AWS Whitepaper

AWS Graviton Performance Testing: Tips for Independent Software Vendors

AWS Graviton Performance Testing: Tips for Independent Software Vendors: AWS Whitepaper

Table of Contents

AWS Graviton Performance Testing: Tips for Independent Software Vendors

Publication date: **September 15, 2021** (*Document history*)

This whitepaper is for decision makers and builders at independent software vendors (ISVs) who are unsure about how to evaluate Amazon Elastic Compute Cloud (Amazon EC2) instance performance and want to learn about best practices and common pitfalls.

The evolving price performance of Amazon EC2 instance types leads to better performance at lower cost for Amazon Web Services (AWS) customers. Using the example of AWS Graviton, this whitepaper shows how to define your test approach when evaluating EC2 instances, set success factors, and compare different test methods and their implementation.

Are you Well-Architected?

The AWS Well-Architected Framework helps you understand the pros and cons of the decisions you make when building systems in the cloud. The six pillars of the Framework allow you to learn architectural best practices for designing and operating reliable, secure, efficient, cost-effective, and sustainable systems. Using the AWS Well-Architected Tool, available at no charge in the AWS Management Console, you can review your workloads against these best practices by answering a set of questions for each pillar.

For more expert guidance and best practices for your cloud architecture—reference architecture deployments, diagrams, and whitepapers—refer to the AWS Architecture Center.

Introduction

AWS Graviton processors are custom-built by AWS to deliver the best price performance for cloud workloads running in Amazon EC2. Better price performance translates into cost savings for customers, enabling independent software vendors (ISVs) whose platforms run on AWS to reduce their cost of goods sold (COGS).

It can also provide a better experience for ISV customers. Many ISVs run their own performance testing efforts to evaluate the price performance of Graviton and to model the impact of unit cost reduction and performance-related reductions in fleet sizes on their bottom line. This paper

provides an overview of Amazon EC2 performance testing best practices and common pitfalls to help you obtain reliable, actionable results.

This paper begins the discussion by introducing a set of key concepts (layers of a typical ISV software stack) and then moves to a discussion of performance testing steps and best practices (define, implement, instrument, run, and visualize performance tests).

What is AWS Graviton?

AWS Graviton processors are custom-built by AWS to deliver the best price performance for cloud workloads. The Graviton processor is one of three processor options and powers Amazon EC2 instance types for general purpose, compute-optimized, memory-optimized, and storage-optimized use cases. Instances powered by Graviton are available in most AWS Regions, as well as GovCloud and the AWS China Regions.

Launched in 2019, Graviton2 is the second generation of AWS Graviton processors. Graviton2-based instance types offer up to 40% better price performance compared to fifth generation instances. (The first generation (A1) of Arm-based, Graviton-powered EC2 instances were launched at re:Invent 2018.) The feature set of the Graviton processor is optimized for cloud workloads and offers the following benefits:

- Large L1 and L2 caches for every virtual central processing unit (vCPU), which means a large portion of your workload will fit in cache without having to go to memory.
- Every vCPU is a physical core, meaning more isolation between vCPUs and no resource sharing between vCPUs except last level cache and memory system.
- Cores connected together in a mesh with ~2TB/s of bisection bandwidth, allowing applications to move very quickly from core to core when sharing data.
- Graviton's memory architecture means you don't need to worry where application memory is allocated from, or which cores are running the application.

Benefits for ISV applications

ISVs use AWS to define, build, operate, and market applications to AWS customers, extending the choice of AWS customers in application domains including security, data analytics, observability, storage and backup, and business applications.

ISV applications benefit from the adoption of Graviton in a number of common ISV use cases, including Software as a Service (SaaS) offerings and the distribution of software as marketplace products to AWS customers. Some of the typical benefits include:

- Better price performance compared to current generation EC2 instance types, leading to higher throughput and lower latency for common cloud workloads.
- Lower COGS, a result of unit cost reductions and the optimization of server footprint based on Graviton's performance advantage.
- Broad ecosystem support for common server operating systems, programming language runtimes and libraries, open-source software such as databases and in-memory-caches, enabling ISV to migrate their applications without the need for refactoring.
- Improved customer experience, leading to higher customer retention and lifetime value as Graviton delivers unparalleled performance for a broad range of ISV workloads.

Defining your test approach

Selecting the right test approach for your workload is important when deciding whether to phase in Graviton. Start with observable properties of a system such response time, latency, throughput, and error rates before considering systems performance at a more granular level. This approach is referred to as *outside-in* testing.

