# Node.js Scalability Investigation in the Cloud

Jiapeng Zhu
University of New Brunswick
Fredericton, NB, Canada
jzhu3@unb.ca

Panagiotis Patros
University of Waikato
Hamilton, Waikato, New Zealand
University of New Brunswick
Fredericton, NB, Canada
ppatros@waikato.ac.nz,patros.panos@unb.ca

Kenneth B. Kent
University of New Brunswick
Fredericton, NB, Canada
ken@unb.ca

Michael Dawson
IBM Runtime Technologies
Ottawa, ON, Canada
Michael_Dawson@ca.ibm.com

## ABSTRACT

Node.js has gained popularity in cloud development due to its asynchronous, non-blocking and event-driven nature. However, scalability issues can limit the number of concurrent requests while achieving an acceptable level of performance. To the best of our knowledge, no cloud-based benchmarks or metrics focusing on Node.js scalability exist. This paper presents the design and implementation of *Ibenchjs*, a scalability-oriented benchmarking framework, and a set of sample test applications. We deploy *Ibenchjs* in a local and isolated cloud to collect and report scalability-related measurements and issues of Node.js as well as performance bottlenecks. Our findings include: 1) the scaling performance of the tested Node.js test applications was sub-linear; 2) no improvements were measured when more CPUs were added without modifying the number of Node.js instances; and 3) leveraging cloud scaling solutions significantly outperformed Node.js-module-based scaling.

**ACM Reference Format:**
Jiapeng Zhu, Panagiotis Patros, Kenneth B. Kent, and Michael Dawson. 2018. Node.js Scalability Investigation in the Cloud. In *Proceedings of (CASCON'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.475/123_4

## 1 INTRODUCTION

Clouds provision on-demand computing resources over a network of nodes in a pay-as-you-go manner. Cloud applications are commonly implemented with the client/server paradigm; thus, cloud workload is mainly driven by requests. Node.js—which is essentially Javascript on the server—has gained popularity in developing cloud applications. It is fast and a good fit for (micro)service-oriented cloud deployment due to its single-threaded, asynchronous non-blocking I/O and event-driven nature. Node.js applications can be deployed and run via Platform as-a-Service (PaaS) cloud software, which abstract large parts of the software/hardware stack.

As the number of concurrent clients and their requests increase, clouds elastically scale provisioned resources such that the deployed applications maintain an appropriate quality of service. Scaling up or vertical scaling adds more resources to a cloud instance, but has minimal benefits for Node.js since its event-loop becomes a bottleneck permitting the execution of only one Javascript function at a time. Instead, via scaling out or horizontal scaling, multiple instances of a cloud application can be started up; PaaS software provide generic scaling solutions for any type of containerized applications. Additionally, Node.js provides its own specific scaling solutions; *Cluster* is such a Node.js module that forks multiple worker processes to handle requests.

Thus, a number of questions arise regarding the efficacy of Node.js in utilizing cloud resources as well as selecting an appropriate scaling strategy. Such knowledge could be leveraged by cloud providers—via improved autoscaling, fine-tuned cloud architecture or even expert advice to their clients—to better satisfy their service level agreements while minimizing costs. To the best of our knowledge, this is the first attempt to formalize a set of tools, methodologies and benchmarking frameworks for experimentally evaluating the scalability of Node.js deployed on PaaS clouds. The main contributions of this paper, which contains elements from the first author's master's thesis [21], are: First, we implement and open source *Ibenchjs*, a scalability-oriented benchmark framework suitable for PaaS, and a set of resource-intensive test applications[1]. Second, we perform various scalability experiments on a six-host Docker Swarm cloud with various scaling strategies and patterns. Third, we analyze scalability effects of different types of test applications; identify associated performance bottlenecks; quantify and characterize Node.js scalability; and leverage regression analysis to fit the measured performance into theoretical scalability models.

## 2 BACKGROUND

**PaaS Clouds and Docker Swarm:** Cloud computing abstracts and virtualizes computing, networking, and storage resources, providing the end-user with elastic and on-demand services in a pay-as-you-go manner. Platform-as-a-Service (PaaS) clouds furnish users with a ready-to-use platform for maintaining a complete software lifecycle: development, deployment, testing and maintenance [14].

