White Paper

NGINX is now part of F5

Cost Optimize NGINX Plus with Amazon EC2 A1

August, 2019

Index

Executive Summary

Amazon Web Services (AWS) introduced the Amazon EC2 A1 instance at AWS re:Invent 2018. These instances use the AWS Nitro-system and are the first instances powered by the AWS Graviton Processor featuring 64-bit Arm Neoverse cores and custom silicon designed by AWS.

Amazon EC2 A1 instances deliver up to 45% cost savings for Arm-native and modern scale-out applications such as web servers, containerized microservices, caching fleets, and distributed data stores that are supported by the extensive Arm ecosystem. These instances also appeal to developers, enthusiasts, and educators across the Arm community as they provide quick, easy and cost effective access to the Arm architecture using familiar EC2 cloud interfaces.

NGINX is one of the most popular scale-out web applications, and is well suited to running on Amazon EC2 A1 instances. NGINX Plus is based on NGINX Open Source, which is the #1 web server at the world's busiest 1,000, 10,000, and 100,000 websites, according to W3Techs. The commercially supported version NGINX Plus for Arm is readily available on the AWS Marketplace, including a free trial, making it very easy to get up and running on A1 instances.

In this document, we showcase the cost and performance benefits of deploying NGINX Plus application delivery and web services on Amazon EC2 A1 instances. In our performance testing, we focused on NGINX configured as a reverse proxy server and as an API gateway. The host infrastructure is configured in a redundant manner that replicates a typical production deployment, and then load tested across a range of total requests per second (RPS) values using a network traffic generator.

Based on our analysis, when configured as a reverse proxy or an API gateway, and serving up to 25,000 requests per second, the Amazon EC2 A1 instances deliver up to **40% cost savings** versus other EC2 configurations. This is great news for cost-concious NGINX customers requiring up to 25,000 RPS from a redundant, scale-out configuration reflecting a very common real world NGINX customer scenario.

The following table highlights NGINX reverse proxy price/performance for a 3-node deployment across multiple EC2 instance types and RPS values. The maximum node utilization was capped at 66% to accommodate a potential node failure scenario.
For instance, to achieve up to 25,000 RPS, the cost for three EC2 a1.large instances is $0.153/hour compared to $0.255/hour for three EC2 c5.large instances, thereby providing cost savings of 40%.

		Redundancy: 3 Max util: 66%			
Total rps	**RPS/ instance**	**a1 - $/hr**	**c5 - $/hr**	**m5 - $/hr**	**m5a -$/hr**
1,000	333	0.153	0.255	0.288	0.258
2,000	667	0.153	0.255	0.288	0.258
4,000	1,333	0.153	0.255	0.288	0.258
10,000	3,333	0.153	0.255	0.288	0.258
15,000	5,000	0.153	0.255	0.288	0.258
20,000	6,667	0.153	0.255	0.288	0.258
25,000	8,333	0.153	0.288	0.288	0.2505
35,000	11,667	0.306	0.255	0.288	0.258
50,000	16,667	0.306	0.255	0.288	0.516
75,000	25,000	0.612	0.51	0.576	0.516
100,000	33,333	0.612	0.51	0.576	1.032
125,000	41,667	0.612	0.51	0.576	1.032
175,000	58,333	1.224	1.02	1.152	1.032
200,000	66,667	1.224	1.02	1.152	2.064

Table 1: NGINX Reverse Proxy Price and Performance per Instance type

Similarly, to serve 50,000 total RPS, three (3) a1.xlarge instances with $/hour value of $0.306 is required compared to three (3) c5.large instances with $/hour value of $0.255. At this performance point c5 instances are upto 15% more cost effective. For details, see Testing Results.

While in our results, we provide cost-performance benefits by showcasing RPS values served by various instance sizes, in real world deployments, customers can further right size their instances by selecting different instance sizes within a given instance family. This will provide more performance granularity and varying benefits based on the specific deployment. Also, these cost comparisons are based on on-demand pricing for these instances and results will vary for Reserved and Spot instances.

