svLoad: An Automated Test-Driven Architecture for Load Testing in Cloud Systems

Jannatun Noor*, Md. Golam Hossain†, Muhammad Ahad Alam‡, Ashraf Uddin§,
Sriram Chellappan¶, A. B. M. Alim Al Islam‖
*‡‖Department of CSE, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh
†§IPvision Canada Inc., Dhaka, Bangladesh, ¶Department of CSE, University of South Florida, Tampa, USA
*mucse066@yahoo.com, {†mghhimu, §raselashraf21}@gmail.com, ‡1005118.maua@ugrad.cse.buet.ac.bd
¶sriramc@usf.edu, ‖alim_razi@cse.buet.ac.bd

***Abstract*—Nowadays, Internet-based technologies possess immense processing power, capacity, flexibility, and are gradually moving towards a service-oriented functionality in order to build new distributed storage systems in the cloud. Several distributed systems are currently running in different geographically located data centers for successful deployment of modern web and social services such as Facebook, Twitter, ringID, etc. Both cache and backend servers in such distributed systems must be functional and reliable for incoming workloads by means of efficient allocation of capacity along with proper configuration and tuning of multiple system resources. To address these challenges, in this paper, we propose a test-driven automated architecture for load testing, named as 'svLoad' to compare the performance of cache and backend servers. Here, we designed test cases considering diversified real scenarios such as different protocol types, same or different URLs, with or without load, cache hit or miss, etc. using tools namely JMeter, Ansible, and some custom utility bash scripts. To validate the efficacy of our proposed methodology, we conduct a set of experiments by running these test cases over a real private cloud development setup using two open source projects - Varnish as the cache server and OpenStack Swift as the backend server. Our focus is also to find out bottlenecks of Varnish and Swift by executing load requests, and then tune the system based on our load test analysis. After successfully tuning the Swift, Varnish, and network system, based on our test analysis, we were able to improve the response time by up to 80%.**

Keywords*—*Load Testing, Cloud, Test Case Metrics, Response Time, System and Network Tuning

I. Introduction

In this era of connected devices, the demand of storage systems are increasing exponentially [1]. Using various open-source projects [1], [2], several distributed private cloud storage systems are now emerging [3], [4]. At the same time, clients are increasingly demanding faster and easier access to data from these systems. In addition, system designers need to test the behavior of these distributed architectures under massive operational loads to designing architectures properly and flawlessly. Furthermore, service providers use cache(s) in front of backend servers for retrieving data faster from distributed private cloud systems. Hence, information about the time elapsed for retrieving data from cache or backend in different test scenarios is necessary to design a reliable system. Analyzing that information, service providers can find out numbers and appropriate locations of the cache and backend servers for achieving the best outcome. Apart from these, load testing is also needed to tune the parameters of the software, hardware, and network used in the system.

As of today, private and public cloud service providers design their own distributed storage systems using several data centers. Choosing best locations for deploying cache and backend cloud servers in data centers is one of the challenging task for most service providers. Here, time delays in object uploading and downloading are directly related to how the cache and backend servers are distributed. Furthermore, for successful deployment of distributed architectures including caches and clouds in production environments, proper load testing is mandatory. As such, several existing research studies focus on load testing tools and architectures based on performance and functional testing criteria. The study in [5] proposes an empirical testing by monitoring user experience and system health in a feedback loop between traffic shifts. Other studies [6]–[8] propose automated approaches to validate whether a performance test resembles the field workload or not. Unfortunately, these studies propose and analyze only real-time test cases without focusing on network and software tuning using the outcomes of the test analysis.

Furthermore, recent studies do not focus on finding a general load testing architecture for testing distributed systems that combine both cache and backend servers. Hence, in this paper, by means of a rigorous study we propose a load test architecture 'svLoad' that facilitates the process of finding out the best values of parameters based on real scenarios. We also locate the bottlenecks of OpenStack Swift and Varnish cache servers when they are operating with extensive load. In our study, we offer extensive load requests from some predefined clients based on our proposed test cases. Hence, the server will be busy on handling the requests. At the same time, we send concurrent download requests using bash scripts and observe percentage of success rates among the requests, and how much time they need for ending up with successful responses. Besides, we identify resource utilization bottlenecks in both system and network performances, and tune the system and networks parameters accordingly, and analyze the system behavior through subsequent load tests.