What it does

An outside-in approach to performance testing allows you to analyze workload performance in the context of customer experience. Customer experience plays an important role in helping your customers adopt and implement your application. Performance testing obviously does not influence the feature depth of your application, but can help with non-functional requirements; in particular, tuning application throughput and latency.

By taking an outside-in approach, you place customer experience first, and then evaluate how different combinations of compute resources such as vCPU, memory, input/output operations per second (IOPS), and network bandwidth impact on performance variables.

How it works

Start by defining important customer experience outcomes your application needs to achieve —such as application throughput, latency, and error rates. Throughput describes the number of requests an application successfully processes within a given unit of time (such as a second, minute, or hour). Latency describes the delay between a request being sent and the acknowledgment of success being received. Error rates describe the number of requests dropped due to some internal failure of the application or its underlying system resources.

Describing customer experience in this way turns a qualitative outcome (customer experience) into a quantitative measure, enabling you to analyze customer experience under different load scenarios and resource configurations.

When to consider this approach

Consider using an outside-in approach when you want to evaluate application performance in the context of customer experience. Graviton's performance benefits drive internal efficiencies and

service improvement, leading to better customer experience and retention. For example, Graviton demonstrates higher throughput for workloads such as Redis and OpenSearch, enabling you to run smaller instance fleets without materially affecting customer experience.

Using an outside-in approach enables you to understand Graviton performance in the context of a real workload, something that is difficult to establish by using standard benchmarks at the resource level. Using standard benchmarks at the resource level (micro-benchmarking) presents an idealized version of the resource under test, as the input and load are pre-determined and consistent (for example, driving the same block size to disk or performing a number of system calls within a unit of time). In reality, the input to and load on a system will fluctuate based on requests from your users or other applications (for example, one user request performs a point query while another performs a range query).

Common pitfalls

Performance testing is an iterative process. Understanding performance bottlenecks is key, as resolving one bottleneck often moves the performance problem somewhere else. For example, resolving a throughput issue by selecting an instance type with higher vCPU count and tuning your application for higher parallelism may not yield the desired results if your load testing configuration runs as a single thread. In this case, resolving a vCPU bottleneck in the system under test moved the throughput issue to the load generator. It is important to understand system or architecture bottlenecks and establish acceptable thresholds early on in order to avoid spending your tuning efforts on a single bottleneck.

The following figure summarizes the things to consider when evaluating performance bottlenecks of a typical ISV application.

The remainder of this paper focuses on aspects of application performance such as throughput and latency, as well as metrics for host system resources such as CPU, Memory, Disk, and Network and their impact on application performance.

Other considerations such as operating system tuning, the selection and tuning of Amazon Elastic Block Store (Amazon EBS), and performance differences in hypervisor technology are out of scope. However, this paper provides links to additional reading resources where appropriate.

Software/hardware stack	Example:	Performance testing objectives:
Application/runtime	Docker, Node, Python, Java, Go	Latency, throughput, utilization, error rates, price
Operating system	Amazon Linux, Ubuntu 18.04	Context switching, swapping, paging, interrupts
Hypervisor	HVM, PV, Nitro	Resource contention/isolation, hardware acceleration
Hardware/host system resources	Intel Xeon, AMD Epyc, AWS Graviton	Utilization, saturation, run-queue length

Layers of a typical ISV application

For a complete discussion of systems performance concepts, performance testing theory, and advanced topics not covered by this whitepaper, consult a performance testing textbook such as *Systems Performance* by Brendan Gregg.

Define success criteria

Any performance testing initiative needs to start with a clear understanding of what you are trying to achieve. This section provides advice on setting goals for performance testing and how to communicate objectives such as performance and cost optimization to business stakeholders.

Improving customer experience

Customer experience is an aspect of competitive differentiation for ISV applications. Examples of customer experience outcomes such as faster response time, lower latency, and reduced error rates due to lower saturation and less requests dropped. These outcomes result in higher customer satisfaction with a service, leading to improved retention and higher lifetime value.

Graviton's performance advantage translates into higher throughput and lower latency for common cloud workloads such as databases and search clusters. This means your application can serve more user requests and complete them in less time, leading to improved customer experience and retention. The introduction of mixed instance policies (instances with different characteristics such as the processor type) in Amazon EC2 Auto Scaling enables you to phase in instances powered by the Graviton processor and to monitor their performance in a real-world setting (a practice known as A/B testing). When instances underperform, you phase them out.