NGINX on AWS Architecture

NGINX Plus in conjunction with the Elastic Load Balancing (ELB) offerings from AWS allow you to build a scalable and resilient application stack. While the AWS Application Load Balancer (ALB) provides some layer 7 load balancing support, we chose to use the NGINX Plus load-balancer to allow for more complex routing requirements. Combining NGINX Plus with an AWS Network Load Balancer (NLB) allows you to leverage high availability and scale NGINX horizontally. Combined with the powerful AWS autoscaling group integrations you can autoscale your NGINX load balancing tier, and you can easily autoscale your backend applications utilizing the NGINX asg-sync package. When combined with the AWS Route53 Global Server Load Balancing (GSLB) solution, you can build a geographically diverse, highly available environment. You can easily build an example of this architecture by using the NGINX Plus on AWS quickstart guide.

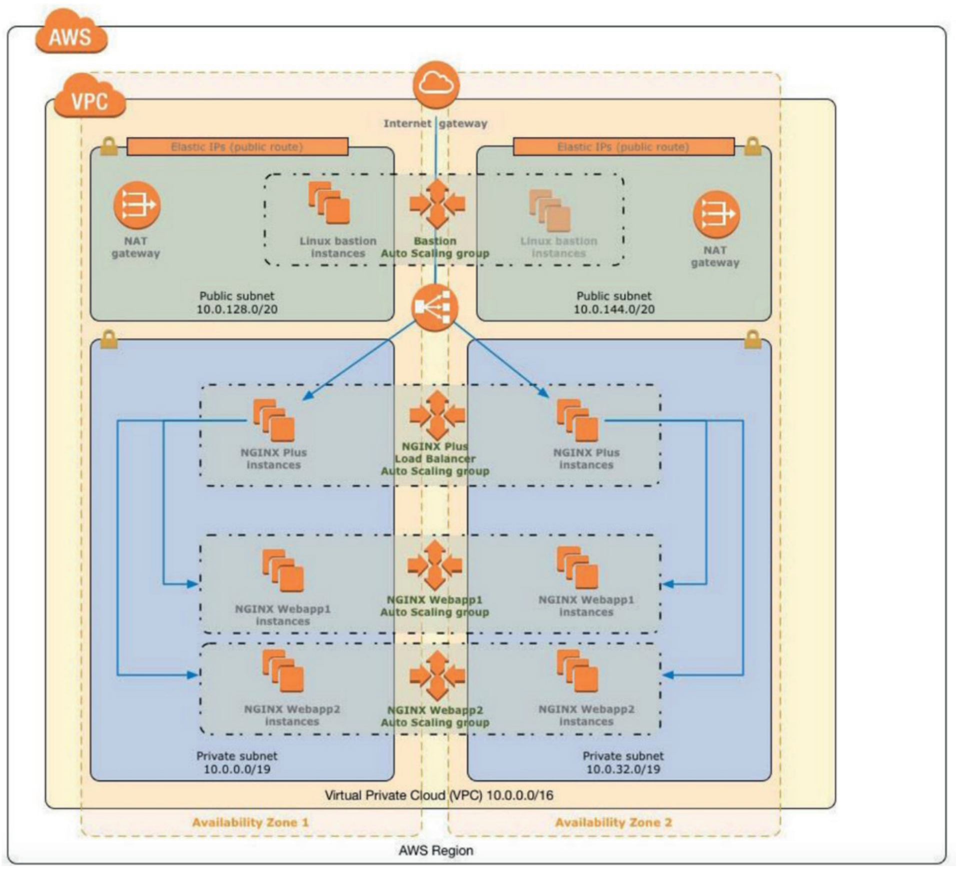

AWS
VPC
Internet gateway
Elastic IPs (public route)
Elastic IPs (public route)
NAT gateway
Linux bastion instances
Bastion Auto Scaling group
Linux bastion instances
NAT gateway
Public subnet 10.0.128.0/20
Public subnet 10.0.144.0/20
NGINX Plus instances
NGINX Plus Load Balancer Auto Scaling group
NGINX Plus instances
NGINX Webapp1 instances
NGINX Webapp1 Auto Scaling group
NGINX Webapp1 instances
NGINX Webapp2 instances
NGINX Webapp2 Auto Scaling group
NGINX Webapp2 instances
Private subnet 10.0.0.0/19
Private subnet 10.0.32.0/19
Virtual Private Cloud (VPC) 10.0.0.0/16
Availability Zone 1
Availability Zone 2
AWS Region

Figure 1: Arcitecture Diagram: NGINX Plus with AWS Network load balancer (NLB) and auto-scale group

Testing Results

This section provides test setup details and performance results for the NGINX Plus reverse proxy and API gateway features. The performance of these two functions often determine the overall performance of many of the NGINX deployments using AWS today.