Docker [2] is a platform for developers and system administrations to build, ship, and run distributed applications, in a platform-agnostic way. Users compile and upload their own filesystem images

*CASCON'18, October 29-31, Markham, ON, Canada*
© 2018 Copyright held by the owner/author(s).
ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

[1]https://github.com/CAS-Atlantic/ibenchjs

that are subsequently deployed and run in containers. A Docker container runs directly within the host's kernel, and only packages the application, its related binaries and libraries. Thus, a separate full OS is not required, which makes it lightweight and fast starting. Docker containers are isolated from each other and are constrained to only utilize the specified computing resources via Linux Group Control (cgroups). Cgroups enable containers to share available resources, optionally enforcing limits and constraints.

Docker Swarm is a container management software; it provides native clustering functionality for Docker and turns a pool of multiple Docker hosts to a single Docker cluster[3]. A Docker host is designated as the manager and is responsible for handling cluster management tasks, such as maintaining cluster states and scheduling services. Multiple Docker worker nodes receive and execute tasks dispatched from the manager node. Docker Swarm allows users to package all the services of the application as a stack, referred to as the service stack that runs in the container. The deployment of a service stack can be automated via a YAML "docker-compose" file. To achieve horizontal scaling, Docker Swarm users can specify a number of replicated containers. Based on this, the application is scaled out by creating Docker containers dispatched across the cluster's nodes. Docker Swarm maintains an overlay network that facilitates communications among Docker daemons participating in the swarm. The overlay network also allows users to attach Docker containers to it and enables container-to-container communications within the same or different swarm nodes. The ingress network is a special overlay network that manages load balancing among containers. Load balancing is a mechanism that distributes incoming loads from clients to servers in the cloud system [8]. Within the ingress network, all swarm nodes participate in the ingress routing mesh and enable load balancing. Thus, any swarm node can receive requests on a published port, and then it routes them to published ports on available nodes of an active container.

**Scalable Node.js:** Node.js is a framework for Javascript running on Google's V8 engine [4]. Node.js is most suited for I/O intensive tasks and promises a fast running speed in the cloud, because of its event-driven, asynchronous and non-blocking I/O nature. Node.js leverages a single thread for Javascript execution. Therefore, it cannot fully use CPU resources in a multicore system without utilizing a module that supports multi-threaded functionality: A Node.js application can be scaled up by adding more CPUs or other hardware; however, its performance is not expected to improve proportionally. We refer to a scaling strategy that provides more CPUs to a Node.js instance as a **UP strategy**.

However, Node.js servers can be scaled via alternate scalability strategies that divide the event loop. Cluster is a Node.js module that spawns multiple Node.js processes [6], offering a pattern of a master parent controlling a single or multiple children. When Cluster is applied to a web server, these worker processes handle requests in parallel—assuming execution on a multicore server—with the master process, which is in charge of managing this system by distributing incoming loads round-robin. We refer to Cluster's scaling strategy as **CM strategy**. Apart from using Node.js-specific solutions, applications can be scaled via horizontal scaling offered by PaaS clouds. In Docker Swarm, a service can be replicated on multiple containers across the cluster. We refer to a scaling strategy that leverages cloud-based horizontal scaling as an **HS strategy**.

**Scalability:** Scalability is a measure of a system's capacity to handle workloads, as hardware resources are added [20]. Relative capacity as the ratio of the capacity with p processors to the capacity with one processor [11]. From a web application's perspective, relative capacity is defined via the maximum throughput in terms of the number of requests handled within a time-frame, while an acceptable response time is maintained. Additionally, scalability models express relative capacity as a function of the number of processors. The intuitive best-case model is that of Linear-scalability:

$$C_L(p) = p \quad (1)$$

Linear scalability predicts a relative capacity equal to the processor count: Doubling the number of CPUs, we should achieve twice as much throughput. Also, linear scalability represents an ideal situation, and a system rarely achieves it due to hardware limitations, service restrictions, network speed, intra- and inter-processor communications, etc.