Reducing computing cost

Computing costs are a key input into lowering COGS, an important measure of SaaS profitability. Industry benchmarks for publicly listed SaaS companies are in the 60–80% range measured as gross profit margin (SaaS revenues less COGS divided by SaaS revenues). This means that lower unit costs per compute instance and smaller instance footprints are important for SaaS providers that are publicly listed and report their financial data in earnings calls.

Graviton's competitive pricing and performance advantage means you can lower computing cost by reducing unit costs and running smaller server footprints. Graviton instances offer up to 40% better price performance compared to current generation instance types. Instances are on average 10–20% cheaper than alternatives in the same instance family.

Graviton's performance advantage also enables you to run smaller server footprints, leading to less instance to maintain and pay for in production. The combination of these benefits leads to lower

COGS, an important measure of profitability and internal efficiency for SaaS providers that are publicly listed.

Measuring price performance

Price performance is the ability of a system to deliver performance at a particular price. The price performance metrics is often used in performance engineering to compare different systems. In the context of modeling the impact of unit cost reduction and performance-related reductions resulting from the introduction of Graviton, use the following definition:

$$Performance\ to\ price\ ratio\ (P) = \frac{Performance\ gain}{Price\ reduction}$$

For example, if you migrate from c5.2xlarge to c6g.2xlarge and see a 5% performance increase, you would observe a 34.6% performance/price improvement (1.05/0.78 – or 5% performance increase, with c6g.2xlarge being 78% the cost of c5.2xlarge per hour). If the intent of your migration is exclusively to lower the cost of operating the workload without impacting the current user experience, then you will have accomplished this goal.

$$P = \frac{1 + 0.05}{0.78} = 1.346 = 134.6\%\ (a\ 34.6\%\ performance\ price\ improvement)$$

Different ways to implement your test

This section discusses different ways to test and evaluate workloads running on EC2 and container orchestration platforms such as Amazon Elastic Container Service (Amazon ECS) and Amazon Elastic Kubernetes Service (Amazon EKS). These implementations range from easy to complex and can be used alone or in any combination. While other options such as micro-benchmarking exist, these methods are more appropriate for testing the impact of performance-related changes, rather than providing a holistic view of the general performance of an instance in a workload.

An application's *bounding resources* are the system level resources (compute, disk, memory, network) on which your application is most dependent.

For example, if an application spends the majority of its time using the CPU, this application is *CPU bound*. If the CPU is faster, the application would be faster. If an application performs better with a faster I/O subsystem (such as disk or network), then this application would be *disk* or *network I/O bound*.

Following is a table that highlights four methods of testing, when to use them, potential constraints, and how to overcome the constraints.

Table 1 – Overview of performance testing methods

Method	When to use	Potential constraints	How to overcome
Instance selection flow chart	When making your initial instance selection and you do not have much prior data	Not really indicative of actual performance or performance bottlenecks	Use the synthetic load testing or test on real workloads
Synthetic load testing	When you want to test resources against real workloads but are unable to phase them into your production environment	Baking assumptions into your load testing script that do not reflect reality	Consider replay

Method	When to use	Potential constraints	How to overcome
Replay	When you have actual data that represents user behavior/requests reliably and you can replay it in a test environment or you can use traffic shadowing to send a copy to your test environment for processing	Can require specific tooling be in place to duplicate customer requests	Consider testing on real world workloads
Testing on real workloads	When you have well-defined key performance indicators (KPIs) that can be used to measure impact of introducing a new instance type	Can be complicated to implement and instrument	Implement blue/green (or red black) deployments

Use the following for all examples

You have a workload that performs aggregations on data over a distributed set of instances, then joins the aggregate result together and returns the result to a customer (for example, MapReduce). In this workload each node in the cluster receives data, decrypts the data, performs an aggregation, then returns the resulting dataset. Based on this description you might determine that this workload is *compute bound* (for aggregations) and *network bound* (for sending data between nodes).

Like for like comparison

Picking a new instance using a "like for like" comparison selects a new instance using a simple flow chart to provide a starting point. If your workload is already running on EC2, you can start by using the newest generation of the current instance type you are using, or Graviton equivalent (for example, if you are using c5, you might test c6g). If you are not currently running your workloads on EC2 or this is a new workload, then you can use a flow chart (following) and the characteristics of your workload to provide a starting point.

It is important to note that, even with the flow chart, you should test several instance types to ensure you are choosing the right one for your workload.

Why this approach?

