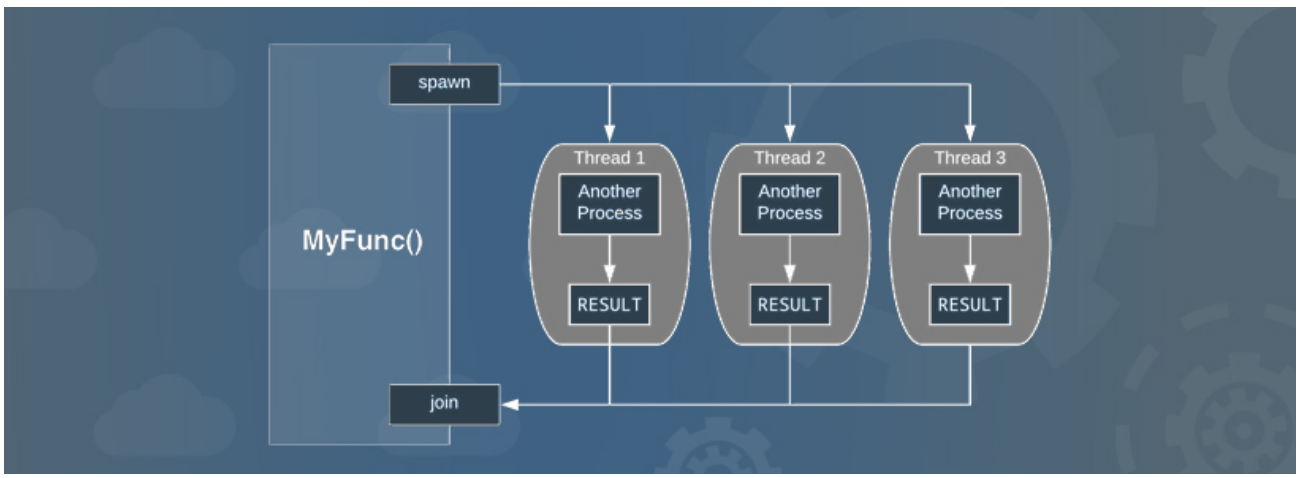
Bob Reselman, Industry Analyst

Asynchronous computing addresses a need that has been at the forefront of application development since the earliest days of computing: ensuring fast application response between systems. Whereas synchronous architectures require a caller to wait for a response from a called function before moving on with the program flow, asynchronicity allows an external function or process to work independently of the caller.

The caller makes a request to execute work, and the external function or process "gets back" to the caller with the result when the work is completed. In the meantime, the caller moves on. The benefit is that the caller is not bound to the application while work is being performed. The risk of blocking behavior is minimal.

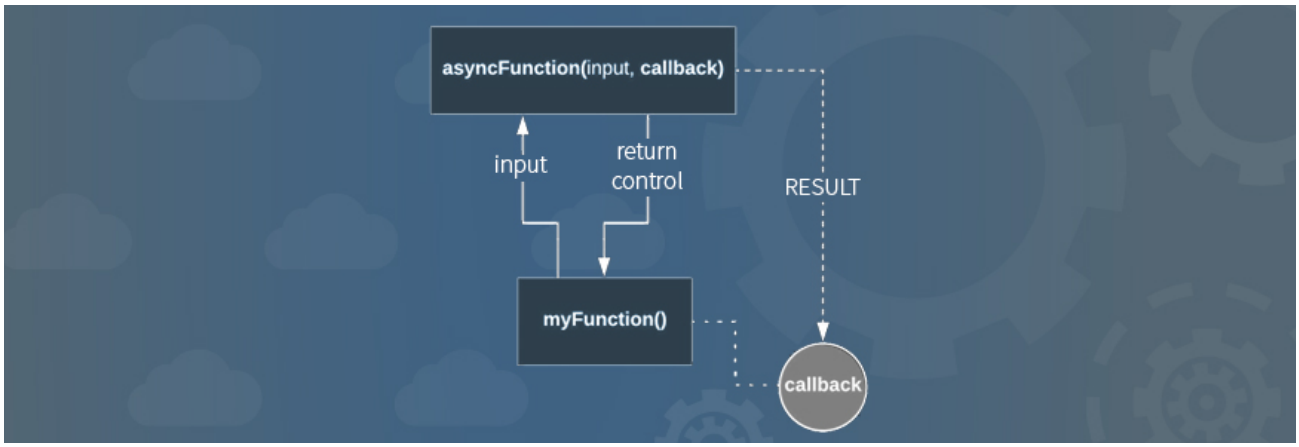
Reducing Wait Time with Multithreading

One common way to implement asynchronicity internally in an application is to use multithreading, as shown below:



In a multithreaded environment, a function spawns independent threads that perform work concurrently. One thread does not hold up the work of another. Then, when work is complete, the results are collected together and passed back to the calling function for further processing.

