Guidelines for Deploying NGINX Plus on Amazon Web Services

Optimize Performance by over 50% and costs savings by 20% with Arm-based AWS EC2 M6g Instances

White Paper

Index

Executive Summary

Amazon Web Services (AWS) recently launched Amazon EC2 M6g, C6g, and R6g instances for general purpose compute workloads. These instances are powered by the AWS Graviton2 Processor featuring 64-bit Arm Neoverse cores.

These AWS Graviton2 based instances deliver significant better performance and cost savings over current generation of M5 instances. They are well suited for web servers, application servers, load-balancers, microservices, gaming servers, mid-size data stores, and caching fleets. These instances also appeal to developers, enthusiasts, and educators across the Arm community as they provide quick, easy, and cost-effective access to Arm ecosystems. In this whitepaper, we explore deploying NGINX Plus with M6g instances.

NGINX is the most popular scale-out web application server, ranked as the world's leading web server powering more than 35% of active websites, according to NetCraft.[1] It is available in two flavors, an open source version and a commercially supported version called NGINX Plus. NGINX Plus for Arm is readily available on the AWS Marketplace, and includes a free trial to make it very easy for developers, startups, and enterprises to get up and running on M6g instances.

In this document, we showcase the cost and performance benefits of deploying NGINX Plus application delivery and web services on Amazon EC2 M6g instances. In our performance testing, we focused on NGINX Plus configured as a reverse proxy server and as an API gateway. Our analysis tests both reverse proxy and API gateway use cases with and without HS256 JWT-based authentication. The HS256 JWT authenticated results demonstrate **average performance gains upto 54% for M6g instances**, compared to M5 instances of the same size.

Performance only tells part of the story, it's even more important to look at price-performance in a real-world scenario. To achieve this, we configured the host infrastructure in a redundant manner that replicates a typical production deployment, and then performed load tests across a range of requests per second (RPS) values using a network traffic generator. To replicate a production deployment, NGINX Plus is deployed

using three nodes and tested across multiple EC2 instance types and RPS values. To allow for redundancy, the maximum node utilization was capped at 66% to ensure the remaining two nodes can handle 100% of the traffic in the case of a potential node failure.

The graphs below demonstrate an average performance benefit of up to 54%. In addition, there is a **20% cost saving** achieved when running on M6g versus M5 instances of the same size:

Table 1-a:
NGINX Plus Reverse Proxy performance with no authentication per instance size between M6g and M5

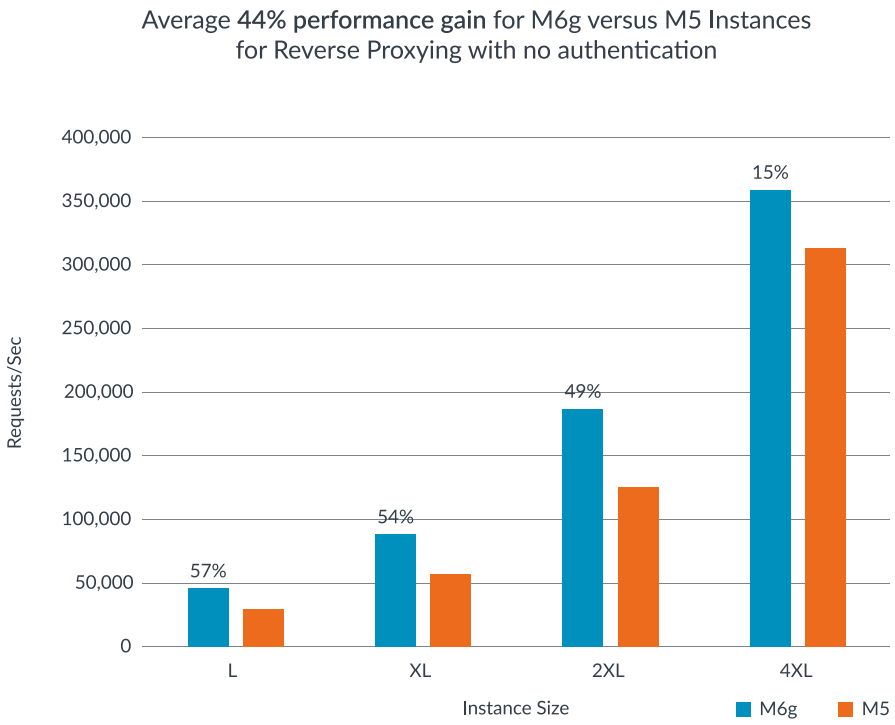


Table 1-b:
NGINX Plus Reverse Proxy performance with HS256 authentication per instance size between M6g and M5