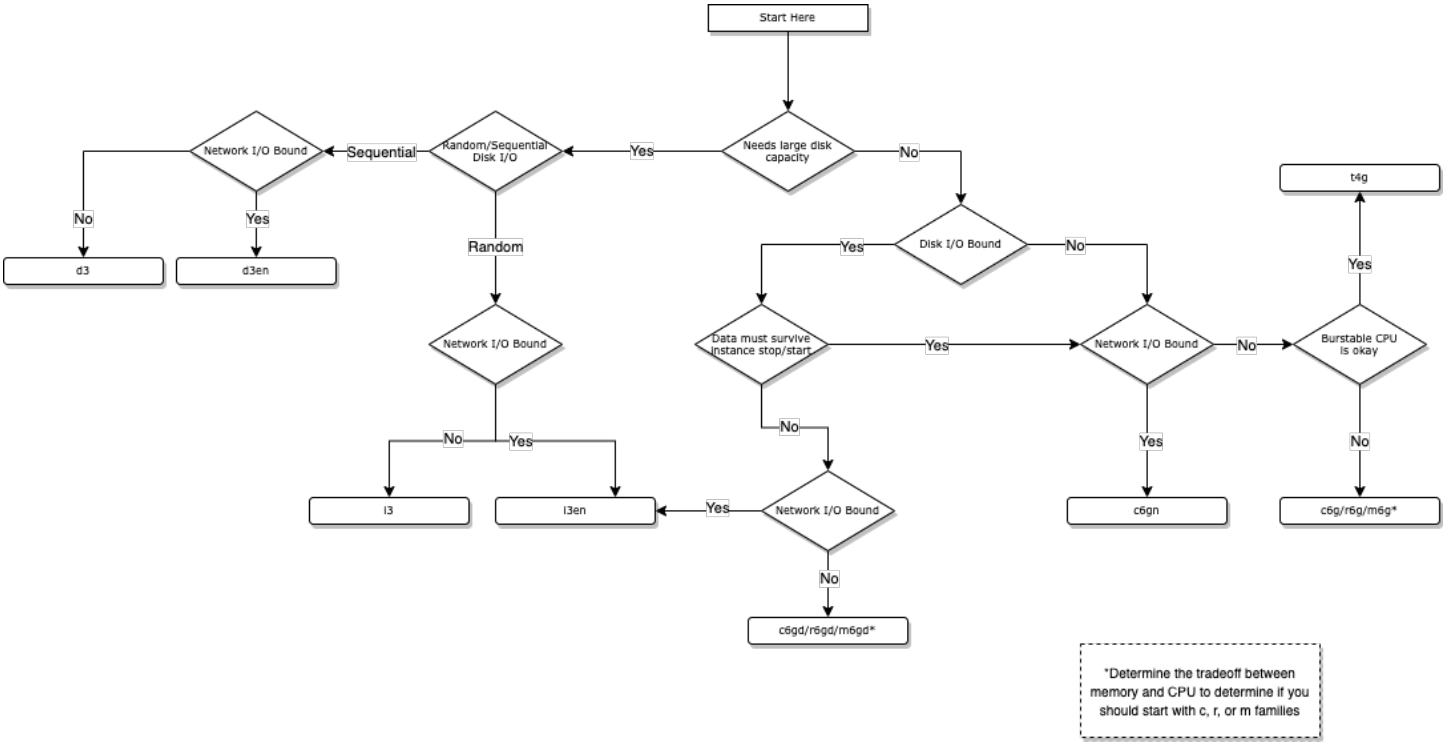
When evaluating new instance types, the easiest way to get started may be to select an instance and run some tests based on coarse information you already know about the workload (assuming you've determined your success criteria). This is a straightforward approach that enables you to build a simple mental model and start testing quickly. Often you will start here, then test the instance type against real workloads, or run macro-benchmarks (synthetics) and micro-benchmarks (benchmarking), both of which are covered below.

Examples of when this approach is most effective include the following (but not limited to):

- This is the first time you are deploying this workload on AWS
- There is no dedicated performance engineering team at your company
- You want to try out new instances for your workload without spending a lot of time and effort

Implementation

This implementation uses a flow chart to determine which instances may be the best fit for your workload. For example, if you have a workload that uses local disk for a fixed size ephemeral cache that is bound to network input/output (I/O), you could arrive at the c6gn instance type, then determine the instance size based on the number of CPU cores and amount of memory your application requires.