An event-driven programming language such as NodeJS implements asynchronicity using callback functions. In the callback model, a caller to a function passes a reference to another function — the callback function— that gets invoked when work is completed.



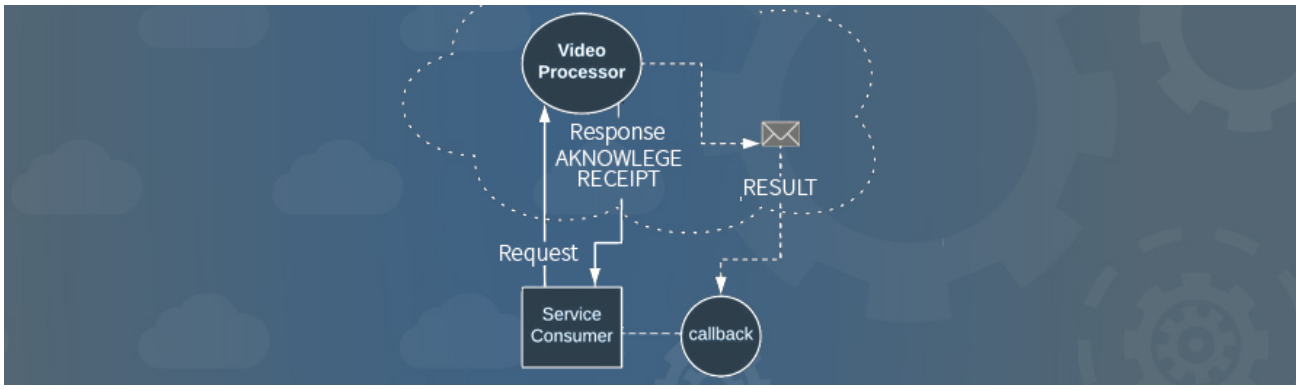
The benefit of using callbacks to implement an asynchronous relation between calling function and called function is that it allows the overall application to work at maximum efficiency. No single process is holding up another process from doing work.

The callback pattern is useful for implementing an asynchronous interaction among distributed systems, particularly web-based systems using HTTP. HTTP is based on the request-response pattern. Typically, a caller makes a request for information to a server on the internet. The server takes the request, does some work and then provides a response with the result. The interaction is synchronous. The calling system — a web page, for example — is bound to the server until the response is received.

When the response takes a few milliseconds to execute, there is little impact on the caller. But when the server takes a long time to respond, problems can arise. However, there are justifiable reasons for having a request that takes a long time to process.

For example, imagine a fictitious service that is able to determine the number of times a particular person smiles in a given YouTube video. The caller passes the URL of the YouTube video to the service. The service gets the video from YouTube, as defined by the submitted URL. Then, the process does the processing.

One video might have a one-minute runtime, and another might have a 30-minute runtime. Either runtime is acceptable to the service looking for smiles. However, a caller using the service does not want to stay connected for the duration of the 30-minute analysis. Rather, the better way to do things is to make it so the video analysis service responds quickly with an acknowledgement of receipt. Then, when work is completed, the analysis service sends the results back to the caller at a callback URL, as shown in the example below:



Asynchronous services are useful. However, they do make performance testing more difficult.