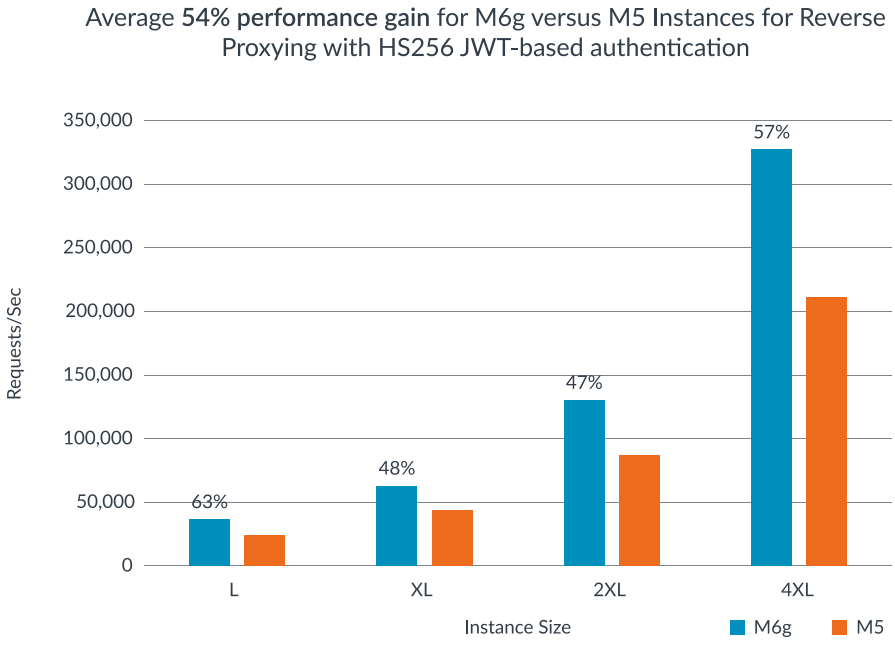


Table 1-c: NGINX Plus API Gateway performance with no authentication per instance size between M6g and M5