Instance selection flow chart

Synthetic load testing

This section outlines using synthetic load testing, why you might take this approach, and how you can implement the approach. Many off-the-shelf applications (such as MySQL and Hadoop) come with synthetic load generators built in. When it comes to testing applications developed by your organization, however, you will need to develop your own synthetic tests. Fortunately, there are open-source libraries and applications that can help, which are covered further in the implementation section.

Why this approach?

If you have selected an instance via the flow chart in the like-for-like implementation section, or by random choice, and don't have the ability to test workloads on real user requests, this approach is ideal.

Use the behavior observed in production environments (such as transactions per second, traffic patterns on API endpoints, or frequency of job completion) to develop synthetic load tests, then run them in development or staging environments.

The benefit of this approach is that you remove the risk of impacting application performance for your real customer base. While this is a great benefit, there are caveats to this approach.

For example, if synthetic tests based on user behavior are not frequently audited, they could become stale and incorrect. Further, any bias from the designer of the load test could unintentionally add incorrect assumptions into the test that don't reflect the reality of your users.

Implementation

For HTTP workloads, there are several workload generators. The wrk2 project is an excellent option that can be used to synthesize a significant load of user requests that accurately represents system latency by accounting for the effects coordinated omission.

For off the shelf applications, like MySQL, MariaDB, and Hadoop, many come with their own synthetic generators. For MySQL and MariaDB there is mysqlslap, for Hadoop there is the aptly named Synthetic Load Generator.

To run the synthetic tests, deploy your workload to a development or staging environment that mirrors the setup for production using the newly selected instance type, then run the tests. Be sure to have the necessary instrumentation for the test, so you can compare to previous data. Instrumentation and visualization are covered in the How to instrument your test section.

Replay

In this section we discuss replaying user requests to a different environment to test new instances. While this approach is the most technically complex, it does allow for maximum flexibility while solving for the caveats of the synthetic load testing approach.

Why this approach?

This approach enables you to test new instances for your workload without introducing them directly to your user base, reducing the risk of an underperforming instance causing a negative impact to customer experience. Furthermore, using real user requests removes the possibility of unintentional test design bias and the risk of test configuration drifting away from the reality of your workload.

This approach is generally beneficial for testing changes any change to your workload, including new EC2 instance and changes to the application code. By implementing the replay approach.

you can test freely without impacting your customer base. Changes other than instance types are outside of the scope of this paper, but warrants a mention as this approach allows you to implement multiple types of tests on your system.

Implementation

Implementing this approach requires a development or staging environment that is a mirror of your production environment. You must deploy additional software on your workload instances, load balancer, or API Gateway to collect and replay user requests to your test environment.

There are several to accomplish the replaying of traffic. GoReplay is a popular open-source solution that runs a single Go binary on your instance to replay requests to a given target endpoint. Ambassador is open-source API Gateway that can run on Kubernetes and replicate user requests using a mechanism they refer to as traffic shadowing. Lastly, Amazon Virtual Private Cloud has a traffic mirroring feature that enables you to copy network traffic from the elastic network interface of an Amazon EC2 instance.

Testing against real workloads

The last and most mature approach is to test new instances in the context of a real-world workload. Testing new instance families on real user requests provides the best insights into how they will perform on your workload and how they compare to your current instance family of choice.

With this approach, you gradually phase Graviton instances into a workload starting with a small percentage of requests. This provides the benefit of seeing your workload running on Graviton and handling real requests, with the additional benefit of being able to quickly remove the instances and failover to your original instance family if needed (if, for example, you see increased latency or increased numbers of failed requests).

If your predetermined success criteria are met with Graviton on a small percentage of traffic, consider gradually increasing (weighting in) the amount of traffic that the Graviton instances are serving within the context of your workload.

The percentage of traffic you serve is ultimately up to you and the needs of your business, but consider starting with a small percentage of your traffic and scaling up to 50% of all traffic. Scaling to a larger percentage of total traffic will allow the resulting dataset to be representative of all users in your workload, and normalizes the resulting metric dataset.

Why this approach?

With this approach, determining the success criteria can be based on the same KPIs of the production workload. For your organization, these metrics may include (but not be limited to) requests per second (RPS), request latency (such as end-to-end, first-byte), time-to-job completion, and pass/error rates.

If these are not known or are not formally written down within your organization, discuss with your teammates or engineering leadership to determine what will make this a success. If you don't have well-defined success criteria, as mentioned earlier in this paper, consider starting with the RPS, request latency, time-to-job completion, or pass/error rates for your current workload as the performance baseline, then determine concrete outcomes from there.