Instead, scalability follows sub-linear trends, whereby relative capacity increases slower than the number of added hardware. Amdahl's Law [10] is such a model: it states that the potential maximum speedup of parallel processing is limited by the serial portion of a computation, referred to the serial fraction $\sigma$. Amdahl's Law can be written based on relative capacity as follows:

$$C_A(p) = \frac{p}{1 + \sigma(p-1)} \quad (2)$$

Extending Amdahl's Law, Gunther's Universal Scalability Law (USL) [9] not only considers the serialization that is referred to as contention, but also the interprocessor communication overhead that is referred to as coherency cost. USL is defined as follows:

$$C_U(p) = \frac{p}{1 + \sigma(p-1) + \lambda p(p-1)} \quad (3)$$

The USL model describes that systems follow sub-linear scalability due to contention ($\sigma$) and coherency ($\lambda$). Contention is caused due to serialization (the workload cannot be processed in parallel, but in serial) or queuing; coherency penalizes the system's performance because the system's parts require to maintain a consistent state. The independent variable $p$ in Equation (2) and (3) can represent the number of virtual users (workload). When the hardware configuration (e.g. number of processors) remains fixed and the workload varies, it is referred to as software scalability. [9]

Load testing is involved in scalability evaluation to investigate the system's scaling capabilities under a specific load. Load testing tools generate large numbers of traffic to a web application. Apache JMeter [1] is such a tool; it executes user-provided test plans, which describe a series of execution steps. In addition, users can customize several load testing parameters, such as the number and concurrency of requests, test duration and the server's URL.

## 3 RELATED WORK

Lei et al. conducted benchmark and scenario tests to compare the performance of Node.js, Python-web and PHP [12]. Their tests followed a one-factor-at-a-time manner; they varied the number of concurrent clients and fixed the number of requests. They utilized test tools to make concurrent request loads on three test applications, Hello World, Fibonacci Calculation and Select Operation of DB, and two scenarios, Login and Encryption. They measured response time and throughput, and concluded Node.js significantly outperformed Python and PHP. We adapt this one-factor-at-a-time experimental in our work but focus on scalability and cloud deployment. Additionally, in a work that leveraged Node.js deployed on Docker Swarm as a cloud service for a specific application, a scalability test was conducted to investigate a satisfactory ratio of Node.js vs. NGINX load-balancer instances [15]. However, testing Node.js scalability was not the focus of that work and no theoretical analysis, regression or resource-specific testing was conducted.

Aronis et al. developed a scalability benchmark suite for Erlang/OTP [7]. A set of synthetic benchmarks that measure the scalability of a specific aspect of Erlang and three real-world benchmarks were included. The architecture of the suite can be divided into four components: the coordinator, the executor, the sanity checker and the graph plotter. The coordinator finds out what and how to execute by reading configuration files; the executor runs a specific benchmark in a particular runtime environment; the sanity checker verifies the output produced by the benchmarks; and the graph plotter processes the scalability measurements and visualizes the results. Our *Ibenchjs* was partially inspired by this design.

Patros et al. introduced Cloud Burners [17], a set of Java EE cloud tenants that target specific hardware resources: CPU, cache, resident memory, disk I/O, and network I/O. Cloud Burners were utilized to propose a set of resource-slowdown and resource-intensiveness metrics. Similarly, the sample benchmarks in this project are designed to target and stress specific resources. However, unlike Node.js, Java is multithreaded and can directly leverage multicores.

Williams and Smith [20] reviewed four models of scalability: Linear, Amdahl's Law, Super-Serial Model, and Gustafson's Law. They concluded that vertical scalability fitted best by Amdahl's Law or Super-Serial Model; horizontal scalability was best described by Gustafson's Law and Linear scalability. Further, Tsai et al. proposed the following metrics for testing the scalability of Software as-a-Service SaaS applications in the cloud [19]: 1) processing time, 2) resource consumption, 3) performance resource ratio, and 4) metric variance. Also, Schwartz [18] described an approach to measure and model scalability with USL. Based on these works, we perform regression analysis to theoretically model Node.js scalability.