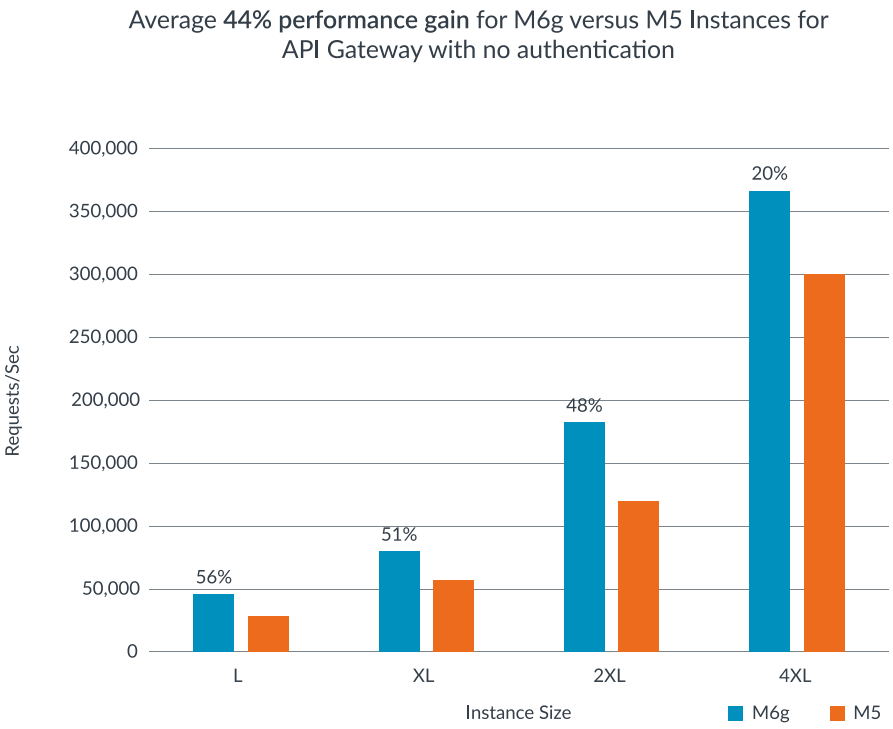
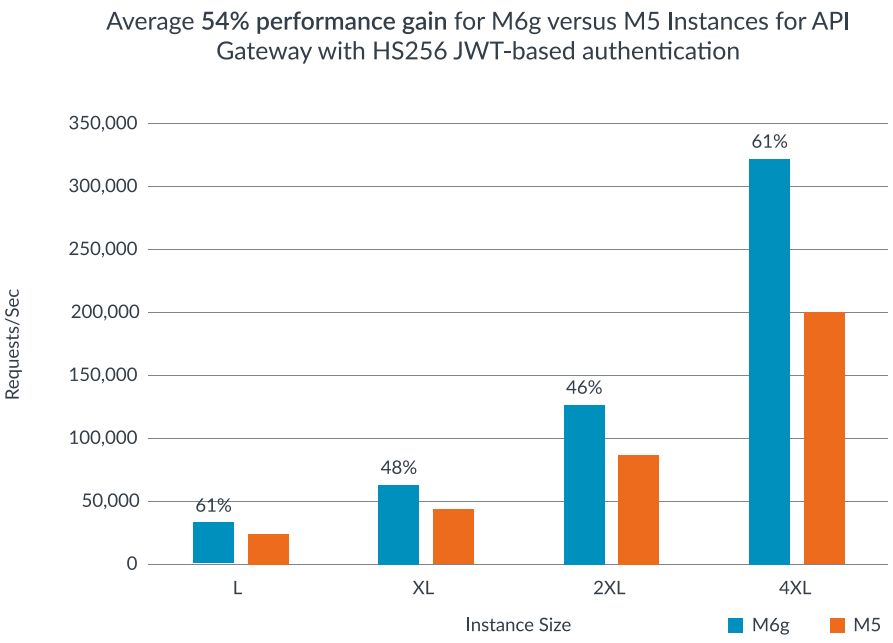
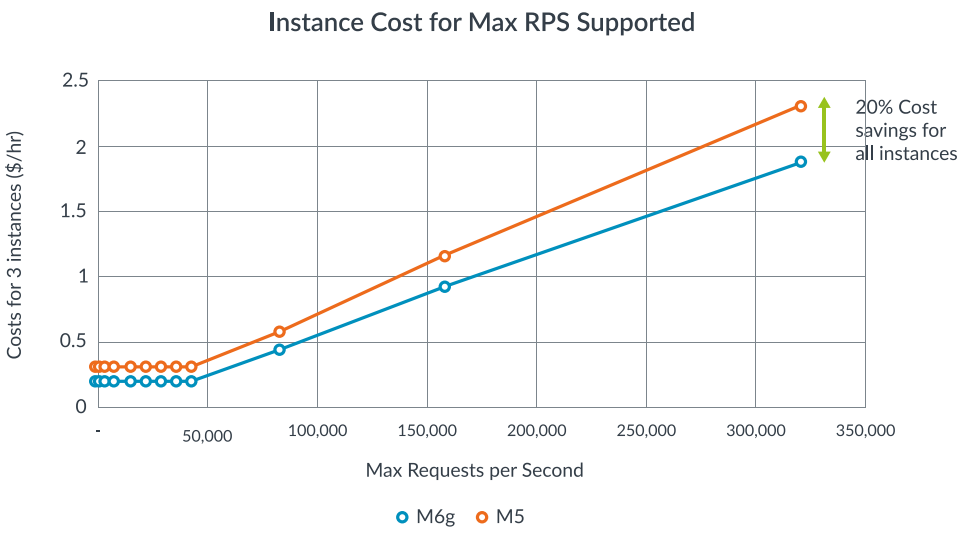


Table 1-d: NGINX Plus API Gateway performance with no authentication per instance size between M6g and M5



When moving from a large to xlarge m6g instance (i.e. double the number of vCPUs), the performance also doubles. Performance scales near-linearly up to 4xlarge instances, where we see the performance scaling start to diminish. This is a result of the processor no longer being the bottleneck as we increase the number of vCPU's when using larger instance sizes.

Table 2: NGINX Plus Cost Savings for M6g vs M5 Instances based on total RPS values



In our results, we demonstrate cost-performance benefits by showcasing RPS values served by various instance sizes. In real-world deployments, customers can further optimize by right sizing their instance choices within a given instance family. This will provide more performance granularity and improved price to performance optimization based on the specific deployment. Also, these cost comparisons are made using on-demand pricing for these instances and results will vary for AWS EC2 Reserved & Spot instances.

Conclusion

These performance results provide you with deployment guidelines for common NGINX Plus configurations across a variety of Amazon EC2 instances and showcase the cost benefits of deploying Amazon EC2 M6g instances. In summary, Amazon EC2 M6g instances demonstrate upto 54% average performance gains and 20% cost savings for scale-out NGINX deployments. This document also provides a framework to deploy additional features and services on M6g instances. With NGINX Amazon Machine Images (AMIs) readily available for M6g instances, you can deploy NGINX Plus[2] on AWS EC2 M6g with ease to achieve the best price-performance for your specific use case.

Try NGINX Plus on AWS M6g instances now.