Suppose you define the success criteria for an instance type as follows:

- End-to-end P50, P95, P99 and P99.9 request latency should be less than the current instance
- Pass rate should be the same or higher
- Error rate should be the same or lower

As with the other approaches, it is critical to remember the overall intent of your migration when using this approach. Consider that, in this example, even if request latency and pass/error rate were identical, the price of operating the same number of instances with Graviton would be less expensive than the x86 equivalent, assuming you stayed within the same instance type and size.

Implementation

The goal of this approach is to see how new instances perform when running under user load. While testing instances in an isolated environment with synthetics can expose many important performance characteristics, running a real-world workload will allow you to measure and compare directly against your existing instance family from the perspective for your users.

Continuing with our data aggregation example workload, consider a service operates on 24 EC2 instances fronted by an Application Load Balancer (ALB), with Amazon Route 53 providing a friendly DNS record.

Given this setup, there are multiple approaches to allow for flexibility to control which instance type receives traffic, and the percentage of how much each processes.

- **Multi-instance type deployment** — Deploy a subset of your instances as Graviton2 running behind the same load balancer
- **Blue/green deployment** — Duplicate your entire stack, running all instances on Graviton2, then use DNS to distribute requests across each stack

As an example of a multi-instance type deployment on AWS, you might create a new target group, then register a set of Graviton instances to the target group. Continuing the example of 24 instances, you could start with three Graviton2 instances in the new target group, then leave 21 of the instances in the existing target group.

Next, modify the listener or rules on the ALB to forward requests to different target groups. With this scenario, 12.5% of the requests would run on Graviton2. More information can be found in the documentation for Auto Scaling groups with multiple instance types and purchase options.

Both options provide the flexibility to control the percentage of requests served to the different backends. Consider the first option when you have the ability to retrieve the desired metrics from each host (for example, from application logs) and have historical data for target response time for the ALB that you can use later to compare to current.

Consider the second approach when you want to keep the ALB metrics separate for each workload or if you desire fine grained granularity of traffic distribution between the two endpoints (Route 53 allows you to go as low as 1/256th of the traffic, or ~0.004%). This list is not comprehensive and it is important to keep in mind any other characteristics of your setup when deploying these setups.

How to instrument your test

This section covers example metrics to collect, options to instrument your tests to collect these metrics, and recommendations on aggregating and viewing results.

The way you instrument a test should be dependent on the way you run your test (See the Different ways to implement your test section). There is no "right" or "wrong" way to instrument a test, or guidebook to tell you what metrics you collect. This is dependent on what matters to your business and your users.

This paper starts by focusing on three categories of metrics: Business, Application-level, and System-level. Each metric category can be used alone or in combination with another category, then paired with a single or more than one test approach.

For example, if you are testing a new instance against real workloads, you might already have instrumentation in place that allows you to measure application and system-level performance at the instance level. You could also use a business metric, such as the number of dropped requests, to measure the impact these instances have on your customers.

You should experiment to find the right combination of metrics that work for you and your business. More metrics may help you understand a problem if it arises at the expense of creating additional data to sort through.

Types of metrics

Business metrics

Business metrics include concepts such as customer sentiment and service-level agreements (SLAs). Business insights help explain whether customers are likely to be satisfied with your service after introducing a change. For example, increased request latency may historically be correlated with lower customer sentiment, leading to higher churn or abandoned transactions.

By monitoring application and system-level metrics and putting them in the context of business insights, you ensure that performance testing and tuning focuses on areas of high priority to the business. Data collection may require you to look beyond the system you are testing to include data sources such as customer surveys or qualitative results from end user testing.

Data sources: surveys, support tickets

Application-level metrics

Application-level metrics include concepts such request rate, job run length, request latency, and error rates. Application-level metrics enable you to measure certain aspects of application performance and are an example of an external metric (observable by an application user).

For example, the application request rate measures the number of requests an application is able to process in a given unit of time. While application-level metrics describe an important business-level outcome, they need to be correlated with system-level metrics in order to gain insight into the root cause of performance degradation or improvement.

On Graviton-based systems, applications have access to the full performance entitlement of the underlying core. This translates into improved performance and higher request rates for cloud workloads.

Data sources: application metrics, logs, and alerts

System-level metrics

System-level metrics include concepts such as CPU utilization, CPU wait time, disk queue depth, and status checks. System-level metrics enable you to measure performance aspects of system components such as vCPU, memory, storage, and networking, and underlying hardware health.