Based on our study, we make the following set of specific contributions in this paper:

- We propose 20 different test cases based on diversified real scenario covering different protocol types (HTTP

978-1-5386-4727-1/18/$31.00 ©2018 IEEE

or HTTPS), URL types (same or different URLs) , load types (with or without loads), and server types (backend, cache) for performing load test on cloud systems using several tools - JMeter [9], Ansible [10], and our custom bash scripts.

- We perform continuous rigorous load testing on two open source cloud systems namely Varnish [2] and Swift [1], and find out bottlenecks in the system and network that are worthy of tuning.
- Subsequently, we perform parameter tuning as per our findings of load testing. Here, first we come up with a comparison of response times for downloading files from cache and backend servers for each test case through rigorous experimentation. Then, we improve the response times and success rates of concurrent requests up to 80% and 90% respectively through our tuning in the Swift, Varnish, and network systems.

II. Literature Survey

The main goal of load testing is to identify the upper limit of systems in terms of performance of the database, hardware, network, etc. Hence, realistic test case based architectures for distributed cloud storage system is critical. Also, while, functional tests may ensure the general performance of a cloud, load tests ensure system reliablity and fault tolerance at even very demanding load requests. Load tests give developers confidence that the cloud is well sized. Hence, the importance of realistic and generalized load tests for cache and backend, today.

Furthermore, load tests for open source caches like Varnish [2], and cloud systems like OpenStack Swift [1] are needed for tuning system parameters for vendors who merge these two components for building large distributed cloud architectures. Recently, several works have appeared on load test tools, cloud evaluation criteria based on load test, performance testing of web applications, workload optimization, continuous validation, etc. in this realm. Here, we present a short summary of these works to movitate our new architecture 'svLoad' for testing load capabilities in cache and backend cloud servers.

A comparative performance study [11] among different testing tools shows Webload is better in terms of assessing response time and throughput, compared to tools like Neoload, LoadImpact, Loadster and LoadUI. The study in [12] presented important factors in cloud computing performance, and analyzed and evaluated cloud performance in various scenarios based on criteria, characteristics, and simulation. The study in [6] used the Load Runner testing tool to capture end-user business processes and created automated performance testing virtual scripts to organize, manage, and monitor load testing through running virtual users. Another study in [5] analyzed the behavior of individual systems and groups of systems to identify resource utilization bottlenecks to ensure Facebook's allocated capacity for servicing download requests via tuning.

In addition, studies like [13], [14] analyze system performance degradation or problems handling required system throughput. Studies in [15]–[18] presented late-cycle measurement-based and model-based approaches. Measurement-based approaches apply testing, diagnosis and tuning late in the development cycle. The study in [7] presented that performance analysts must continually validate whether their tests are reflective of the field or not. Such validation may be performed by comparing execution logs from the test and the field.

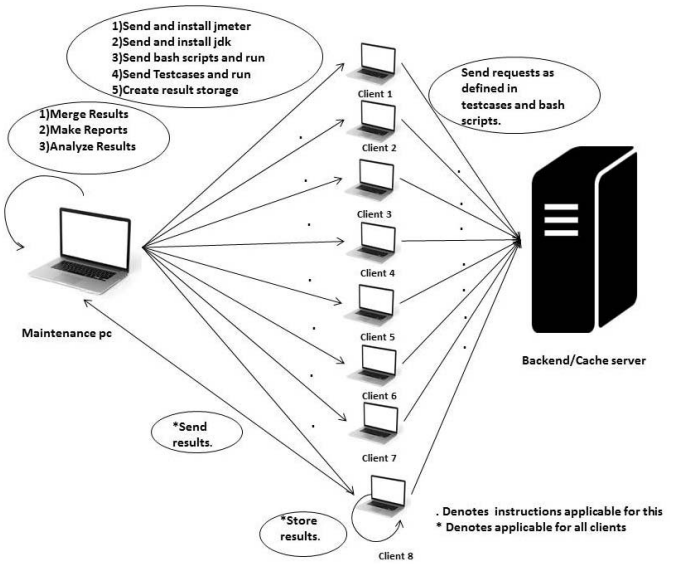


Fig. 1. Architectural overview of our proposed load test model

After going through all these studies, we can say that these studies on load tests address some real time test cases, and do focus on load test based on automated framework, but they did not cover the general load test scenarios based on real metrics. Furthermore, these studies barely concentrated on network and software tuning along with load tests to make the system best suited while using open source projects such as Swift, Varnish, etc. To the best of our knowledge, load test architectures that vary real metrics such as network protocol, URL type, load amount, and server type using JMeter, Ansible, and use case centric bash scripts as load test aids for load testing in distributed storage system are yet to be accomplished. This motivates our paper.