- NGINX Plus Basic – Amazon Machine Image (AMI) on AWS
- NGINX Plus Enterprise - Amazon Machine Image (AMI) on AWS
- NGINX Plus Developer - Amazon Machine Image (AMI) on AWS

Testing Configuration and Raw Results

This section provides test setup details and performance results for the NGINX Plus Reverse Proxy (RP) and API Gateway (APIGW) features. The performance of these two functions often determine the overall performance of many of the NGINX deployments on AWS today.

Test Setup

The test setup is designed to measure the throughput of an RP or an APIGW. The throughput metric generated by this test is HTTPS Requests Per Second (RPS). For client traffic generation, we used the open source benchmarking application wrk2.

The following describes the test flow:

1. The client (wrk2) sends an HTTPS request for a 1kb static file to the RP/APIGW.
2. If JWT authentication is not enabled, this step is skipped. If JWT authentication is enabled, the RP/APIGW verifies the signature with the appropriate algorithm. If the signature is deemed valid, we move on to the next step. Otherwise a 401 error is returned to the client.
3. The RP/APIGW checks if the requested Uniform Resource Identifier (URI) should be rewritten. If the URI is not rewritten, this is the RP case. If the URI is rewritten, this is the APIGW case.
4. After the URI check and potential rewrite, the request is sent to one of the upstream server instances (round robin load balancing in our case).
5. The upstream that receives the request responds to the RP/APIGW with the requested file.
6. Last, the RP/APIGW sends the file to the client to complete the request.
7. Our test repeats the above steps as often as it can for a 60-second duration. At the end of the 60s test, the client (wrk2) calculates the RPS (i.e. throughput) achieved.

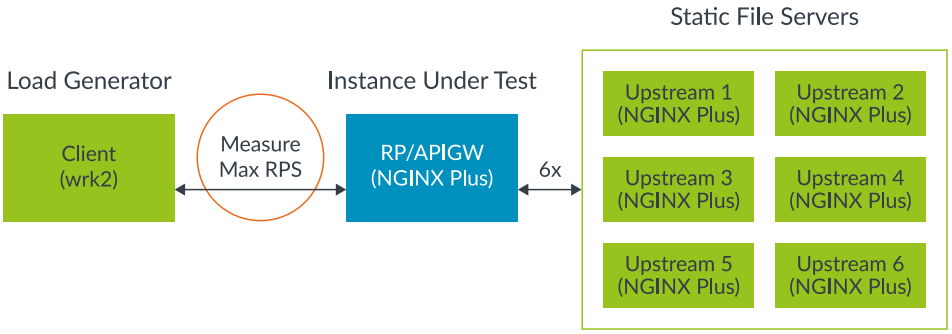


Figure 1: Test Setup

In the diagram above, the instance that is running the RP/APIGW (middle - blue) is being measured for throughput. To ensure the client and upstreams (left/right - green) are not a bottle neck, we selected very large instances for those components of the test setup (see Appendix B – Load Generator & Upstreams Config).

The payload size is a relatively small 1kb because larger files stress the AWS network rather than the instances. Since our interest is in the performance of the instances themselves, smaller payloads are more appropriate for this test. The complete NGINX configs used for all instances in the diagram above can be found in Appendix D – NGINX Configurations.

The following table shows the various instance types we used for the RP/APIGW:

Table 3: AWS EC2 Instances

AWS Instance	Sizes	Number of vCPUs
M6g	Large XLarge 2XLarge 4XLarge	2 4 8 16
M5	Large XLarge 2XLarge 4XLarge	2 4 8 16

Additional instance configuration can be found in Appendix A – Reverse Proxy/API Gateway Configs.

For more information on AWS EC2 instances, please visit https://aws.amazon.com/ec2/instance-types/

Test Scenario #1: NGINX Plus reverse proxy

In our setup, a single AWS EC2 instance is stress tested with NGINX Plus RP functionality to achieve maximum RPS per instance using the load generator, which self-throttles based on increased latency values for the responses.

In the real world, these instances are configured with N+1 configuration where N is the number of instances required to meet the performance requirements[4] (typically a minimum of two) to achieve high availability. In addition, there is at least one instance reserved as backup in the event of an instance failure. We configured three nodes to achieve an N+1 configuration with each node capped at a maximum

of 66% utilization based on the maximum measured response RPS values. This allows sufficient headroom on each node to ensure service continuity in the case of an instance failure.

Table 4 below shows the effective hourly cost of each instance type for a three-node setup based on the responses per second necessary for the deployment. It also highlights the range of RPS achieved for a three-node deployment for various EC2 instances.

For example, to achieve up to 40,000 RPS with three instances operating at 66% utilization ratios, we look at each instance's RPS values at 66% from table 6. For example, three M6g.large instances can achieve 40,000 RPS (22,543 RPS *3). Similarly, three M5.large instances can achieve this same level of aggregate performance, but do so at a higher cost. Based on these RPS requirements and the $/hour values for each instance type, we are able to provide optimal cost-performance guidance.