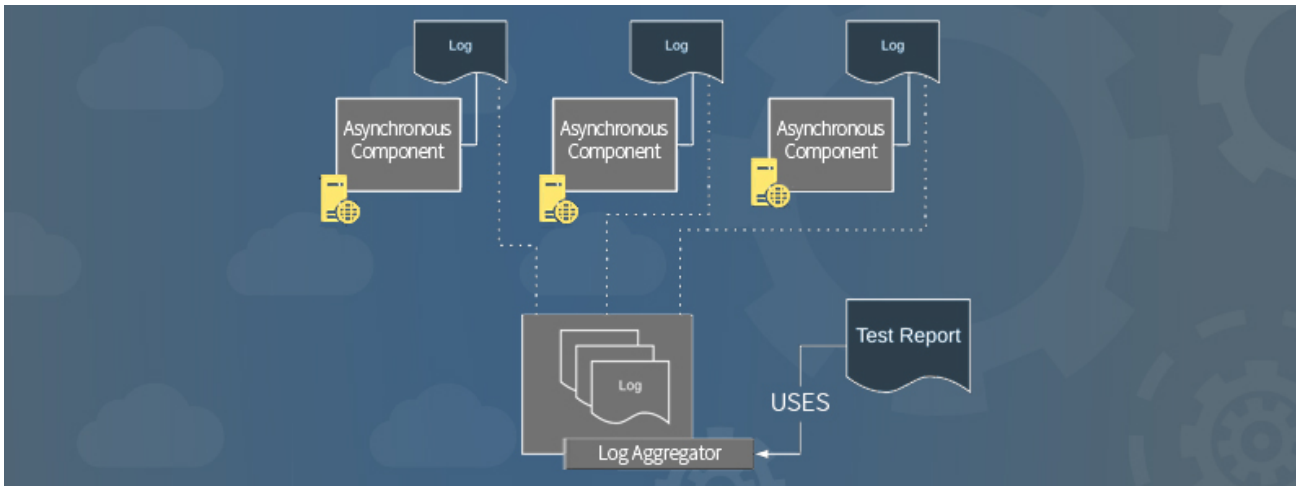
Measuring the performance of a synchronous service typically happens in a sequential manner: You define the virtual users that will execute the given test script. The test runs, requests are made and the results from responses are gathered. One step leads to another. But when it comes to asynchronous testing, one step does not lead to another. One step might start an asynchronous process that finishes well after the test is run.

The risk when testing asynchronous applications is that the process becomes independent of its caller. When the called process is on an external machine or in another web domain, its behavior becomes hidden. The performance of the called process can no longer be accurately tested within the scope of the request-response exchange. Other ways of observing performance need to be implemented.

Leveraging Event Information in Logs

One of the easiest ways to observe performance in a distributed asynchronous application is to leverage the power of logs. Logs provide the glue that binds separate systems together. Depending on the logging mechanism in force, logs can capture function timespan, memory and CPU capacity utilization, as well as network throughput, to name a few metrics.

Once a system is configured so that every component of an asynchronous application is logging information in a consistent, identifiable manner, what remains is to aggregate the log information to create a unified picture of performance activity overall, as shown below:



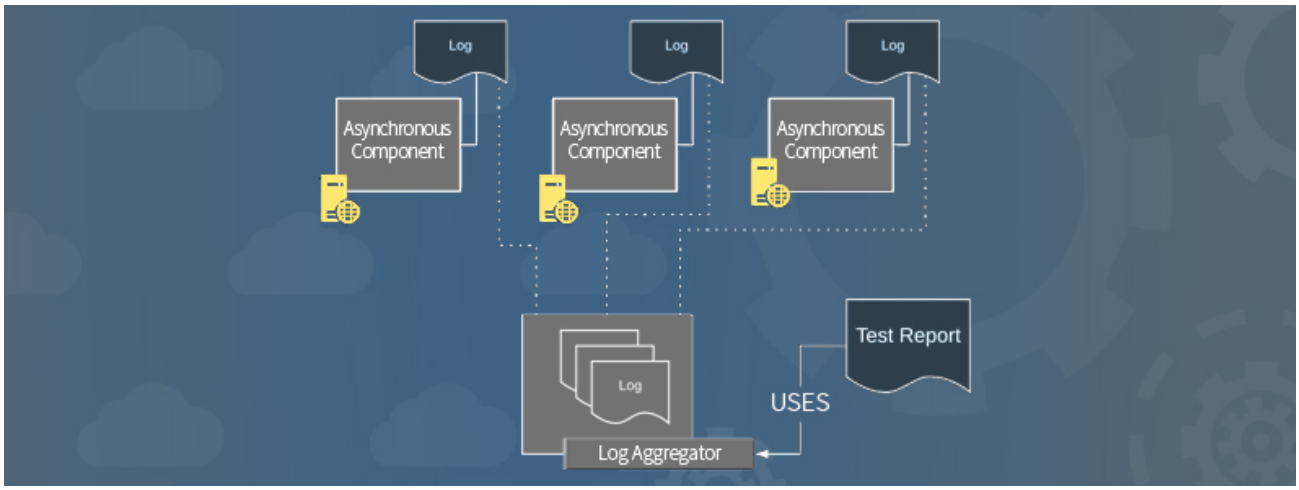
Using a unique correlation ID that gets passed among all components that are part of an asynchronous transaction provides a reliable mechanism by which to group various log entries for subsequent analysis. (See the details of working with correlation IDs here.)