III. Proposed Methodology

In this paper, we propose a general load test architecture for distributed storage system. Fig. 1 presents the architectural overview of our proposed load test model. Our proposed operational methodology over this architecture comprises several key steps, which we present in the following subsections.

A. Load Test Planning

The main technique for measuring performance of servers is to give extensive concurrent requests to respective servers. For this, we need to run the load tests in regular basis as the results vary with software and system variable factors such

TABLE I. TEST CASE SCENARIOS FOR DIFFERENT METRICS

Test case ID	Protocol		URL		Load		Server		
	HTTP	HTTPS	Same	Diffe-rent	With out	With	Back end	Cache	
								Hit	Miss
TC0	X		X		X		X		
TC1	X		X			X	X		
TC2	X		X		X			X	
TC3	X		X			X		X	
TC4	X		X		X				X
TC5	X		X			X			X
TC6	X			X	X		X		
TC7	X			X		X	X		
TC8	X			X	X				X
TC9	X			X		X			X
TC10	X			X	X			X	
TC11	X			X		X		X	
TC12		X	X		X		X		
TC13		X	X			X	X		
TC14		X	X		X			X	
TC15		X	X			X		X	
TC16		X	X		X				X
TC17		X	X			X			X
TC18		X		X	X		X		
TC19		X		X		X	X		
TC20		X		X	X				X
TC21		X		X		X			X
TC22		X		X	X			X	
TC23		X		X		X		X	

as network bandwidth, CPU, memory usage, etc. Hence, we focus on automating the whole process with minimal effort. We give importance on several necessary questions as follows:

1) Which tools should be used for load testing purpose?
2) How many machines should be used?
3) What would be the required machine configurations?
4) How to automate the whole process?
5) What would be the most important metrics for designing the test cases?
6) What components of the hardware, software, and network systems would be the limiting factors of the performance?

In recent times, there are several tools for load testing with various use cases. Among them, finding the appropriate tools for serving vendors purposefully is tough. Besides, designing the test bed using available machines, picking the highly configured machines, designing test case scenarios based on real findings are most challenging. Furthermore, for getting better and optimized performance from distributed systems, developers need to find out the hardware, software, and network system components which limit the performance. However, there are many open source software for testing functional behavior and performance. We choose Apache JMeter [9] as a testing tool and Ansible as IT automation engine because as they are simple, powerful and cross platform supportive. We use 10 client machines for giving concurrent loads.

B. Creating Test Scenarios

Our strategy is to make servers busy with highest concurrent loads. In the meantime, we send requests to download a specific file concurrently using curl request [19] as much as possible during heavy loads and save the data output metrics for further analysis. To compare the performances between conditions with and without load, we send request to download

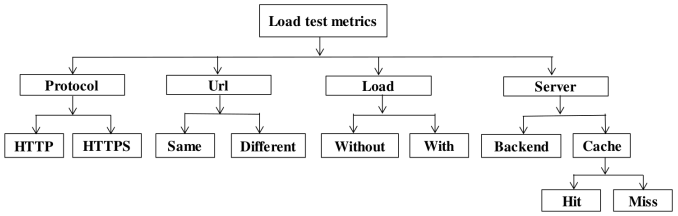


Fig. 2. Test case hierarchy of proposed load test metrics

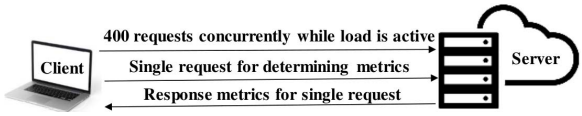


Fig. 3. Architecture of svLoad for single client

a specific file repeatedly as much as possible without load. Recently, maximum systems support HTTPS along with HTTP requests for enhancing the security. A simple overhead arises related to HTTPS requests, as it takes more time to download HTTPS type URL for resolving SSL/TLS keys.

Furthermore, concurrent requests may contain same URL or different URL which can also affect performances. Hence, request protocol type and URL type are two necessary metrics to design test case scenarios. As we have two type of servers, cache and backend cloud server, the server type is also a variable metric. Cache hits and misses are other obvious metrics to design test case scenarios for cache servers. Since in case of a cache miss, it takes more time to deliver a response than in the case of a cache hit. To summarize, we state that server type, protocol type, URL type, with or without load conditions, cache hits and misses are our metrics for designing test case scenarios. We propose several test case scenarios varying those parameter metrics such as:

1***) Protocol:*** Request protocol types are of two types i.e. HTTP and HTTPS request protocol.