A performance model of a system with a certain number of clients firing repeating requests on a tight loop for a certain number of processing elements has been proposed [16]. In that model, as the workload increases, the response time remains stable until the system is saturated, after which, starts degrading linearly. Instead, according to the model, throughput increases until the system becomes saturated and remains stable thereof. The predictions made by this model generally corroborate with our results; however, we measured increased degradation after the saturation point—presumably due to Node.js instances competing for a fewer number of CPUs, which that model did not consider.

## 4 NODE.JS SCALABILITY BENCHMARKING

To investigate Node.js scalability in PaaS clouds, we develop *Ibenchjs*, a scalability-oriented benchmark framework, and a set of sample test applications. *Ibenchjs* can evaluate and measure different scalability strategies applied in Node.js. Overall, *Ibenchjs* follows a two-tier architectural model, which consists of a client side and a server side (Figure 1). Users can configure and launch a benchmark run, post-process the captured raw data, and generate a final report. The server side maintains images of the test applications and collects the deployed containers' resource consumption.

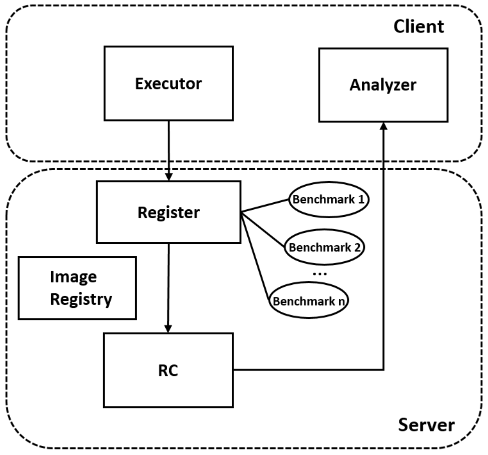


**Figure 1: *Ibenchjs* Benchmark Suite Architecture**

**Table 1: *Ibenchjs*'s Scalability Variables**

| Variable | Description |
|---|---|
| Number of nodes | Independent variable for the number of hosts participating in the cloud. |
| Concurrency | Independent variable for the number of parallel requests fired to the server. |
| Number of Node.js instances | Independent variable for the number of Node.js instances created by the different scalability strategies. |
| Running time | Independent variable for the total time spent running one benchmark. It starts when users fire requests, and ends when users receive all responses. |
| Response time | Dependent variable for the time elapsed between a user initially sending a request and receiving its response. |
| Throughput | Dependent variable for the number of requests handled within a time-frame. |
| Computing resource usage | Dependent variable for the amount of computing resources used during a benchmark run. |

### 4.1 Scalability Variables

Table 1 shows the scalability variables of *Ibenchjs*; they can be classified into independent (configuration) and dependent (measured) variables. The independent variables represent the scaling aspect of the execution environment. They include *number of nodes*, *workload concurrency*, *number of Node.js instances*, and *running time*. The dependent variables represent the aspects of the system behavior that can be affected by changing independent variables, which can be manipulated for a scalability analysis. The dependent variables in the benchmark framework include *throughput*, *response time* and *computing resource usage*. All these variables can be utilized to quantify and evaluate scaling effects in different scalability strategies, essentially defining a multi-variable configuration-outcome space.

## 4.2 Benchmark Framework

The *Ibenchjs* benchmark framework consists of five components: the executor, the RC, the analyzer, the register, and the image registry. This service-oriented design makes for a convenient choice for deployment on PaaS clouds, even the client components can be deployed as cloud services; the various parts of *Ibenchjs* communicate asynchronously and, in future embodiments, could be scaled out individually to scalability-stress-test larger deployments.