The important thing to remember is that in an asynchronous application, any execution path can use a variety of processes that work independently of one another. Application flow can just hop about in a nonlinear manner. Thus, the way to tie all relevant performance metrics together is to aggregate log data from the various systems that make up the application.

Using an Application Performance Monitor

Application performance monitors (APMs) are fast becoming a standard part of most deployment environments. An APM is an agent that gets installed on a machine — either virtual or pure hardware — as part of the provisioning process. APMs observe machine behavior at a very low level. For example, most APMs will report CPU utilization, memory allocation and consumption, network I/O and disk activity. The information that an APM provides is very useful when determining application performance during a test.

Information provided by APMs is particularly useful when it comes to performance testing asynchronous applications. Much in the same way that log information from independent components can be tied together to provide a fuller understanding of performance test behavior overall, so too can information that gets emitted from an APM.



A bottleneck occurring in one system might not be readily apparent in the initial request-response that started the asynchronous interaction, and a fast request-response time can be misleading. More information is needed. APM data from all the systems used by an application will reveal actual performance shortcomings, even if poor performance might not be evident when measuring the initial request and response that started the interaction with the application.

Putting It All Together

Performance testing asynchronous applications is difficult, but it's made easier if proper test planning takes place. Testing asynchronous applications requires more than measuring the timespan between the call to an application and the response received back. Asynchronous applications work independent of caller, so the means by which to observe behavior throughout the application must be identified before any testing takes place.

A good test plan will incorporate information from machine logs and application performance monitors to construct an accurate picture of the overall performance of the application. Should log and APM information not be available, then test personnel will do well to work with DevOps personnel to make the required information available. Being able to aggregate all the relevant information from every part of a given asynchronous application is critical for ensuring accurate performance testing. A performance test is only as good as the information it consumes and the information it produces.

Performance Testing

Performance Testing: Adding the Service Registry and Service Mesh into the Mix

Bob Reselman, Industry Analyst

Ephemeral architectures in which microservices scale up automatically to meet capacity requirements are a game changer. Whereas deploying container-backed microservices and orchestrating the environment they're to run in used to be a laborious manual task, today's technologies such as Kubernetes automate much of the process of packaging and deploying applications into the cloud.

But there is a problem: service management. One of the crucial tasks that needs to happen in a microservice environment is that microservices need to find one another and interact safely. Back when the number of microservices in play was minimal, it was possible to manually configure one microservice IP address to another and declare the operational behavior between them.

But that was then and this is now. Today, a single enterprise might have thousands of microservices in force. Many of these microservices will be created and destroyed as needed — which, by the way, is the nature of ephemeral computing. Continually fiddling with configuration settings manually to maintain reliable communication between an enterprise's microservices is archaic.

Modern microservice management requirements exceed human capacity, and better solutions are needed. Enter the service registry and the service mesh.

Understanding the Service Registry

A service registry is a database that keeps track of a microservice's instances and locations. Typically, when a microservice starts up, it registers itself to the service registry. Once a microservice is registered, most times the service registry will call the microservice's health check API to make sure the microservice is running properly. Upon shutdown, the microservice removes itself from the service registry.

There are a number of open source service registry technologies available. The Apache Foundation publishes ZooKeeper. Also, Consul and Etcd are popular solutions.

The service registry pattern is pervasive. These days, most microservices are assigned IP addresses dynamically, so the service registry is the only way that one microservice can get the information it needs to communicate with another service. In fact, the service registry is so important that it serves as the foundation of the next generation of distributed computing technology: the service mesh.

Evolution into the Service Mesh

A service mesh is a technology that provides a layer of infrastructure that facilitates communication between microservices. Most service mesh projects provide service registration internally, as would be expected. But a service mesh adds functionality for declaring operational behavior such as whitelist security, failure retries and routing behavior.

For example, imagine a microservice that calls out to another microservice, such as `time.jsontest.com`, which is outside the application domain. Unless the service mesh is configured to allow calls to the external resource at `time.jsontest.com`, any service using the external resource will fail until access is granted. (Restricting access to external domains by default is a good security practice.)

The security capabilities alone are a compelling reason to use a service mesh to coordinate microservice behavior, but there's more. Most service mesh projects can publish a graph of microservice connections and dependencies, along with performance all along the way.

There are many open source service mesh technologies available. One is Istio, which was started by developers from Google, IBM and Lyft. Linkerd is a service mesh project sponsored by the Cloud Native Computing Foundation. Consul, which is mentioned above as a service registry project, has evolved into a full-fledged service mesh product. These are just a few of the many service mesh products that are appearing on the distributed cloud-based computing landscape.