2***) URL:*** Test cases vary according to request URL (Uniform resource locater). Hence, performance of cache and backend servers depend on the concurrent hits of same or different URL.

3***) Server:*** Backend and cache servers are two parameter metrics for designing the test cases. For cache server, two other important metrics are cache hit and miss based on different URL.

4***) Load:*** For analyzing the system's functionality properly, developers should find out how the system will behave with or without load conditions. Hence, we choose them as important metrics for designing the test case scenarios.

From the combination of these parameters, we design a total of 24 test case scenarios i.e. TC0, TC1, TC2, up to TC23. Table I and Fig. 2 presents the test case scenarios. Here, TC4, TC5, TC16, and TC17 are invalid test cases as same URL cache miss can not be possible. Hence, we run load tests for each of the other 20 test cases.

C. Creating and Disseminating Scripts

We design several shell scripts for determining some metrics e.g. download time, connection time, HTTPS resolve time, etc. We run the scripts to hit URL either in backend or cache,

depending on test cases. We also run JMeter [9] in every machine independently. Hence, we have machine independent data for further processing. Furthermore, we design JMeter scripts for all 20 test cases and a bash script to send a specific requests multiple times for measuring performances. We manage the whole task from one management machine using Ansible [10] to install required software, transfer scripts to all machines and run test cases for specified durations.

After running test cases, results are saved to a specific folder in each machine. They are then moved to a management node to merge them for analysis, and converted to a central excel file. The whole task is completed from management node using minimal commands. Since we also need to extract all URL information of user accounts from backend server for concurrent get requests, we design the following script files:

1*)* ***UrlExtractScript:*** Scripts used to extract all URL from backend server. These script were run from management node.

2*)* ***JMeterScript:*** JMeter script for all test cases. This script was run individually from all test client machines.

3*)* ***ResponseMetricsScript:*** This script is used to collect data variables from each curl requests after a complete transfer of requests. This script extracts some predefined data metrics from every URL request and averages the data results. We collect several necessary data metrics among the responses from curl requests. Besides, we find out that these data metrics are important for analyzing system behavior for further improvement. Here, we summarize the critical data variable metrics from curl response [19]:

Size_download: The total amount of bytes that are downloaded. *Size_header:* The total amount of bytes of the downloaded headers. *Size_request:* The total amount of bytes that are sent in the HTTP request. *Speed_download:* The average download speed that curl measured for the complete download. *Time_appconnect:* The time, in seconds, needed from the start until the SSL/SSH/etc connect/handshake to the remote host is completed. *Time_connect:* The time, in seconds, it takes from the start until the TCP connect to the remote host (or proxy) is completed. *Time_namelookup:* The time, in seconds, it takes from the start until the name resolving is completed. *Time_pretransfer:* The time, in seconds, it takes from the start until the file transfer is just about to begin. *Time_redirect:* The time, in seconds, it takes for all redirection steps including name lookup, connect, pre-transfer and transfer before the final transaction is started. *Time_starttransfer:* The time, in seconds, it takes from the start until the first byte is just about to be transferred. This includes time_pretransfer and also the time the server needs to calculate the result. *Time_total:* The total time, in seconds, that the full operation lasted.

4*)* ***DataAnalysisScript:*** This script moves all files to central management node and generates an excel file from response metrics.

Furthermore, in a distributed denial-of-service (DDoS) attack, multiple compromised computer systems attack a server, website or other network resource. In our proposed methodology, our target is also to flood requests in cache or backend server using proposed test cases and observe miss rate, CPU, and memory usages. From statistics of these usages, we can check stability of our system under DDoS attacks as well.

TABLE II. GEOGRAPHIC LOCATION OF ALL MACHINES

Machine name	Machine type	Geographic location
Ba1	Backend Server	Montreal, Canada
Ca1	Cache Server	Montreal, Canada
Ma1	Management Server	Montreal, Canada
A	Client 1	Montreal, Canada
B	Client 2	Montreal, Canada
C	Client 3	Toronto, Canada
D	Client 4	Montreal, Canada
E	Client 5	Toronto, Canada
F	Client 6	Toronto, Canada
G	Client 7	New Jersey, USA
H	Client 8	Montreal, Canada
I	Client 9	Toronto, Canada
J	Client 10	Montreal, Canada