Table 3 also illustrates an average of 45% performance gain for M6g instance over M5 across multiple instance sizes. When JWT authentication is added, there's an expected drop in overall RPS served by each instance type, but the performance gains still remain significant for M6g over M5.

Reverse Proxy 1KB - Performance results for M6g vs M5 per instance size

Instance Size	No Authentication – Max Request per Second			HS256 JWT Authentication – Max Request per Second		
	M6g	**M5**	**Performance Gains**	**M6g**	**M5**	**Performance Gains**
Large	45,836	29,211	**57%**	34,028	20,895	**63%**
XLarge	88,453	57,416	**54%**	64,634	43,591	**48%**
2XLarge	187,244	125,346	**49%**	128,589	87,691	**47%**
4XLarge	359,123	312,075	**15%**	326,608	207,418	**57%**
Avg Gains			**44%**			**54%**

Table 3: NGINX Reverse Proxy Performance per Instance size between M6g and M5

Table 5 shows three EC2 M6g.large instances cost $0.23/hour compared to $0.29/hour for three EC2 M5.large instances based on published on-demand EC2 pricing[3] as of August 2020. In the table below, the green cells identify the least expensive solution that meets the required performance.

Table 4: NGINX Reverse Proxy Price & Performance per Instance type

		Redundancy: 3 Max util: 66%	
Total RPS	RPS/instance	M6g - $/hr	M5 - $/hr
250	83	0.23	0.29
500	167	0.23	0.29
1,000	333	0.23	0.29
2,000	667	0.23	0.29
4,000	1,333	0.23	0.29
8,000	2,667	0.23	0.29
16,000	5,333	0.23	0.29
24,000	8,000	0.23	0.29
32,000	10,667	0.23	0.29
40,000	13,333	0.23	0.29
80,000	26,667	0.46	0.58
160,000	53,333	0.92	2.3
320,000	106,667	1.85	2.3

Table 5 shows instance sizes, maximum RPS, and RPS values at 66% utilization ratios per instance type.

Table 5: Maximum Reverse Proxy RPS values per Instance type based on HS356 Authentication

Reverse Proxy 1KB				
Instance	vCPU	maximum RPS	RPS at 66% utilization	$/hour
M6g.Large	2	34,028	22,458	$ 0.077
M6g.XLarge	4	64,634	42,658	$ 0.154
M6g.2XLarge	8	128,589	84,869	$ 0.308
M6g.4XLarge	16	326,608	215,562	$ 0.616
M5.Large	2	20,895	13,790	$ 0.096
M5.XLarge	4	43,591	28,770	$ 0.192
M5.2XLarge	8	87,691	57,876	$ 0.384
M5.4XLarge	16	207,418	136,896	$ 0.768

Test Scenario #2: NGINX API Gateway

As the leading high-performance, lightweight reverse proxy and load balancer, NGINX Plus has the advanced HTTPS processing capabilities needed for handling API traffic. The NGINX APIGW can address multiple use cases in an efficient and scalable manner. One advantage of using NGINX Plus as an APIGW is that it can perform that role while simultaneously acting as an RP, load balancer, and web server for existing HTTPS traffic. If NGINX Plus is already part of your application delivery stack, then it is generally unnecessary to deploy a separate APIGW. However, some of the default behavior expected of an APIGW differs from that expected for browser based traffic. For that reason, and for our testing purposes, we separate the APIGW configuration.

An APIGW takes all API calls from clients, then routes them to the appropriate microservice with request routing, composition, and protocol translation. It handles a request by invoking multiple microservices and aggregating the results, to determine the best path for that request. It can translate between web protocols and web unfriendly protocols that are used internally.

The test setup is similar to the RP example, with additional application level monitoring to make sure the request gets to the right server. The file size tested is 1KB.

The table below demonstrates the cost effectiveness of Amazon EC2 M6g instances when configured as an NGINX APIGW. The performance results and cost benefits are similar to the NGINX Plus RP with Amazon M6g instances providing 45% average performance gains and 20% cost-savings compared to EC2 M5 instances depending on the RPS range and instance size selected using three redundant instances.