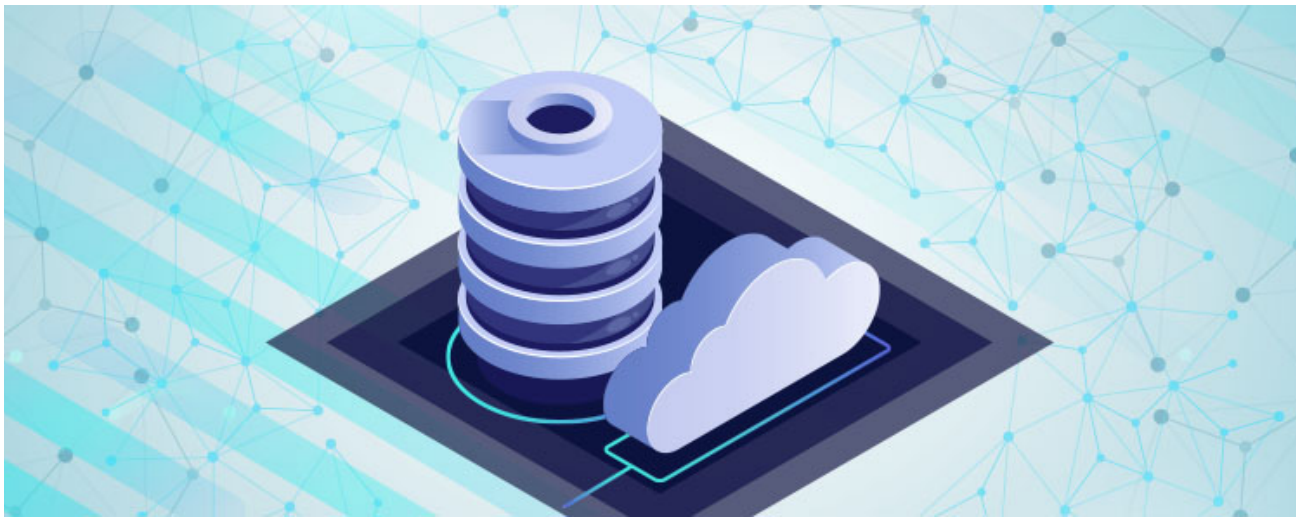
Why a Service Registry and a Service Mesh are Important to Performance Testing

As working in the cloud has become more prevalent in the world of performance testing, so too will working with service discovery technologies such as the service registry and, particularly, the service mesh. The service registry and service mesh are not low-level pieces of infrastructure; rather, they are first-class players in the enterprise's digital computing infrastructure.

In the old days, when applications where monolithic and everything lived behind a single IP address, all the performance tester needed to be concerned with was behavior at that IP address. Today, it's different. There might be thousands of IP address in play that change at a moment's notice, and there might be different operational and security policies in force at each address.

Performance testing is no longer about measuring the behavior of a single request and response at a single entry point. There will be hundreds, maybe thousands of points along the execution path to consider, so at the least, the tester is going to have to accommodate collecting performance data from throughout the ecosystem. The performance tester also is going to have to know at least enough about the workings of the service mesh in order to collaborate with DevOps engineers to set routing configurations for A/B testing and to determine environment optimization throughout the system.

Modern performance testing goes well beyond writing procedural scripts and collecting test results. Today's digital ecosystem is intelligent. As systems become more autonomous and ephemeral, having a strong grasp of technologies such as the service registry and service mesh will go from being a nice-to-have skill to one that is required for testing professionals. The modern performance tester needs to be able to work with intelligent systems to design and implement comprehensive tests that are as dynamic as the systems being tested.