TABLE III. CONFIGURATION OF MACHINES USED IN LOAD TEST

Informations	Backend server	Cache server	Management machine	Client machine
Architecture	x86_64	x86_64	x86_64	x86_64
CPU(s)	16	48	16	1
On-line CPU(s) list	0-15	0-47	0-15	0
Thread(s) per core	2	1	2	1
Core(s) per socket	4	12	4	1
Socket(s)	2	4	2	1
NUMA node(s)	2	8	2	1
CPU family	6	16	6	6
Model name	Intel(R) Xeon(R) CPU E5620 @2.40GHz	AMD Opteron(tm) Processor 6174	Intel(R) Xeon(R) CPU E5620 @2.40GHz	QEMU Virtual CPU version 1.5.3
CPU MHz	2394.141	2199.967	2394.103	2393.998
Virtualization type	VT-x	AMD-V	VT-x	full Storage

In summary, JMeter allows maximum 400 to 500 concurrent requests from a single machine. We ran around concurrent 4000 load requests using JMeter from all client machines to the respective cache or backend servers. Besides, we sent a single get request sequentially using our proposed script algorithm for obtaining download related necessary information under this huge load test (in Fig. 3). We also transfer and collect files and automate the architecture using Ansible tool. This test based architecture using tools JMeter, Ansible and proposed scripts for load testing is not proposed yet in any literature.

IV. EXPERIMENTAL EVALUATION

We evaluate performance of our proposed load test architecture through a real implementation. We also present a comparison of performance between Varnish cache and Swift backend server after tuning the network system, Varnish cache, and Swift backend parameters in real scenarios. Before this, we first elaborate our experimental settings.

A. Experimental Settings

We use state-of-the-art configured machines i.e. one Swift cluster, one Varnish cache, one management machine, and ten client machines which are distributed to three different geographical locations i.e., Montreal and Toronto in Canada, and New Jersey in USA (Table II). Table III presents the hardware and software related informations of machines used for load

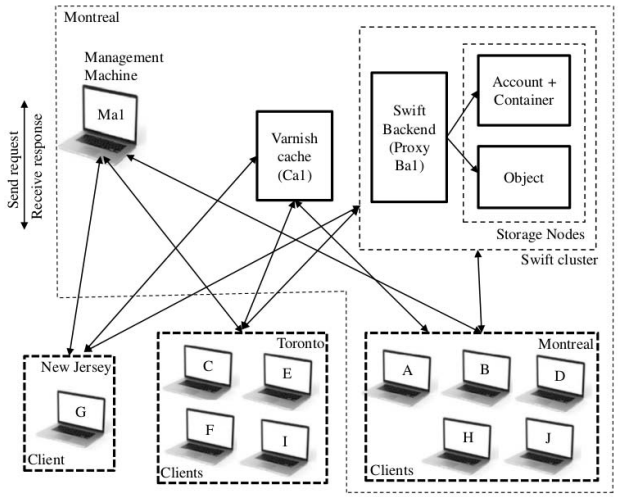


Fig. 4. Experimental settings of test bed

TABLE IV. VARNISH CACHE SERVER TUNING PARAMETERS

Parameters	Default values	Intermediate tuned values	Final tuned values
Thread_pool_minimum	5	5	400
Thread_pool_maximum	500	1000	5000
Thread_queue_limit	20	50	100
Workspace_thread	2k	4k	8k
Workspace_session	4k	0.5k	0.5k
Pipe_timeout	60 sec	15 sec	30 sec
Lru_interval	2 sec	10 sec	20 sec
Listen_depth	1024	2048	4096

testing. The average upload speeds of machines located in Montreal, Toronto, and New Jersey are 5.66 Mbps, 8.64 Mbps, and 207.9 Mbps respectively. Average download speeds are 14.35 Mbps, 3.98 Mbps, and 9.33 Mbps respectively.

Varnish cache server has 16 Gb memory, 64 Gb hard disk and six 1 Gb network interface cards. We install one proxy, one account-container, and one object server for Swift cluster. The memory and disk configurations of Swift servers are as follows: one proxy having 32 Gb memory and 1.2 Tb disk, one account-container having 32 Gb memory and 3 disks each of 400 Gb, and one object having 32 Gb memory and 3 disks each of 400 Gb. Each server had six 1 Gb network interface cards. Figure 4 presents the experimental setups of our testbeds. Here, we focus on proxy server as backend server as all requests hit through proxy server for further processing. We create $10,000$ accounts and $10,000$ containers in Swift cluster and upload around $55,000$ image files in those accounts for concurrent requests. We use 10 clients to provide concurrent loads on server, and from each client, 400 concurrent requests are sent for each test case.