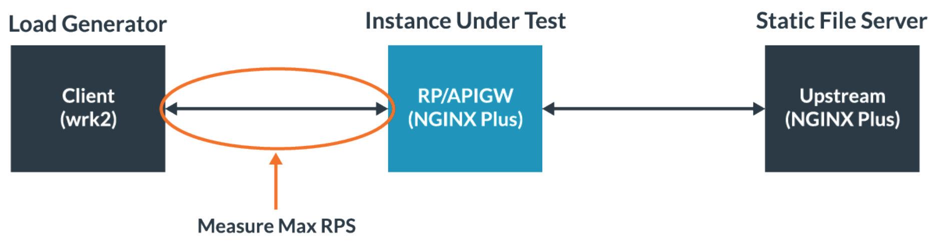
Test Setup

The test setup is designed to measure the max HTTPS Requests Per Second (RPS) of a Reverse Proxy (RP) or an API Gateway (APIGW). In the figure below, the client makes requests for a resource to an instance that is capable of testing both RP & APIGW use cases. In addition, there is an upstream web server that will serve static files.

For HTTP/HTTPS traffic generation, an open source application wrk2 is used which is built as a modern HTTP benchmarking tool capable of generating significant load for a single multi-core CPU. It combines a multithreaded design with scalable event notification systems, allowing the user to create scripts to generate requests, process the responses, and report the test outcomes very easily.

Following describes the test workflow:

1. Client sends requests for static files to the RP/APIGW
2. Upon receipt, RP/APIGW checks if the requested Uniform Resource Identifier (URI) should be rewritten
3. 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 the upstream server instance
5. The upstream server will respond to the RP/APIGW with the requested resource (static file in our case)
6. The RP/APIGW will send the resource back to the client where the request originated. The client running wrk2 produces the RPS metric.



As shown in the above diagram, a single instance of wrk2 was used to generate traffic required to stress test the two common NGINX use cases of reverse proxy and API gateway on the various EC2 instances. These are single instances tested with an NGINX web server instance configured in the back end to respond to requests generated by the wrk traffic generator. This ensures that a full bi-directional traffic flow is being tested for the respective NGINX features. By stress testing these single instances, we're able to achieve the maximum requests per second (RPS) that each instance can support before performance degradation occurs.

The following table shows the various instances tested:

AWS Instance	Sizes	Number of vCPUs
a1	large xlarge 2xlarge 4xlarge	2 4 8 16
m5a	large xlarge 2xlarge 4xlarge	2 4 8 16
c5	large xlarge 2xlarge 4xlarge	2 4 8 16
m5	large xlarge 2xlarge 4xlarge	2 4 8 16

Find more information on AWS EC2 instances here.

Use Case #1: NGINX Plus reverse proxy

A **proxy server** is a go between or intermediary server that forwards requests for content from multiple clients to different servers across the Internet. A **reverse proxy server** is a type of proxy server that typically sits behind the firewall in a private network and directs client requests to the appropriate back end server. A reverse proxy provides an additional level of abstraction and control to ensure the smooth flow of network traffic between clients and servers. Common uses for a reverse proxy server include load balancing, web acceleration and security.

In this setup, the NGINX Plus reverse proxy server is sending HTTP requests generated by the wrk2 traffic generator to the back end NGINX web server and returning the response back to the client that requested it. The HTTP request in our tests were crafted to return a "404 page not found message" as a proxy for a small-size payload messages returned to the requester.

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

In real world, these instances are configured with N+1 configuration where N stands for the number of instances required to meet the performance requirements, typically a minimum of 2 to achieve high availability. In addition, there is at least an instance reserved to be available in an event of an instance failure. We configured three (3) nodes to achieve N+1 configuration with each node operating at a maximum of 66% utilization from it's maximum response RPS values tested in order to handle an instance failure by spreading the load without causing any disruption in service continuity to end customers.