Performance Testing

How the Service Mesh Fits with Performance Testing

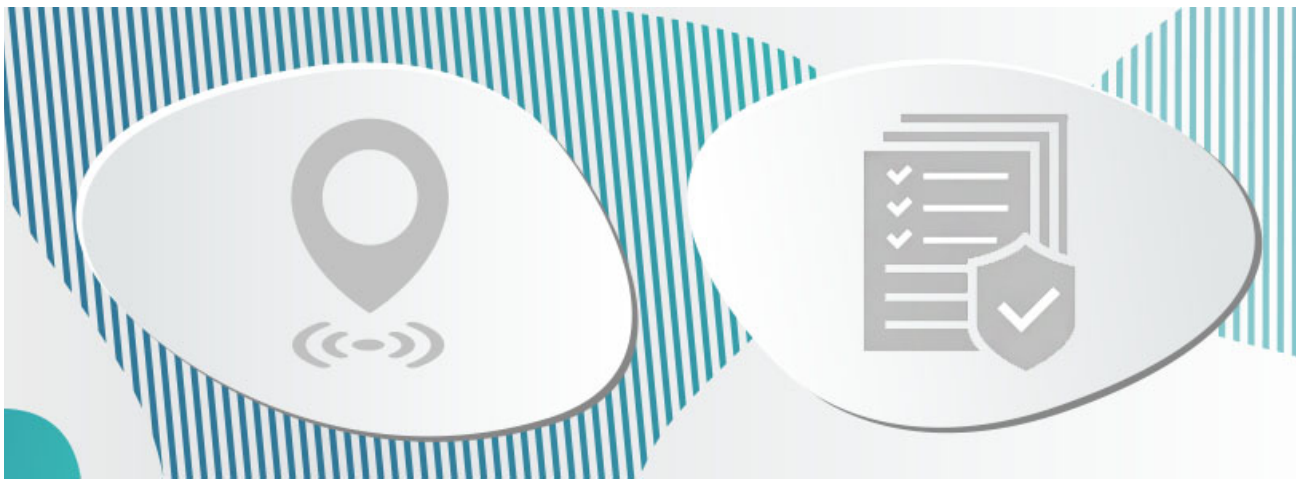
Bob Reselman, Industry Analyst

The digital infrastructure of the modern IT enterprise is a complex place. The days of one server attached to a router at a dedicated IP that's protected by a hand-configured firewall are gone.

Today, we live in a world of virtualization and ephemeral computing. Computing infrastructure expands and contracts automatically to satisfy the demands of the moment. IP addresses come and go. Security policies change by the minute. Any service can be anywhere at any time. New forms of automation are required to support an enterprise that's growing beyond the capabilities of human management. Once such technology that is appearing on the digital landscape is the service mesh.

The service mesh is expanding automation's capabilities in terms of environment discovery, operation and maintenance. Not only is the service mesh affecting how services get deployed into a company's digital environment, but the technology is also going to play a larger role in system reliability and performance, so those concerned with performance and reliability testing are going to need to have an operational grasp of how the service mesh works, particularly when it comes to routing and retries.

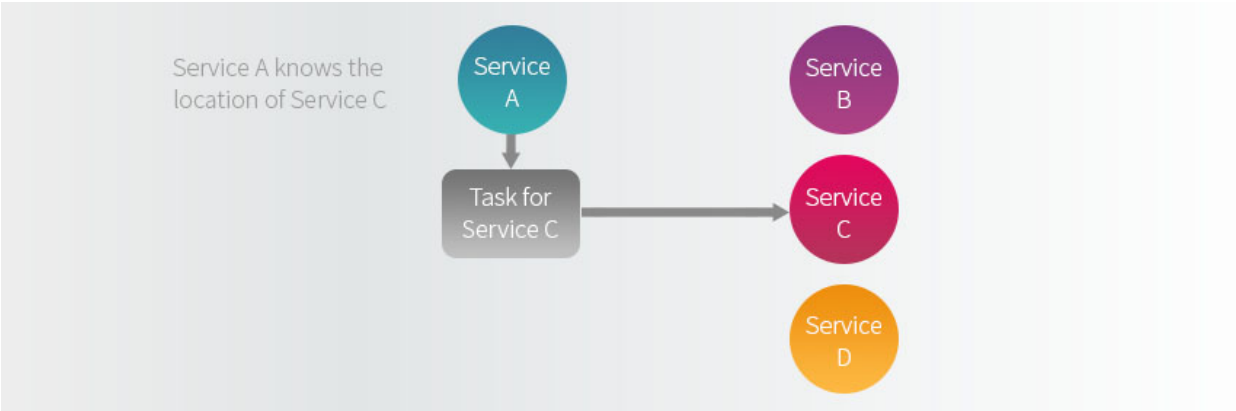
As the service mesh becomes more prevalent as the standard control plane, performance test engineers will need to be familiar with the technology when creating test plans that accommodate architectures that use a service mesh.