B. Experimental Results

In this section, first, we present experimental results. Next, we delineate the parameters for system and network tuning.

Running the Scenario: We run the whole testing process 5 to 6 times with 2 hours duration for each test cases. We needed around 15 to 20 days for collecting the results and tuning the system. We also collect some predefined data response metrics from curl responses i.e., HTTP connection time, dns lookup time, download speed, app connection time, connection time, pre-transfer, start transfer, total response time, etc. to measure performances for each test case [19].

TABLE V. SWIFT BACKEND SERVER TUNING PARAMETERS

Parameters	Variables	Default values	Intermediate tuned values	Final tuned values
Memcached	MAXCONN	1024	2048	4096
	CACHESIZE	64k	1024k	4096k
File Descriptor	fs.file-max	8192	32768	2097192
Ulimit	Hard limit	4096	100000	400000
	Soft limit	1024	4096	100000

TABLE VI. SYSTEM NETWORK TUNING PARAMETERS

Parameters	Default values	Intermediate tuned values	Final tuned values
net.core.wmem_default	212992	131072	262144
net.core.wmem_max	212992	1048576	4194304
net.core.rmem_default	212992	131072	262144
net.core.rmem_max	212992	1048576	4194304

Monitoring the Scenario: We monitor and collect the output of CPU and memory usage, disk utilization, process queues, JVM out of memory exceptions, etc. while running test cases in corresponding server. We observed that 70% to 80% memory is used for all 48 cores in Varnish cache due to leveraging tasks to OS. After running 1st round of load test, we observe that some parameters of Swift, Varnish, and machine's network system must be tuned for better performance. We find out several bottlenecks related to these software through continuous load testing.

Next, we present necessary components and proper parameter values related to system and network tuning from analysis of data metrics through rigorous load testing. We benchmark system behavior through changing default values gradually identifying system metrics. Here, we only present best tuning values due to lack of space.

Swift Tuning: We locate three bottlenecks related to Swift tuning. When maximum load is given to the backend Swift proxy server, the memory cache (*memcached*) fails to handle large amount of requests, and after sometime, unsuccessful responses are generated. So, we change the *memcache.conf* file and increase the size of memory, cache and maximum number of allowed connection, and executed the load tests again. We also find out some parameters related to *memcache*, file descriptor, and ulimit that have great impact on load tests by changing them repeatedly. Table V presents the default and tuned values for Swift.

Varnish Tuning: We tune the necessary parameters of Varnish cache i.e. Thread_pool_minimum, Thread_pool_maximum, Thread_queue_limit, Workspace_thread, Workspace_session, Pipe_timeout, Lru_interval and Listen_depth, keeping other values default. Table IV presents default and tuned values for Varnish cache.

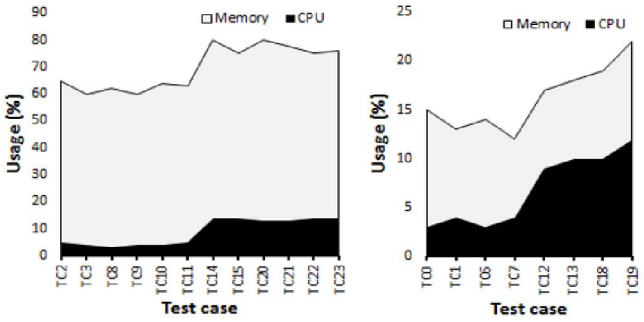
Network system tuning: Kernel buffer parameters i.e. net.core.wmem_default, net.core.rmem_default, net.core.rmem_max and net.core.wmem_max show the

TABLE VII. SUCCESS AND MISS RATES (%) OF REQUESTS

Test case id	Before Tuning		After Tuning	
	Success	Miss	Success	Miss
TC13	57%	43%	100%	0%
TC15	99%	1%	100%	0%
TC19	51%	49%	99%	1%
TC21	99%	1%	100%	0%
TC23	99%	1%	100%	0%

TABLE VIII. PERCENTAGE OF AVERAGE RESPONSE TIME IMPROVEMENT FOR ALL CLIENTS

Test case id	Response time (s)		Improvement
	Before tuning	After tuning	
TC0	0.10	0.02	79%
TC1	0.14	0.41	-66%
TC2	0.07	0.03	57%
TC3	0.93	1.13	-18%
TC6	0.10	0.02	81%
TC7	1.20	0.47	60%
TC8	0.08	0.02	69%
TC9	0.71	0.42	41%
TC10	0.08	0.02	69%
TC11	0.78	0.82	-5%
TC12	0.27	0.24	10%
TC13	0.27	1.83	-85%
TC14	0.27	0.25	6%
TC15	1.56	2.14	-27%
TC18	0.27	0.24	10%
TC19	2.50	1.66	34%
TC20	0.28	0.25	10%
TC21	1.79	1.18	34%
TC22	0.28	0.25	12%
TC23	1.59	1.67	-5%