You can correlate system-level metrics with application outcomes such as request rate to gain better insight into the overall performance of your system. Graviton offers a feature set that is optimized for cloud workloads. Each vCPU offers better resource isolation, full access to the performance entitlement of a physical core.

Graviton's improved memory architecture offers ~2TB/sec bisectional bandwidth, allowing data to move faster between cores. The combination of these features gives you access to unparalleled performance at the lowest cost in a family.

Data sources: system metrics, logs, and alerts

Next, this paper focuses on tools you can use to instrument your tests.

Test instrumentation

In this section we'll discuss options for instrumenting tests. We cover two options, using instrumentation at the cloud infrastructure level and using third-party provided tools. Other

options, such as system or instance level instrumentation that make use of tools such as eBPF, are outside of the scope of this paper. You can find a list of reading material on eBPF in the Further reading section of this paper. Regardless of how you choose to instrument your test, the metrics you collect for later analysis can be collected from a single test or multiple tests.

Cloud infrastructure instrumentation

This option selects Amazon CloudWatch Metrics to collect performance data at the cloud infrastructure resource level. Highlight the CloudWatch agent.

Using the CloudWatch agent to instrument your tests allows you to leverage all the pre-existing features and dashboards of Amazon CloudWatch, and reduce the amount of manual work you need to do to aggregate result data. Starting to use the agent is as simple as installing it, creating the CloudWatch agent configuration file, and starting the agent.

Once the agent starts emitting metrics, you can begin visualizing the data in CloudWatch dashboards and (optionally) share the dashboards with your team and leadership. When it comes to aggregating data, CloudWatch has built in support for statistics and percentiles that you can make use of in your visualizations.

Third-party options for Instrumentation

Instrumenting your tests with a third-party monitoring solution provides similar to the benefit of using Amazon CloudWatch Metrics. There are many third-party solutions, too many to list within the scope of this paper, each with their own benefits.

As with CloudWatch Metrics, the biggest benefit is that these solutions generally are agent-based and provide meaningful data and visualizations "out of the box", with little configuration required. The benefit of these tools is that if your organization is already making use of them, the amount of effort required to implement data collection on a set of instances you are testing should be minimal.

Test running

Running your tests

When running your tests, there are several factors to take into consideration to help you build an effective result set.

The first you may consider is the amount of time that you run a test. If you are running a benchmark or synthetic test, are you running it for a length of time that is representative of your workload?

If running a new instance against a real-world workload, are you letting it service user requests for an amount of time that includes peak usage and allows you to derive meaningful insights from the data?

Another factor is the number of instances in the test setup. If you are running a test one time on a one instance, your result set is representative of that instance, and you might experience a different result when scaling up a workload across hundreds or thousands of instances. This number may vary depending on the type of test you are running.

Aggregating and viewing results

Aggregating result data helps you reason about workload performance as a whole, rather than the performance of a single well or poorly performing node. At first, you might be tempted to use the average (or mean) over a set of common servers.

Average or mean, however, will only tell you that 50% of the workload is performing worse. To determine the long tail of performance, and quantify what is "worse", you might make use of the P95, P99, and/or P99.9 of the result set.

Aggregating results is one step, but visualizing the data is what will help make it meaningful. Seeing the visualization will help you "see" what's wrong and give you an indication of where to start your investigation. Consider displaying each relevant metric and the corresponding values on a distribution graph, rather than focusing on visualizing a fixed set of values. Viewing the data on a distribution graph will allow you to investigate details and outliers at a finer grain.

Benefits and trade-offs

Each of these options have benefits and trade-offs depending on your business, desired metric(s), and timeline.

Amazon CloudWatch and third-party monitoring solutions are able to get you up and running quickly and provide the simplicity of a built-in dashboarding solution, but might not provide the level of granularity that you need when deep diving on performance differences. Other options, such as eBPF, which is outside the scope for this paper, can be as extensive as you can come up with, but might take more time to implement and create consumable visualizations from.

No matter the option you choose, when deciding on instrumentation, be sure to first determine why you want the data you are collecting, what you will use the data for, and how you instrument the collection. This helps avoid overloading yourself with metrics that may not be necessary for profiling your workload, or be relevant to the instance migration.

Other considerations

This section introduces a set of additional considerations that help you make the most of your performance testing project. Consider these points as you plan your performance testing and instance selection project.