The **executor** runs on the client side and launches a complete benchmark run. It is implemented using a set of bash scripts that start various types of applications based on the user's configurations. The executor has the following responsibilities: 1) It controls the entire benchmark process; 2) It maintains the service stack in the PaaS cloud (Docker Swarm); 3) It generates and fires concurrent requests to the server side using JMeter (a workload driving tool); 4) It triggers the RC (to be presented shortly) component, which is running on the server side; and 5) It generates the final report. The executor is configurable, receiving parameters, such as server's URLs, workload concurrency and number of instances, and passes them to the necessary technology stacks such as, JMeter and the underlying PaaS cloud.

In addition to the benchmark runtime environment configuration, the executor deploys and removes benchmarks as a service stack in and from the PaaS cloud. They are both implemented by sending a remote CLI through SSH.

Once users complete the service stack maintenance (deletion or deployment) on the PaaS cloud, the executor triggers the initial data collection. Afterwards, the executor starts the load testing tool, JMeter, to generate and fire several concurrent requests to the server side. Finally, the executor finishes the load testing and the second-time resource usage data collection, and triggers the analyzer component to parse and post-process the raw data, printing out a benchmark-run final report.

The **RC**, implemented in Node.js, runs on the server side and it takes a snapshot of resource usage before and after a benchmark run. It runs in the manager node as a web server, publishing an open port and waiting for requests. The executor can access it via HTTP—as discussed in the previous paragraph. In turn, the RC component measures the current resource usage and then transmits it back to the executor component. The RC collects various types of data from all worker nodes via remote SSH connections. For example, the RC captures container runtime information and resource usage of containers, measured by either access cgroups (pseudo-filesystem) or executing the appropriate shell commands.

During the first-time data collection, the RC writes the captured snapshots to a JSON file on the disk—a future embodiment could outsource the storage of this data to a persistence PaaS service, preferably one that offers object storage. When the RC is triggered to perform the second-time data collection, it captures the required data; retrieves the previously stored JSON file; converts the contents in the JSON file to a JSON object; adds the newly captured data to the JSON object; and transmits this object back the client.

The **analyzer** is responsible for post-processing collected data and is implemented in Node.js. There are two data sources as inputs that are passed by the executor component. Resource usage data that are collected from the RC and load testing log files that are generated from JMeter. The original resource usage data is JSON-formatted and contains results of two data collections in a key-value pair format. In addition, the total benchmark execution time, RSS data collected in a period of 5 seconds, and container placement among swarm nodes information are also attached. The analyzer component parses such JSON files and extracts diverse types of resource usage data from all worker nodes. JMeter's log file is a CSV with headers, each line representing a single request. We consider the columns: 1) timestamp; 2) response code (e.g. 200 indicates a successful request); and 3) response latency in our embodiment. The extracted and parsed data from the log file are utilized to calculate response time and throughput. Once the analyzer finishes parsing, extracting and post-processing the various data sources, the final metrics are determined, which include scalability related metrics and container-placement information.

The **register** is a Node.js *Express* web server with a set of sample and/or user-defined test applications associated with it. It acts as an access point, accepting concurrent requests from the client side, and calls the associated test applications to handle them. *Express* is a Node.js web application framework that provides a set of features for web applications [5]. The register handles HTTP requests from the client, which is implemented via a method of the *Express* module *app* object that corresponds to HTTP methods.

The routing methods of the *Express* module have two arguments passed, one is a defined endpoint, and the other one is a callback function. When the benchmark server receives a request to a specific endpoint from clients, the callback function is executed. In this case, the callback function corresponds to a specified test application, and the test application runs once a request accesses its associated endpoint. The register is containerized to package all benchmarks and itself and form a service stack, running in a container. Once the containerization is completed, an image of the service stack is created and it can be stored in an image registry. It should be stressed that when scaling out, the whole combination of the register and the benchmarking applications is replicated.

Finally, the **Image registry** runs in the server side to store images and distribute them to any node in the PaaS cloud. Therefore, users can push their customized images to it from any node as a client, and in turn any node as a client can access it and pull images. Consequently, containers can be distributed and created based on pulled images in different nodes in the PaaS cloud. The private image registry limits users' images to their private PaaS cloud for improved security and privacy, instead of pushing, potentially sensitive, data in a public environment.