(a) Cache server (b) Backend server

Fig. 5. Average memory and CPU usage of cache and backend server for 20 test cases

default and maximum write (receiving) and read (sending) buffer size allocated to any type of connection. The default values are low since the allocated space is taken from the RAM. Increasing this improves the performance for systems running servers. Table VI presents the defaults and tuned values for network system.

After 1st round of load test, we observe that for TC0, TC1, TC2, TC3, TC6, TC7, TC8, TC9, TC10, TC11, TC12, TC14, TC18, TC20, and TC22 success response is 100%. Five test cases have highest miss rate due to HTTPS requests and lack of tuning. Hence, we tune the system and perform load test multiple times. After tuning, success rates for all test cases improved to 99%. We present the success and miss rates for TC13, TC15, TC19, TC21, and TC23 in Table VII.

TABLE IX. IMPROVEMENT OF CACHE SERVER RESPONSE TIMES OVER BACKEND SERVER

Backend Test case id	Cache Test case id	Improvement (%)
TC0	TC2	29%
TC1	TC3	-85%
TC6	TC8	23%
	TC10	18%
TC7	TC9	40%
	TC11	34%
TC12	TC14	0%
TC13	TC15	-83%
TC18	TC20	-4%
	TC22	-4%
TC19	TC21	29%
	TC23	36%

TABLE X. COMPARISON OF HTTP REQUEST RESPONSE TIMES OVER HTTPS REQUEST RESPONSE TIMES

HTTP Test case id	HTTPS Test case id	Improvement on HTTP (%)
TC0	TC12	91%
TC1	TC13	77%
TC2	TC14	87%
TC3	TC15	47%
TC6	TC18	91%
TC7	TC19	71%
TC8	TC20	90%
TC9	TC21	64%
TC10	TC22	90%
TC11	TC23	50%

Table VIII shows the %improvement of response time after final tuning. In these test cases, miss rate is high hence total response time is bit higher before tuning. Furthermore, after tuning the system, miss rate is decreased hence lower the response time. Figure 5 presents average memory and CPU usage of cache and backend server. These usage are remain almost same after the tuning.

C. Experimental Findings

In this section, we present the findings of tuning the system, cache and backend, and comparing request times of HTTP and HTTPS by analyzing the results.

Analyze test results: We analyze test cases for comparing the average response time from all the clients for Varnish cache and Swift backend server. This behavior remains same after tuning the servers as response time ratio from cache and backend servers varies depending on the test cases. Note that, each client machine requests for downloading same URL once when testing without load criterion, hence clients hit total 10 requests concurrently. Besides, each client machines requests for downloading same or different URL for 400 times when testing with load criterion, hence clients hit total 4000 requests concurrently.

Furthermore, we present comparison of cache and backend server response time in Table IX. In every machine, TC2's response time is 29% faster than TC0's. The condition holds upto 6-7 threads per machine. TC1's response time is 85% faster than TC3's. The response time for cache is 5 to 6 times higher than backends. TC6's response time is 23% and 18% slower than TC8's and TC10's. TC7's response time is 40% and 34% slower than TC9's and TC11's. TC12's response

time is almost the same to TC14's. TC13's response time is 83% faster than TC15's as the server takes some extra time to translate a HTTPS request to HTTP request. TC18's response time is 4% faster than TC20's and TC22's as cache hit and miss in without load condition. TC19's response time is 29% and 36% slower than TC21's and TC23's.

In addition, we present comparison of HTTP and HTTPS requests response time in Table X. Here, HTTP type test cases are up to 90% faster than HTTPS protocol type as more times needed for resolving SSL/TLS keys. Besides, TC0, TC1, TC2, TC3, TC6, TC7, TC8, TC9, TC10, and TC11 are 91%, 77%, 87%, 47%, 91%, 71%, 90%, 64%, 90%, 50% faster than TC12, TC13, TC14, TC15, TC18, TC19, TC20, TC21, TC22, and TC23 respectively. In summary, we conclude that for each test case scenario in every machine, response time for cache is lower than the backend except for the same URL requests. Varnish restricts concurrent request at a time for same URL's.