Understand key differences

Unlike fifth generation EC2 instances with x86-based processors, sixth generation Graviton2 processors do not use simultaneous multithreading (SMT). This means the vCPU count of an instance matches its physical core count (and vice versa), leading to implications for performance testing and benchmarking when comparing instances with an equal vCPU count.

For example, if your application code uses only half of the vCPU on a fifth generation instance because of the impact of hyperthreading, then this check is no longer required on sixth generation Graviton2 instances and you may want to disable it.

Upgrade operating systems and language runtimes

Before running your test on Graviton, make sure to upgrade your operating system and language runtime to the most recent version. Some older operating systems and language runtime versions are not optimized to run on modern processors like Graviton2.

Your workload will not have access to the full performance entitlement of Graviton2 in this case and will run slower than expected. Upgrade your operating system, runtime version, and code base to a higher version number first. This will give you access to the latest EC2 instance capabilities. Follow the AWS Graviton Getting Started Guide and the AWS Graviton for Independent Software Vendors whitepaper when planning your migration approach.

Test side-by-side

Next, retest your workload in your current environment (your normal test environment using current generation Intel or AMD-based instance types) to measure the effects of the upgrade.

Successful upgrades provide access to the latest EC2 instance capabilities and performance optimization for your instance type, leading to better workload performance and cost.

Retesting your workload in your current environment also allows you to gather data and draw comparisons between Graviton2 and fifth generation instance types in the respective instance family. Once you know that your workload performs satisfactorily, you can then move on to migrating your workload to Graviton2 for performance testing.

Test different instance shapes and sizes

Phase in Graviton instances next and consider testing across multiple instance types and sizes in a given instance family. This enables you to systematically detect issues that relate to instance size such as performance bottlenecks on very small or very large Graviton instance types.

Each Graviton2 vCPU matches a physical core, providing you access to the full performance entitlement. This opens up opportunities to consider other instance shapes (such as instances with less vCPU with same or similar memory capacity) when selecting the right instance for your workload. Also consider other performance tuning advice for Graviton2. Both the Graviton Getting Started Guide and the Graviton for Independent Software Vendors (ISV) whitepaper are great places to start.

Conclusion

This whitepaper introduced a systematic approach to performance testing when phasing in AWS Graviton powered instances in common ISV use case scenarios—including SaaS and marketplace offerings. The paper reviewed different test methodologies and discussed their benefits and limitations. This included outside-in approaches that focus on testing and measuring the impact of performance tuning on customer experience outcomes, such as request latency and error rates.

It also included inside-out approaches that focus on testing the performance of system resources such as CPU, memory, disk, and network in the context of an application workload. The whitepaper discussed three ways in which to implement performance testing in service including Amazon EC2, and offered ways in which you can instrument your workload and gather reliable performance data.

AWS continuously evolves the portfolio of compute instances available on the platform. Make sure to subscribe to our Compute Blog and What's New announcements to stay on top of new instance types and innovations by AWS.

Contributors

Contributors to this document include:

- Karsten Ploesser, Senior ISV Solutions Architect, Amazon Web Services
- Maxwell Moon, Senior ISV Solutions Architect, Amazon Web Services

Special thanks to:

- Jesse Chen, Principal Performance Engineer, Splunk, Inc.
- Derek Feriancek, Performance Software Engineer, Splunk, Inc.
- Ali Saidi, Senior Principal Engineer, Amazon Web Services
- Arthur Petitpierre, Senior Specialist Solutions Architect, Amazon Web Services
- Jeff Underhill, Principal Compute GTM Specialist, Amazon Web Services

Further reading

For additional information, see:

- Gregg, B. 2020, *Systems Performance: Enterprise and the Cloud*, 2nd Edition, Pearson, New York City, New York.
- Calavera, D and Fontana, L. 2020, *Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking*, O'Reilly Media Inc., Sebastopol, California.
- Getting Started with AWS Graviton
- AWS Graviton2 for Independent Software Vendors
- AWS Well-Architected Framework – Performance Efficiency Pillar
- AWS re:Invent 2013 | Day 2 Keynote with Werner Vogels
- AWS Architecture Center

Document history

To be notified about updates to this whitepaper, subscribe to the RSS feed.

Change	Description	Date
Initial publication	Whitepaper first published.	September 15, 2021

Note

To subscribe to RSS updates, you must have an RSS plug-in enabled for the browser that you are using.

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

AWS Glossary

For the latest AWS terminology, see the AWS glossary in the *AWS Glossary Reference*.