Table 6: NGINX Plus API Gateway Performance per Instance size between M6g and M5

API Gateway 1KB - Performance results for M6g vs M5 per instance size						
	No Authentication – Max Request per Second			HS256 JWT Authentication – Max Request per Second		
Instance Size	**M6g**	**M5**	**Performance Gains**	**M6g**	**M5**	**Performance Gains**
Large	44,460	28,571	56%	32,859	20,400	61%
XLarge	84,752	55,997	51%	62,691	42,432	48%
2XLarge	178,815	120,500	48%	124,802	85,194	46%
4XLarge	359,855	300,313	20%	322,483	200,821	61%
Avg Gains			**44%**			**54%**

Table 7 : NGINX Plus API Gateway Price & Performance per Instance type based on HS356 Authentication

		Redundancy: 3 Max util: 66%	
Total RPS	RPS/instance	M6g - $/hr	M5 - $/hr
250	83	0.23	0.29
500	167	0.23	0.29
1,000	333	0.23	0.29
2,000	667	0.23	0.29
4,000	1,333	0.23	0.29
8,000	2,667	0.23	0.29
16,000	5,333	0.23	0.29
24,000	8,000	0.23	0.29
32,000	10,667	0.23	0.29
40,000	13,333	0.23	0.29
80,000	26,667	0.46	1.15
160,000	53,333	0.92	2.3
320,000	106,667	1.85	2.3

Below is a table of the results of the APIGW testing that shows the max RPS values and RPS values at 66% utilization ratios for each instance tested.

Table 8: Maximum API Gateway RPS values per Instance type based on HS356 Authentication

API Gateway 1KB				
Instance	vCPU	maximum RPS	RPS at 66% utilization	$/hour
M6g.XLarge	2	32,859	21,687	$ 0.077
M6g.XLarge	4	62,691	41,376	$ 0.154
M6g.2XLarge	8	124,802	82,370	$ 0.308
M6g.4XLarge	16	322,483	212,839	$ 0.616
M5.Large	2	20,400	13,464	$ 0.096
M5.XLarge	4	42,432	28,005	$ 0.192
M5.2XLarge	8	85,194	56,228	$ 0.384
M5.4XLarge	16	200,821	132,542	$ 0.768

Background

This section provides an overview of technologies and solutions offered by NGINX (part of F5), Arm, and Amazon Web Services.

NGINX

NGINX Plus is a lightweight, flexible, portable, and all-in-one software load balancer, reverse proxy, web server, content cache, and APIGW. By replacing a number of single-function point solutions with NGINX Plus, you can modernize and simplify your application architecture, reducing costs without compromising performance or functionality.

Arm Neoverse

Arm advanced, energy-efficient processor designs have enabled intelligent computing in more than 160 billion chips and our technologies now securely power products from the sensor to the smartphone and the supercomputer. The Arm ecosystem has been very strong in markets such as mobile, smart IoT, and infrastructure. From cellular base stations to routers and servers, there are more Arm processors shipping into infrastructure than any other architecture with nearly 30%-unit share, and growing.

The Arm Neoverse platform is specifically designed from the ground up for infrastructure with a roadmap committed to delivering more than 30% higher performance per generation. Arm Neoverse-powered products enable a diverse set of high-performance, secure and scalable solutions required for the infrastructure foundation in a world of trillion intelligent devices.

For more information on Arm Neoverse family of products, please visit https://www.arm.com/solutions/infrastructure

Amazon EC2 M6g Instances

Amazon EC2 M6g instances are powered by AWS Graviton2 processors that feature 64-bit Arm Neoverse cores and custom silicon designed by AWS. They deliver up to 54% better price performance over current generation M5 instances and offer a balance of compute, memory, and networking resources for a broad set of workloads. They are the best choice for applications built on open-source software such as application servers, microservices, gaming servers, mid-size data stores, and caching fleets. Developers can also use these instances to build Arm-based applications natively in the cloud, eliminating the need for cross-compilation and emulation,

and improving time to market. M6g instances are available with local NVMe-based SSD block-level storage option (M6gd) for applications that need access to high-speed, low latency local storage.

For more information, please visit https://aws.amazon.com/ec2/instance-types/m6/

Appendix A – Reverse Proxy/API Gateway Configs

Type	Size	Ubuntu 18.04 AMI	Kernel	NGINX	OpenSSL
M6g	Large, XLarge, 2XLarge, 4XLarge	ami-0400a1104d5b9caa1	4.15.0	nginx/1.17.6 (nginx-plus-r20)	1.1.1
M5	Large, XLarge, 2XLarge, 4XLarge	ami-07ebfd5b3428b6f4d	4.15.0	nginx/1.17.6 (nginx-plus-r20)	1.1.1

Appendix B –
Load Generator & Upstreams Config

	Instance Type	Ubuntu 18.04 AMI	Kernel	Installed App	OpenSSL
Upstream (File server)	M5.16XLarge	ami-07ebfd5b3428b6f4d	4.15.0	nginx/1.17.6	1.1.1
Client (Load Generator)	M5.16XLarge	ami-07ebfd5b3428b6f4d	4.15.0	Wrk2 Commit 44a94c17d8e6a0bac8559b53da76848e430cb7a7*	1.1.1

*The wrk2 project does not have a clear versioning scheme, so we reference a commit hash.