## 4.3 Sample Benchmarks

*Ibenchjs* comes with a number of preset test applications referred to as benchmarks. They are resource-intensive, because such programs stress resource usage in systems and easily cause scaling when reaching certain resource thresholds, for scalability evaluation purposes. Furthermore, resource-intensive benchmarks can be utilized to detect systems' performance bottlenecks, caused by a specific resource saturation. To this end, the sample benchmarks are classified into CPU-intensive benchmark, disk-intensive benchmark, network-intensive benchmark and memory-intensive benchmark. They are designed to target and stress one resource type:

**Table 2: Resource Usage of Sample Benchmarks**

| Metric / Benchmark | CPU (%) | Net. Thru. Inc. (MB/s) | Net. Thru. Out. (MB/s) | Disk Thru. (MB/s) | RSS (MB) |
|---|---|---|---|---|---|
| **CPU int.** | 94.84 | 0.11 | 0.15 | 0.00 | 38.18 |
| **Net Int. In** | 45.04 | 2.76 | 0.44 | 0.00 | 42.50 |
| **Net Int. Out** | 48.80 | 0.27 | 25.20 | 0.00 | 43.37 |
| **Disk Int.** | 65.00 | 0.18 | 0.25 | 0.68 | 74.58 |
| **Mem. Int.** | 4.56 | 0.00 | 0.00 | 0.00 | 101.17 |
| **Baseline** | 77.56 | 0.19 | 0.26 | 0.00 | 38.94 |

The **CPU-intensive benchmark** decides if a large integer is a prime. It executes a tight loop to process the calculation conducting any other non-CPU intensive operations, such as I/O. Users can specify an integer to be tested. By changing the examined value, the number of loop iterations execution can be varied accordingly. In other words, this affects how much a single request stresses the CPU. The **disk-intensive benchmark** writes a number, in terms of current milliseconds, to a file. This file is stored on disk when writing operations complete and the file is closed, which produces disk I/O operations to target disk resources. There are no other operations involved, excluding disk I/O operations. The **network-intensive benchmark** targets incoming and outgoing network traffic. The incoming network-intensive test application receives a graph from the client via POST without conducting any processing. Essentially, it works as a network drain, consuming any incoming network bandwidth. A JMeter test plan for POST method is created in the executor component, which is similar with the GET method test plan, excluding the HTTP method change and a file path to the graph added. Once the benchmark server receives the graph, a callback function referenced from the network-intensive test application is called to respond to the client. The network-intensive test application targeting the outgoing network traffic sends fixed-size text to the client side. The **memory-intensive benchmark** keeps inserting a number of objects in an array, which consequently causes the memory size to grow to a large size but without exceeding the total memory size allocated to the container. We use a *sleep* function, requesting the OS assigning CPU to another processes, such that the CPU is stressed less. The reason is the memory-intensive benchmark should not target the CPU, only the memory. Furthermore, each request creates a unique array and cannot be shared, thus it avoids any CPU cache coherence and contention effects. Finally, the **baseline benchmark** immediately responds back to the client, utilizing only the application server.

We initialized *Ibenchjs* with our resource-intensive benchmarks to verify their resource targeting as well as to test our benchmarking framework and cluster/cloud setup. We utilized *Ibenchjs* in Docker Swarm to perform the benchmark runs. This verification is neither a stress test nor a scalability analysis, thus we did not launch heavy workloads, but fired two parallel requests to one Node.js instance for 100 seconds.

Each benchmark was effective in targeting its designated resource. The CPU-intensive application scored the highest CPU utilization, approximately 95%, which is expected since it executes a tight loop. The two network-intensive applications caused the highest network throughput, for both incoming and outgoing traffic respectively: both types send and receive large amounts of data over the network. The disk-intensive benchmark test application obtained the highest disk I/O throughput; Node.js applications normally do not involve disk I/O operations. The memory-intensive benchmark test application consumed much more resident memory than others, as it stores multiple objects on the heap. Finally, the baseline test scored moderately high CPU utilization, which can be attributed to the application server having to complete smaller and thus, more frequent requests.