The table below shows the effective hourly cost of each instance type for a three (3) 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 25,000 RPS with three (3) instances operating at 66% utilization ratios, we look at each instance's RPS values at 66% from table TABLE 5. Three (3) a1.large instances can deliver the performance required to achieve 25,000 RPS (10,618.74 RPS *3). Similarly, three (3) c5.large instances can achieve this level of aggregate performance for this configuration, but 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. For example, three EC2 a1.large instances cost $0.153/hour compared to $0.255/hour for three EC2 c5.large instances.

In the table below, the green cells identify the least expensive solution that meets the required performance. The light green, yellow and red cells cover the $/hr ranges for each instance type to meet the required RPS values and redundancy.

		Redundancy: 3 Max util: 66%			
Total rps	RPS/ instance	a1 - $/hr	c5 - $/hr	m5 - $/hr	m5a -$/hr
1,000	333	0.153	0.255	0.288	0.258
2,000	667	0.153	0.255	0.288	0.258
4,000	1,333	0.153	0.255	0.288	0.258
10,000	3,333	0.153	0.255	0.288	0.258
15,000	5,000	0.153	0.255	0.288	0.258
20,000	6,667	0.153	0.255	0.288	0.258
25,000	8,333	0.255	0.288	0.288	0.2505
35,000	11,667	0.306	0.255	0.288	0.258
50,000	16,667	0.306	0.255	0.288	0.516
75,000	25,000	0.612	0.51	0.576	0.516
100,000	33,333	0.612	0.51	0.576	1.032
125,000	41,667	0.612	0.51	0.576	1.032
175,000	58,333	1.224	1.02	1.152	1.032
200,000	66,667	1.224	1.02	1.152	2.064

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

The table 5 below shows how instance sizes and its maximum RPS and RPS values at 66% utilization ratios achieved per instance type:

	Reverse proxy			
Instance	vCPU	maximum RPS	RPS at 66% utilization	$/hour
a1.large	2	16089	10618.74	$ 0.051
a1.xlarge	4	36157	23863.62	$ 0.102
a1.2xlarge	8	76004	50162.64	$ 0.204
a1.4xlarge	16	169037	111564.42	$ 0.408
c5.large	2	31075	20509.5	$ 0.085
c5.xlarge	4	75955	50130.3	$ 0.170
c5.2xlarge	8	188886	124664.76	$ 0.340
c5.4xlarge	16	450308	297203.28	$ 0.680
m5.large	2	33774	22290.84	$ 0.096
m5.xlarge	4	74151	48939.66	$ 0.192
m5.2xlarge	8	187147	123516.36	$ 0.384
m5.4xlarge	16	437523	288765.18	$ 0.768
m5a.large	2	25006	16503.96	$ 0.086
m5a.xlarge	4	47615	31425.9	$ 0.172
m5a.2xlarge	8	97045	64049.7	$ 0.344
m5a.4xlarge	16	227763	150323.58	$ 0.688

Table 5: Maximum Reverse Proxy RPS values per Instance type

Use Case #2: NGINX API Gateway

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

An API gateway 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 reverse proxy example, with additional application level monitoring to make sure the request gets to the right server. The file size tested is 1kb to mimic the most common HTTP requested file size.

The table below demonstrates how cost-effective Amazon EC2 A1 instances are when configured as an NGINX API gateway. The performance results and cost benefits are similar to the NGINX reverse proxy with Amazon A1 instances providing upto 40% cost-savings compared to higher performance EC2 instances for up to 25,000 RPS when delivered using three redundant instances.

		Redundancy: 3 Max util: 66%			
Total rps	**RPS/ instance**	**a1 - $/hr**	**c5 - $/hr**	**m5 - $/hr**	**m5a -$/hr**
1,000	333	0.153	0.255	0.288	0.258
2,000	667	0.153	0.255	0.288	0.258
4,000	1,333	0.153	0.255	0.288	0.258
10,000	3,333	0.153	0.255	0.288	0.258
15,000	5,000	0.153	0.255	0.288	0.258
20,000	6,667	0.153	0.255	0.288	0.258
25,000	8,333	0.153	0.255	0.288	0.258
35,000	11,667	0.306	0.255	0.288	0.258
50,000	16,667	0.306	0.255	0.288	0.516
75,000	25,000	0.612	0.51	0.576	0.516
100,000	33,333	0.612	0.51	0.576	1.032
125,000	41,667	0.612	1.02	1.152	1.032
150,000	50,000	1.224	1.02	1.152	1.032
175,000	58,333	1.224	1.02	1.152	2.064
200,000	66,667	1.224	1.02	1.152	2.064