Appendix C – Other Networking Configuration Notes

Linux Networking Stack

Our automation system opens up the Linux network stack. The parameters that are changed by the automation system should not impact the outcome of the test cases presented in this paper. However, in the interest of full disclosure, we list the parameters below along with commands that can be used to change them. A great resource for understanding the parameters listed below is to run “man tcp” and “man listen” in a shell.

Below are the commands used to set these parameters:

```
# To view the default value before writing, remove the assignment.
For example
# to view net.ipv4.ip, run “sysctl net.ipv4.ip_local_port_range”

sysctl net.ipv4.ip_local_port_range=”1024 65535”
sysctl net.ipv4.tcp_max_syn_backlog=65535
sysctl net.core.rmem_max=8388607
sysctl net.core.wmem_max=8388607
sysctl net.ipv4.tcp_rmem=”4096 8388607 8388607”
sysctl net.ipv4.tcp_wmem=”4096 8388607 8388607”
sysctl net.core.somaxconn=65535
sysctl net.ipv4.tcp_autocorking=0
```

Appendix D – NGINX Configurations

Top Level Default nginx.conf (/etc/nginx/nginx.conf) for RP/APIGW and Upstreams

```
user www-data;
worker_processes auto;
worker_rlimit_nofile 1000000;
pid /run/nginx.pid;

events {
    worker_connections 1024;
    accept_mutex off;
    multi_accept off;
}
http {
    ##
    # Basic Settings
    ##
    sendfile on;
    tcp_nopush on;
    tcp_nodelay on;
    keepalive_timeout 75;
    keepalive_requests 1000000000;
    types_hash_max_size 2048;

    include /etc/nginx/mime.types;
    default_type application/octet-stream;

    ##
    # Logging Settings
    ##
    access_log off;

    # error_log off will not turn off error logs. Error logs will
redirect to /usr/share/nginx/off
    # The below comes the closest to actually turning off error logs.
    error_log /dev/null crit;

    ##
    # Virtual Host Configs
    ##
    include /etc/nginx/conf.d/*.conf;
    include /etc/nginx/sites-enabled/*;
}
```

The following are notes on the configuration shown above:

- **sendfile, tcp_nopush, and tcp_nodelay** are common optimizations for NGINX. These optimizations reduce context switching and improve the flow of packets through the Linux network stack.
- **worker_rlimit_nofile** is set to a large number to avoid too many open file errors.
- **worker_connections** - the number of concurrent worker connections is doubled to achieve higher RPS values.
- **access_log off** - logging is disabled because it can affect the consistency of test results. Please evaluate logging requirements for production environments.
- **keepalive_requests** increase the number of requests that can be made over a single connection to reduce the overhead of establishing and destroying connections.

Upstream Servers default.conf (/etc/nginx/conf.d/default.conf)

```
# HTTPS file server
server {
    listen 443 ssl reuseport backlog=65535;
    root /usr/share/nginx/html;
    index index.html index.htm;
    server_name $hostname;

    ssl on;
    ssl_certificate /etc/nginx/ssl/ecdsa.crt;
    ssl_certificate_key /etc/nginx/ssl/ecdsa.key;
    ssl_ciphers “ECDHE-ECDSA-AES256-GCM-SHA384”;

    location / {
        limit_except GET {
            deny all;
        }
        try_files $uri $uri/ =404;
    }
}
```

The follow are notes on the configuration shown above:

- TLS/SSL is configured to use ECDHE for key exchange, ECDSA for authentication, AES256-GCM for bulk encryption, and SHA384 for message authentication.
- We set the listen directive backlog to 65535, this is the same value that we set the **net.core.somaxconn** Linux networking stack parameter.

RP/APIGW default.conf (/etc/nginx/conf.d/default.conf)

```
# Upstreams for https
upstream ssl_file_server_com {
    server 192.168.97.128:443;
    server 192.168.97.31:443;
    keepalive 1024;
}

# JWT Authentication configs
auth_jwt “Performance Test API”;
auth_jwt_key_file /etc/nginx/jwk/server_jwt_keys.jwk;

# HTTPS reverse proxy and API Gateway
server {
    listen 443 ssl reuseport backlog=65535;
    root /usr/share/nginx/html;
    index index.html index.htm;
    server_name $hostname;

    ssl on;
    ssl_certificate /etc/nginx/ssl/ecdsa.crt;
    ssl_certificate_key /etc/nginx/ssl/ecdsa.key;
    ssl_ciphers “ECDHE-ECDSA-AES256-GCM-SHA384”;

    # API Gateway Path
    location ~ ^/api_old/.*$ {
        limit_except GET {
            deny all;
        }
        rewrite ^/api_old/(.*)$ /api_new/$1 last;
    }
    location /api_new {
        internal;
        proxy_pass https://ssl_file_server_com;
        proxy_http_version 1.1;
        proxy_set_header Connection “”;
    }

    # Reverse Proxy Path
    location / {
        limit_except GET {
            deny all;
        }
        proxy_pass https://ssl_file_server_com;
        proxy_http_version 1.1;
        proxy_set_header Connection “”;
    }
}
```