V. Conclusion and Future Work

Performing load testing, identifying system bottlenecks and tuning them accordingly are not focused much in the literature for cloud systems. Therefore, in this paper, we perform our study on these aspects. Here, we investigate several load testing considering diversified real scenarios and tuning system parameters based on findings of load testing. Our tuning results in substantial improvement in most cases. There is a plenty of room to extend the study. We plan to explore in future that how the system will behave for extensive PUT and POST requests, as the proxy server has extra overhead related to processing and I/O operations of such requests. Our future plan also aims to use more clients and expand the cache and backend servers.

Acknowledgments

This research was funded and supported by the ICT Division, Government of Bangladesh and IPvision Canada Inc. This work was also supported in part by the US National Science foundations under grants IIS 1559588 and CBET 1743985. Any opinions, thoughts and findings are those of the authors and do not reflect views of the funding agency. The authors would like to thank Hasan I. Akbar and Ruhul A. Sujon for their help during the study.

References

[1] J. Arnold and members of the SwiftStack team, *OpenStack Swift*. O'Reilly, Copyright SwiftStack, Inc., ISBN 978-1-491-90082-6, 2015.

[2] "Varnish cache." [Online]. Available: http://dx.doi.org/10.1090/S0894-0347-96-00192-0. Accessed: Mar. 28, 2018.

[3] J. Noor and A. B. M. A. A. Islam, "ibuck: Reliable and secured image processing middleware for openstack swift," in *IEEE International Conference on Networking, Systems and Security (NSysS)*, Dhaka, Bangladesh, Jan. 2017.

[4] J. Noor, H. I. Akbar, R. A. Sujon, and A. B. M. A. A. Islam, "Secure processing-aware media storage (spms)," in *36th IEEE International Performance Computing and Communications Conference (IPCCC)*, San Diego, California, USA, Dec. 2017.

[5] K. Veeraraghavan, J. Meza, D. Chou, W. Kim, S. Margulis, S. Michelson, R. Nishtala, D. Obenshain, D. Perelman, and Y. J. Song, "Kraken: Leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, Nov. 2-4, 2016.

[6] R. Khan and M. Amjad, "Performance testing (load) of web applications based on test case management." *Perspectives in Science*, vol. 8, pp. 355–357, Sep. 2016.

[7] M. D. Syer, W. Shang, Z. M. Jiang, and A. E. Hassan, "Continuous validation of performance test workloads." *Automated Software Engineering*, vol. 24, no. 1, pp. 189–231, Mar. 2017.

[8] A. Avritzer, J. Kondek, D. Liu, and E. J. Weyuker, "Software performance testing based on workload characterization," in *WOSP '02*, Rome, Italy, Jul. 24-26, 2002.

[9] "Apache jmeter," [Online]. Available: http://jmeter.apache.org/. Accessed: 7 Jun. 2017.

[10] "Ansible," [Online]. Available: https://www.ansible.com/. Accessed: 7 Jun. 2017.

[11] R. Bhatia and A. Ganpati, "In depth analysis of web performance testing tools," *IRACST Engineering Science and Technology: An International Journal (ESTIJ), ISSN: 2250-3498*, vol. 6, no. 5, Sep.-Oct. 2016.

[12] N. Khanghahi and R. Ravanmehr, "Cloud computing performance evaluation: Issues and challenges," *International Journal on Cloud Computing: Services and Architecture (IJCCSA)*, vol. 3, no. 5, Oct. 2013.

[13] E. J. Weyuker and F. I. Vokolos, "Experience with performance testing of software systems: Issues, an approach, and case study," *IEEE Transactions on Software Engineering*, vol. 26, no. 12, Dec. 2000.

[14] S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, J. J. L. A. Bertolino, and H. Zhu, "An orchestrated survey on automated software test case generations." *The Journal of Systems and Software*, 2013.

[15] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," *Future of Software Engineering(FOSE'07), 0-7695-2829-5/07, IEEE*.

[16] Compuware, "Applied performance management survey," Sep.-Oct. 2006.

[17] C. U. Smith, *Performance Engineering of Software Systems*. Addison Wesley, 1990.

[18] C. Smith, *Software Performance Engineering*. Encyclopedia of Software Engineering, Wiley, 2002.

[19] "Curl.1 the man page," [Online]. Available: https://curl.haxx.se/docs/manpage.html. Accessed: 7 Jun. 2017.