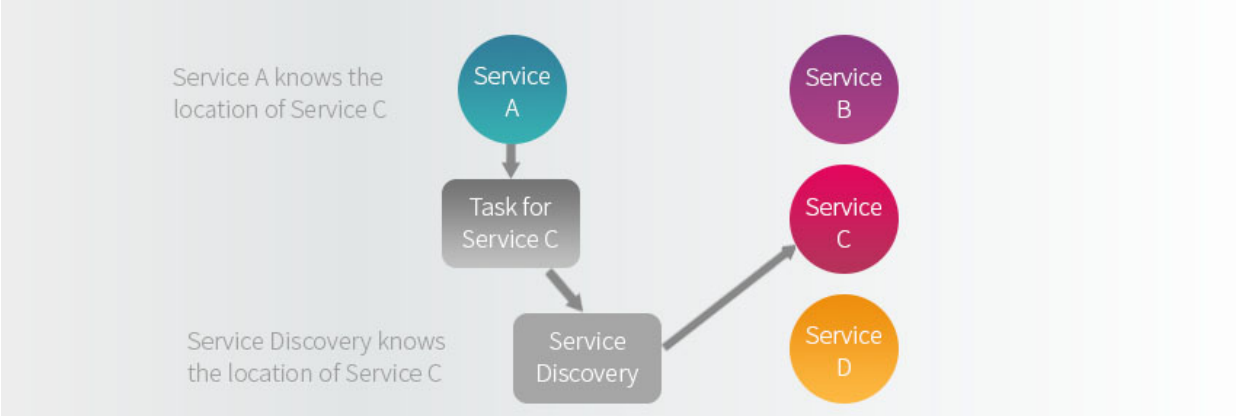
The Case for the Service Mesh

The service mesh solves two fundamental problems in modern distributed computing: finding the location of a service within the technology stack, and defining how to accommodate service failure.

Before the advent of the service mesh, each service needed to know the location of service upon which it depended. In the example below, for Service A to be able to pass work to Service C, it must know the exact location of Service C. This location might be defined as an IP address or as a DNS name. Should the location of the dependent service change, at best, a configuration setting might need to be altered; at worst, the entire consuming service might need to be rewritten.

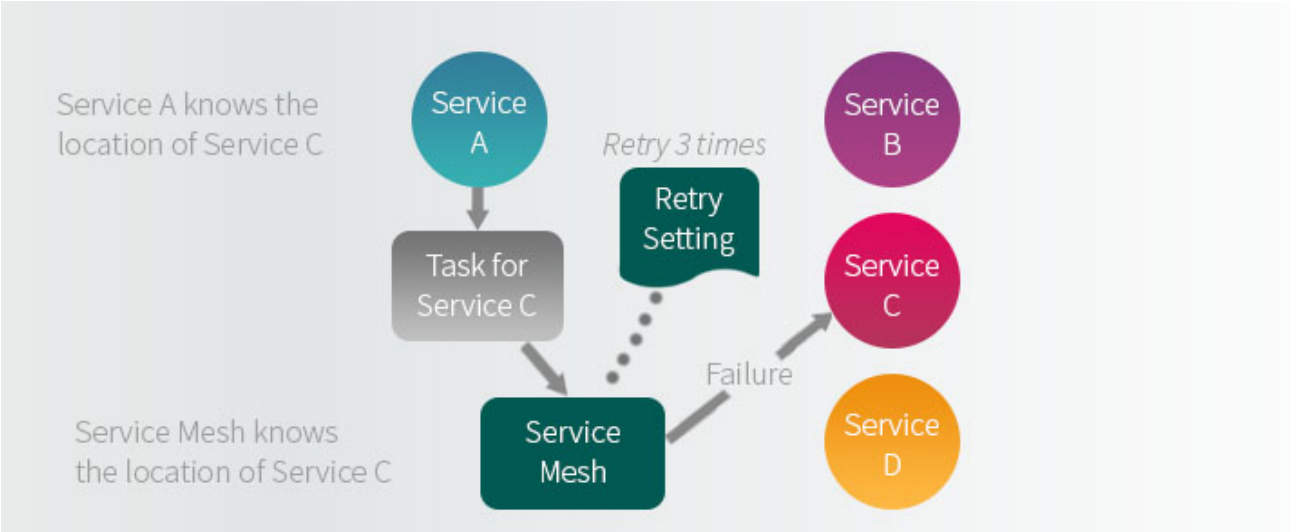


Tight coupling between services proved to be brittle and hard to scale, so companies started to use service discovery technologies such as ZooKeeper, Consul and Etcd, which alleviated the need for services to know the location of other services upon which they depended.



However, one of the problems that was still outstanding was what a service does when one of the dependencies fails. Should the service just error out? Should it retry? If it should retry, how many times? This is where the service mesh comes in.

The service mesh combines, among other things, service discovery and failure policy. In other words, not only will the service mesh allow services to interact with one another, it will also execute retries, redirection or aborts, based on a policy configuration, as shown in the example below.