Table 6: NGINX API Gateway Price & Performance per Instance type

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

	API Gateway 1kb HTTPS ECDHE-ECDSA-AES256-GCM-SHA384			
Instance	**vCPU**	**maximum RPS**	**RPS at 66% utilization**	**$/hour**
a1.large	2	14287	9429.42	$ 0.051
a1.xlarge	4	32140	21212.4	$ 0.102
a1.2xlarge	8	63761	42082.26	$ 0.204
a1.4xlarge	16	126353	83392.98	$ 0.408
c5.large	2	25722	16976.52	$ 0.085
c5.xlarge	4	59530	39289.8	$ 0.170
c5.2xlarge	8	133562	88150.92	$ 0.340
c5.4xlarge	16	384364	253680.24	$ 0.680
m5.large	2	31041	20487.06	$ 0.096
m5.xlarge	4	56232	37113.12	$ 0.192
m5.2xlarge	8	141565	93432.9	$ 0.384
m5.4xlarge	16	366050	241593	$ 0.768
m5a.large	2	20209	13337.94	$ 0.086
m5a.xlarge	4	40456	26700.96	$ 0.172
m5a.2xlarge	8	83224	54927.84	$ 0.344
m5a.4xlarge	16	182184	120241.44	$ 0.688

Table 7: Maximum API Gateway RPS values per Instance type

Conclusion

These performance results provide you with deployment guidelines for common NGINX configurations across a variety of Amazon EC2 instances and showcases the cost benefits of deploying Amazon EC2 A1 instances – demonstrating up to 40% cost savings for scale-out NGINX deployments. This document also provides a framework for you to deploy additional features and services on A1 instances. With NGINX Amazon Machine Images (AMIs) readily available for A1 instances, you can deploy NGINX Plus on AWS EC2 A1 with ease to achieve the best price-performance for your specific use case.

Background

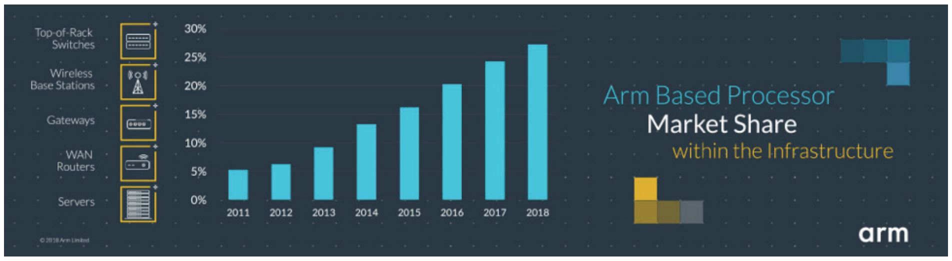
This section provides an overview of NGINX, Arm and Amazon EC2 A1 instances

NGINX

NGINX Plus is a lightweight, flexible, portable, and all-in-one software load balancer, WAF, reverse proxy, web server, content cache, and API gateway. 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 technology is at the heart of a computing and connectivity revolution that is transforming the way people live and businesses operate. Arm's advanced, energy-efficient processor designs have enabled intelligent computing in more than 145 billion chips and Arm technologies now securely power products from the sensor to the smartphone and the supercomputer. The Arm ecosystem has been very strong in markets like 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.



To further fuel innovation in the infrastructure space, in October 2018 Arm announced a dedicated infrastructure roadmap with Arm Neoverse-powered products enabling 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 here.

Amazon EC2 A1 Instances

Amazon EC2 A1 instances are the first EC2 instances powered by AWS Graviton Processors that feature 64-bit Arm Neoverse cores and custom silicon designed by AWS. These instances deliver up to 45% cost savings for scale-out and Arm-based applications such as the web servers, containerized microservices, caching fleets, and distributed data stores that are supported by the extensive Arm ecosystem.

For more information, please visit https://aws.amazon.com/ec2/instance-types/a1/.