## 5 EXPERIMENTAL INVESTIGATION

To investigate Node.js scalability issues, we experimented in a private and isolated cloud, and collected scalability metrics using *Ibenchjs* with its sample benchmarks. Node.js is heavily applied in server development and the network I/O and computational tasks are often involved in Node.js servers. Therefore, We mainly utilize the baseline, CPU- and network-intensive test application with *Ibenchjs* to conduct several scalability experiments. We built our private cloud, which provides a scalable, distributed and multicore cloud environment, using Docker Swarm. To obtain a clean, isolated and consistent experimental environment, we installed one Oracle VirtualBox VM, managed by Vagrant, on each of the six physical servers. All installed VMs ran Ubuntu 16.04.3 LTS OS, and allocated the same number of CPU cores and amount of memory as their host machines. Static IP addresses were assigned on each VM. Docker engine was installed on each VM and Docker Swarm was then enabled on the manager and each of the five worker nodes.

### 5.1 Methodology

We define two scalability patterns: 1) **Single/Multiple Instance on Single Node (S/MISN) Pattern**: refers to single or multiple instances of a Node.js application deployed in one Docker Swarm node. In this scalability pattern, we compare three scalability strategies, UP, HS and CM is conducted; 2) **Multiple Instances on Multiple Nodes (MIMN) Pattern**: is referred to as multiple instances of Node.js application deployed in multiple swarm nodes participating in the Docker swarm. This pattern extends the single node to multiple nodes, up to five, in swarm. In this scalability pattern, we evaluate only the HS strategy—CM cannot leverage multiple nodes and UP proved (as the reader will see shortly) to be ineffective in producing any speedups on a single node.

We define the following scalability strategies: **UP** allows the VM to utilize more of the CPUs of the bare-metal host, mapping one virtual CPU (vCPU) to one CPU. **HS** applies both the S/MISN and MIMN scalability patterns. Each Docker container is constrained to run only one Node.js instance; its resource is limited to one vCPU and sufficient memory, which enables one instance to run in one process, taking only one CPU core and partial memory from the underlying VM level. **CM** uses only the S/MISN scalability pattern. The CM does not allow a worker process to be forked in a separate node; instead, the master process and the worker process must run together in the same node, which limits the CM to run in a single node. For this reason, the MIMN pattern that involves multiple nodes is not suitable to be applied to the CM strategy. In this group, a larger Docker container with an equal number of CPU cores and memory size equal to the underlying VM is created, in which single or multiple worker processes are forked. When we conduct scalability experiments, we runs the executor

**Table 3: Experimental Conditions and Conducted Tests**

| | S/MISN | MIMN |
|---|---|---|
| **HS** | Baseline, CPU-Int., Net-Int. | Baseline, CPU-Int., Net-Int. |
| **CM** | Baseline | |
| **UP** | CPU-Int., Net-Int. | |

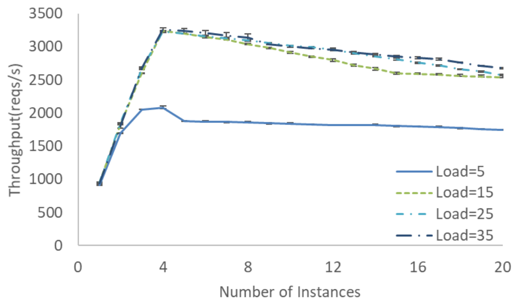


**Figure 2: Baseline App. Throughput (S/MISN)**

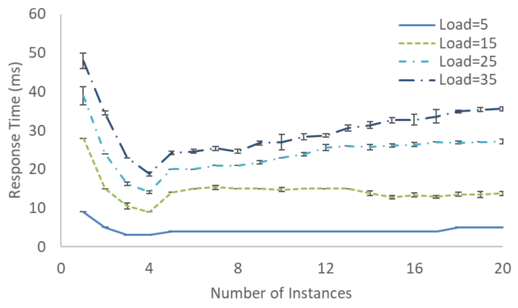


**Figure 3: Baseline App. 95% Resp. Time (S/MISN)**

in a client host isolated from the server cluster to ensure JMeter is not a bottleneck impacting the validity of experimental results. Within the server cluster, the RC runs in the manager node; and the benchmark service stack is only deployed and run in worker nodes. When the executor performs the load testing, it lasts 100 seconds and the thinking time is set to zero to significantly stress the system. Each experimental condition (Table 3) was repeated 15 times; the following graphs display averages with standard deviations as error bars.