The service mesh is the control plane that routes traffic between services and provides fail-safe mechanisms for services. In addition, a service mesh logs all activity in its purview, thus providing fine-grain insight into overall system performance. This type of logging makes distributed tracing possible, which makes monitoring and troubleshooting activities among all services in the system a lot easier, no matter their location.

The more popular technologies on the service mesh landscape are Linkerd, Envoy and Istio.

What's important to understand about the service mesh from a performance testing perspective is that the technology has a direct effect on system performance. Consequently, test engineers should have at least a working knowledge of the whys and hows of service mesh technologies. Test engineers also will derive a good deal of benefit from integrating the data that the service mesh generates into test planning and reporting.

Accommodating the Service Mesh in Performance Test Planning

How might performance test engineers take advantage of what the service mesh has to offer? It depends on the scope of performance testing and interest of the test engineer. If the engineer is concerned with nothing more than response time between the web client and web server, understanding the nature and use of a service mesh has limited value. However, if the scope of testing goes to lower levels of any application's performance on the server side, then things get interesting.

The first, most telling benefit is that a service mesh supports distributed tracing. This means that the service mesh makes it possible to observe the time it takes for all services in a distributed architecture to execute, so test engineers can identify performance bottlenecks with greater accuracy. Then, once a bottleneck is identified, test engineers can correlate tracing data with configuration settings to get a clearer understanding of the nature of performance problems.

In addition to becoming an information resource, the service mesh becomes a point of interest in testing itself. Remember, service mesh configuration will have have a direct impact on system performance, and such an impact adds a new dimension to performance testing. Just as application logic needs to be performance tested, so too will service mesh activity. This is particularly important when it comes to testing **auto-retries**, **request deadline** settings and **circuit-breaking** configuration.

An **auto-retry** is a setting in service mesh configuration that makes it so that a consuming service will retry a dependent service when it returns a certain type of error code. For example, should Service A call Service B, and Service B returns a 502 error (bad gateway), Service A will automatically retry the call for a predefined number of times. 502 errors can be short-term, thus making a retry a reasonable action.

A **request deadline** is similar to a timeout. A request is allowed a certain period of time to execute against a called service. If the deadline is reached, regardless of retry setting, the request will fail, preventing an undue load burden from being placed on the called service.

Circuit breaking is a way to prevent cascading failure, when one point in the system — a service, for example — fails and causes failure among other points. A circuit breaker is a mechanism that is "wrapped" around a service so that if the service is in a failure state, a circuit breaker "trips." Calls to the failing service are rejected as errors immediately, without having to incur the overhead of routing to and invoking the service. Also, a service mesh circuit breaker will record attempted calls to the failed service and alert monitors observing service mesh activity that the circuit breaker has been "tripped."

As service mesh becomes part of the enterprise system architecture, performance test engineers will do well to make service mesh testing part of the overall performance testing plan.

Putting It All Together

The days of old-school performance testing are coming to a close. Modern applications are just too complex to rely on measuring request and response times between client and server alone. There are too many moving parts. Enterprise architects understand the need to implement technologies that allow for dynamic provisioning and operations without sacrificing the ability to observe and manage systems, regardless of size and rate of change.

As the spirit of DevOps continues to permeate the IT culture, the service mesh is becoming a key component of the modern distributed enterprise. Having a clear understanding of the value and use of service mesh technologies will allow test engineers to add a new dimension to performance test process and planning, and performance testers who are well-versed in the technology will ensure that the service mesh is used to optimum benefit.

About TestRail

We build popular software testing tools for QA and development teams. Many of the world's best teams and thousands of testers and developers use our products to build rock-solid software every day. We are proud that TestRail – our web-based test management tool – has become one of the leading tools to help software teams improve their testing efforts.

Gurock Software was founded in 2004 and we now have offices in Frankfurt (our HQ), Dublin, Austin & Houston. Our world-wide distributed team focuses on building and supporting powerful tools with beautiful interfaces to help software teams around the world ship reliable software.

Gurock part of the Idera, Inc. family of testing tools, which includes Ranorex, Kiuwan, Travis CI. Idera, Inc. is the parent company of global B2B software productivity brands whose solutions enable technical users to do more with less, faster. Idera, Inc. brands span three divisions – Database Tools, Developer Tools, and Test Management Tools – with products that are evangelized by millions of community members and more than 50,000 customers worldwide, including some of the world's largest healthcare, financial services, retail, and technology companies.