Appendix A - Software versions used

	Linux Image	Kernel version	Application version	Open SSL
Arm A1 Instance	Ubuntu 18.04 (ami-01ac7d-9c1179d7b74)	4.15.0-1028-aws	nginx/1.15.10 (nginx-plus-r18)	Openssl: 1.1.0
x86 instances	Ubuntu 18.04 (ami-024a64a6685d05041)	4.15.0-1039-aws	nginx/1.15.10 (nginx-plus-r18)	Openssl: 1.1.0
Load Generator - M4.4xlarge	Ubuntu 18.04 (ami-024a64a6685d05041)	4.15.0-1039-aws	wrk2 (version 4.0)	Openssl: 1.1.0
Upstream File server - M4.4xlarge	Ubuntu 18.04 (ami-024a64a6685d05041)	4.15.0-1039-aws	nginx/1.15.10 (nginx-plus-r18)	Openssl: 1.1.0

Appendix B - AWS EC2 Instance AMI Configurations

Type	Size	Ubuntu 18.04 AMI	Kernel	Nginx	Open SSL
A1	large, xlarge, 2xlarge, 4xlarge	ami-01ac7d-9c1179d7b74	4.15.0-1028-aws	1.15.10 (nginx-plus-r18)	1.1.0
C5	large, xlarge, 2xlarge, 4xlarge	ami-024a64a6685d05041	4.15.0-1039-aws	1.15.10 (nginx-plus-r18)	1.1.0
M5	large, xlarge, 2xlarge, 4xlarge	ami-024a64a6685d05041	4.15.0-1039-aws	1.15.10 (nginx-plus-r18)	1.1.0
M5a	large, xlarge, 2xlarge, 4xlarge	ami-024a64a6685d05041	4.15.0-1039-aws	1.15.10 (nginx-plus-r18)	1.1.0

*All instances were setup in the us-east-1 (N. Virginia) region

Appendix C - Load Generator and Web Server Configuration

	Instance Type	Ubuntu 18.04 AMI	Kernel	Installed App	Open SSL
Upstream (File server)	M5.4xlarge	ami-024a64a6685d05041	4.15.0-1039-aws	1.15.10 (nginx-plus-r18)	1.1.0
Client (Load Generator)	M5.4xlarge	ami-024a64a6685d05041	4.15.0-1039-aws	Wrk2 4.0.0	1.1.0

Client and Upstream EC2 Instance Configuration

Additionally, we set **net.ipv4.tcp_tw_reuse** and **net.core.somaxconn** on the RP/APIGW and upstream instances with the following commands:

- **net.ipv4.tcp_tw_reuse** - increases the reusability of connections that are waiting to be closed
- **net.core.somaxconn** - is set to a much larger value, in this case 32768, from the default 128 so that more connections can be handled concurrently

```
sysctl net.ipv4.tcp_tw_reuse=1
sysctl net.core.somaxconn=32768
```

Appendix D - NGINX Configurations

Common Top Level Default nginx.conf for RP/APIGW and Upstream

```
user  nginx;
worker_processes  auto;
worker_rlimit_nofile 1000000;

error_log  /var/log/nginx/error.log crit;
pid        /var/run/nginx.pid;

events {
   worker_connections  1024;
}

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

   access_log     off;

   sendfile       on;
   tcp_nopush     on;
   tcp_nodelay    on;

   keepalive_timeout  65;
   keepalive_requests 1000000000;

   include /etc/nginx/conf.d/*.conf;
}
```

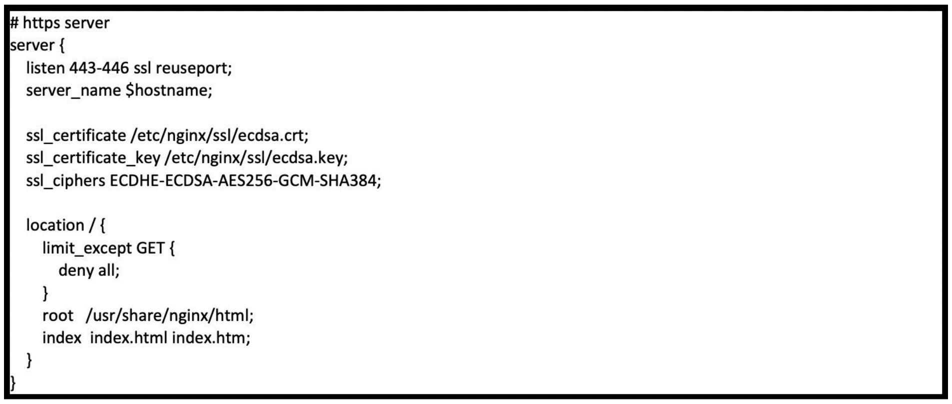
Following highlights the changes made to the nginx.conf file:

- **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 performanc and achieve consistency in
- **keepalive_requests** increase the number of requests that can be made over a single connection to reduce the overhead of establishing and destroying connections
- **gzip** removed from the default.conf file to remove the CPIU overhead associated with compressing response headers

Following highlights default nginx.conf file:

- **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.

The default.conf (stored in /etc/nginx/conf.d/) sets up a static file server for the upstream file server and is shown below:



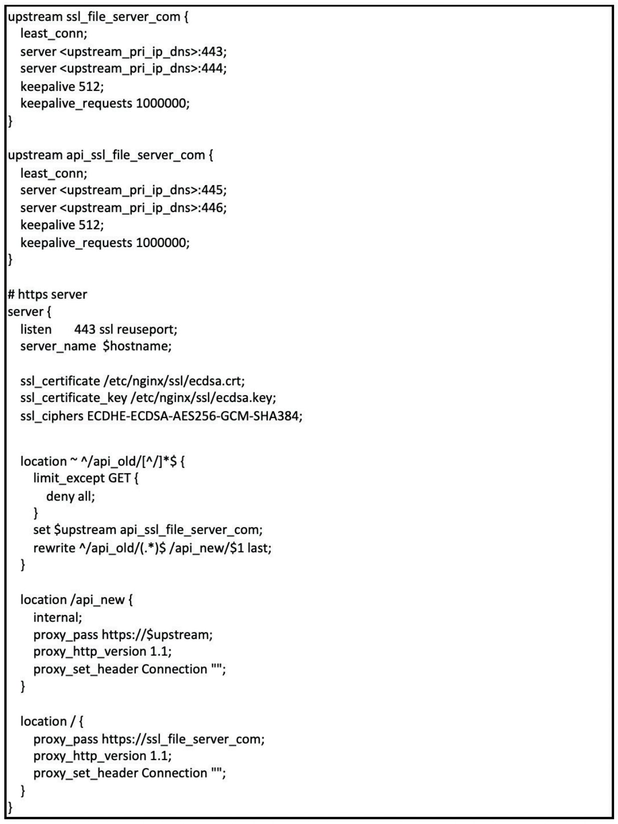
```
# https server
server {
    listen 443-446 ssl reuseport;
    server_name $hostname;

    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;
        }
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }
}
```

Upstream default.conf

- Server listens on ports 443-446 to simulate load balancing
- TLS/SSL is configured to use ECDSA for authentication, ECDHE for key exchange, AWSGCM-256 for encryption and SHA384 for message authentication
- In the location block we configure the server to reject all HTTP methods except for the GET method
- Upstream server is configured to serve files from /usr/share/nginx/html

RP/APIGW Nginx Configurations

The default.conf for RP/APIGW which is stored in /etc/nginx/conf.d/ is shown below



```
upstream ssl_file_server_com {
    least_conn;
    server <upstream_pri_ip_dns>:443;
    server <upstream_pri_ip_dns>:444;
    keepalive 512;
    keepalive_requests 1000000;
}

upstream api_ssl_file_server_com {
    least_conn;
    server <upstream_pri_ip_dns>:445;
    server <upstream_pri_ip_dns>:446;
    keepalive 512;
    keepalive_requests 1000000;
}

# https server
server {
    listen       443 ssl reuseport;
    server_name  $hostname;

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


    location ~ ^/api_old/[^/]*$ {
        limit_except GET {
            deny all;
        }
        set $upstream api_ssl_file_server_com;
        rewrite ^/api_old/(.*)$ /api_new/$1 last;
    }

    location /api_new {
        internal;
        proxy_pass https://$upstream;
        proxy_http_version 1.1;
        proxy_set_header Connection "";
    }

    location / {
        proxy_pass https://ssl_file_server_com;
        proxy_http_version 1.1;
        proxy_set_header Connection "";
    }
}
```