The follow are notes on the configuration shown above:

- The upstream context shows that we are load balancing between six upstreams.
- The directive **auth_jwt_key_file** points to the JWK file that contains secrets and public keys for verifying JWT token signatures. Having this directive present enables JWT verification. To test without JWT verification, remove this line and the one above it (**auth_jwt**), at which point all requests are forwarded to the upstreams without requiring a token.
- This config can be used to test both RP and APIGW use cases. The first two location blocks represents the APIGW path. Here we see that if the uri contains “/api_old/” in it’s base, then it will get rewritten with “/api_new/”. This simulates an APIGW translation. The last location block represents the RP path which forwards request to the upstreams without a rewrite.

Appendix E – TLS Keys & JWT Secrets/Keys

TLS Keys

All tests were done with TLS (HTTPS) enabled. As noted in the NGINX configs above, we used a cipher suite of ECDHE-ECDSA-AES256-GCM-SHA384. This means you need to generate ECDSA keys to establish the TLS session. In the testing above, our key was based on a P-384 curve.

JWT Secrets & Keys

A good resource for understanding JWT is www.jwt.io.

If testing HS256, then a secret must be generated. Our secret was 816 bits long after Base64url encoding. This length was selected so that even HS512 tokens can be tested if desired. This is because according to the rfc7518 spec, the secret has to be at least as large as the hash function used (816>512).

If testing RS256, then a public/private key pair must be generated. The keys we used were 2048 bits long (spec minimum). There are numerous resources that show how to create RSA keys on the internet.

Although we did not explore ES (eliptic curve) tokens, it is also possible to test these as well. Similarly to the RS algorithms, the ES algorithms require public/private key pairs to be created. There are numerous resources that show how to create ECDSA keys on the internet.

Appendix F – Commands

Creating the Files to Serve

Run the following commands on the upstreams to generate the files that will be served:

```
# Create 1kb file in RP use case directory
dd if=/dev/urandom of=/usr/share/nginx/html/1kb bs=1024 count=1
#Create 5kb file in RP use case directory
dd if=/dev/urandom of=/usr/share/nginx/html/5kb bs=1024 count=5
#Create 10kb file in RP use case directory
dd if=/dev/urandom of=/usr/share/nginx/html/10kb bs=1024 count=10

# Copy files into the APIGW use case directory
mkdir -p /usr/share/nginx/html/api_new
cp /usr/share/nginx/html/1kb /usr/share/nginx/html/api_new
cp /usr/share/nginx/html/5kb /usr/share/nginx/html/api_new
cp /usr/share/nginx/html/10kb /usr/share/nginx/html/api_new
```

Load Generator Commands

Build the wrk2 HTTP benchmark

```
sudo apt update
sudo apt install -y make gcc zlib1g-dev libssl-dev
git clone https://github.com/giltene/wrk2
cd wrk2
make
```

Reverse-Proxy Test Commands

Below is an example command for testing the RP with a 1kb file:

```
./wrk --rate 10000000000 -t 64 -c 640 -d 60s https://<rp_apigw_ip_
dns>/1kb
```

<rp_apigw_ip_dns> is the private IP or DNS name of the RP/APIGW instances that is to be tested.

API Gateway Commands

For the APIGW test, we use the same command as the RP test case, but with a different URL (one that will be rewritten). Below is an example command for testing with a 1kb file:

```
./wrk --rate 10000000000 -t 64 -c 640 -d 60s https://<rp_apigw_ip_
dns>/api_old/1kb
```

About Wrk2 Command Line Options

- We select an extremely high rate (--rate) to ensure we are measuring throughput.
- The number of threads (-t) is set to 64 which is the number of vCPUs on the Load Generator (m5.16xlarge).
- The number of connections (-c) is set to 10 times the number of threads which is 640. We found this produces the highest throughput.
- The test duration (-d) is 60 seconds. We found this to be a sufficient amount of time to get repeatable results.

References

1. NetCraft web server survey – May 2020
2. Quick start guide for NGINX Plus on AWS
3. AWS EC2 Instance Pricing
4. Nginx basic tuning blog