Following highlights the changes made to RP/APIGW default.conf:

- **upstream_pri_ip_dns** should be set to the private DNS/IP address of the upstream servers with ports 443 and 444 allowing load balancing
- **keepalive** reduces the overhead of establishing and destroying connections between the RP and the upstream server
- **proxy_http_version and proxy_set_header** enables the keepalive connections as referred in the Nginx basic tuning blog

Following highlights the RP/APIGW default.conf details:

- **upstream_ssl_file_server_com** represents the reverse proxy use case and establishes upstream server connection
- **upstream_api_ssl_file_server_com** represents the API Gateway use case
- **location** specifies various RP and APIGW URI checks with specific actions such as:
  - For /**api_old**/* set **$upstream** to api_ssl_file_server_com (APIGW) and URI gets rewritten
  - For /**api_new**/* set **$upstream** to api_ssl_file_server_com with proxy_pass directive instructing APIGQ to forward and load balance the request between ports 445 and 446 of the upstream server
  - Example URL for APIGQW use case would be https://<upstream_pri_ip_dns>/api_new/1kb
  - For reverse proxy use case, request is forwarded to the upstream labeled ssl_file_server_com forwarding the URL to upstream server without modification. An example URL for this would be https://<RP_APIGW_IP_DNS>/1kb

Appendix E - Commands

File Commands

```
# 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

```
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 the command we used to run a max RPS test. The URL below will return a 404 error which is useful for testing small payloads

```
./wrk --rate 1000000000 --latency -t 50 -c 1000 -d 60s https://<rp_apigw_ip_dns>/nothing
```

Here's an example for testing with a 1kb resource:

```
./wrk --rate 1000000000 --latency -t 50 -c 1000 -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.

Command line options:

- A rate of one billion RPS ensures that we are measuring max RPS
- The latency switch enables detailed information about latency during the test
- The number of threads is set to 50. This is because we originally started our testing on M5.8xlarge instances. M5.8xlarge instances have 32 vCPUs, so we selected a thread count that is a bit higher than the max vCPUs of the instance. Later, we realized we could achieve the max RPS with the smaller M5.4xlarge instance without changing the number of threads
- 1000 connections were found to produce higher results
- The test time is 60 seconds

API Gateway Commands

For the APIGW test, use the same command as the RP test case, but with a different URL (one that will not be rewritten). Below is an example command for testing with a 1kb resource:

```
./wrk --rate 1000000000 --latency -t 50 -c 1000 -d 60s https://<rp_apigw_ip_dns>/api_old/1kb
```

Here's an example for testing the APIGW with a very small payload (404 errors):

```
./wrk --rate 1000000000 --latency -t 50 -c 1000 -d 60s https://<rp_apigw_ip_dns>/api_old/nothing
```

ECDSA keys and Certificate Commands

Run the following commands to create the keys and certificate. These commands will need to be run on each instance that is running Nginx. This includes the upstream server. The last two commands will copy the key and certificate to the location specified in the Nginx configuration files.

ECDSA keys and Certificate Commands

Run the following commands to create the keys and certificate. These commands will need to be run on each instance that is running Nginx. This includes the upstream server. The last two commands will copy the key and certificate to the location specified in the Nginx configuration files.

```
openssl ecparam -out ecdsa.key -name prime256v1 -genkey
openssl req -new -sha256 -key ecdsa.key -out server.csr \
      -subj "/C=US/ST=Your State/L=Your Town/O=Your Org/CN=www.yoursite.net"
openssl x509 -req -sha256 -days 365 -in server.csr -signkey ecdsa.key -out ecdsa.crt
cp ecdsa.key /etc/nginx/ssl/ecdsa.key
cp ecdsa.crt /etc/nginx/ssl/ecdsa.crt
```

Note: If you see an error that states 'unable to write random state', try deleting ~/.rnd.

References

https://aws-quickstart.s3.amazonaws.com/quickstart-nginx-plus/doc/nginx-plus-on-the-aws-cloud.pdf

Nginx basic tuning blog

https://aws.amazon.com/ec2/pricing/