## 5.2 S/MISN Pattern Analysis

We applied the S/MISN pattern on the CPU-intensive, network-intensive and baseline test applications, using the HS scaling strategy. A single worker node was utilized for these experiments with 4 vCPUs and 4GB of RAM; the network-intensive application was tested with 8 vCPUs to further exclude CPU being the bottleneck.

**Baseline:** Figure 2 illustrates the baseline benchmark's throughput as a function of the number of isolated Node.js instances when workloads of 5, 15, 25, and 35 parallel clients were launched; similarly, Figure 3 presents this benchmark's response time. As the workload increased, so did the response time for any number of Node.js instances. However, throughput behaved differently: it rose as more instances and more workload was added, reached a peak

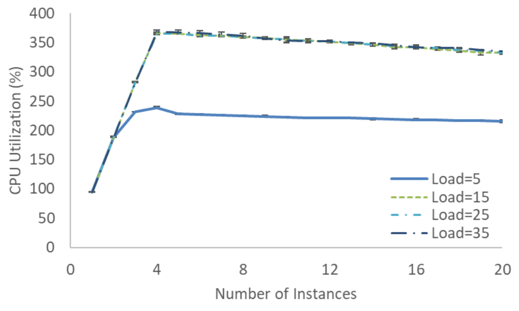


**Figure 4: Baseline App. CPU Utilization (S/MISN)**

and then started degrading. According to Little's Law, $L = \lambda W$ under a steady state, the number of queued requests ($L$) is equal to the rate at which requests arrive ($\lambda$) multiplied by the time a request takes to process ($W$) [13]. Essentially, for a given number of clients ($L$), throughput ($\lambda$) and response time ($W$) are inversely proportional, which is consistent with our results. In terms of varying the number of instances, the best performance was attained by four instances, which exactly matches the number of available CPUs. With fewer instances, the system remains underutilized; whereas with more, the Node.js processes compete against each other—consider OS-related delays due to scheduling and context switching as well as CPU-related delays due to cache misses.

Figure 4 demonstrates the baseline benchmark's CPU utilization, which corroborates with our previous observations: When there are four instances spawned, the CPU utilization reaches the peak (approximately 366% of 400%) and then starts to decrease. The baseline benchmark run is CPU-intensive as it runs a full application server. With more than four instances, the four CPUs are saturated and the CPU resource becomes a bottleneck.

We also measured the baseline application's network bandwidth—we skip plotting the results for brevity as they are similar to the CPU utilization graph. Similarly to the previous metrics, when the number of instances exceeds four, limited CPU resources cannot support any more requests in parallel. Furthermore, increased contention further degrades the application's network bandwidth. Because, the baseline test application does not involve large data transmissions, the overall network bandwidth is less than 1.5 MBps; the network bandwidth is not a performance bottleneck in this case.

**CPU-intensive:** Figure 5 and Figure 6 illustrate the throughput and response time of the CPU-intensive test application. Throughput reaches the maximum and response time the minimum when four Node.js instances are spawned, which is equal to the number of available hardware processors. Figure 7 show CPU-utilization resource usage results for the CPU-intensive test application—network traffic data were also collected but are not plotted for brevity as they look similar with the throughput results. It can be seen that the CPU utilization reaches the peak (370% out of 400%) when there are four Node.js instances created and then it keeps a flat trend. Moreover, the CPU utilization diagram becomes a cluster when launching workload of 15, 25, and 35. This indicates the CPU resource has been saturated and cannot support more workloads; therefore, the CPU resource is the bottleneck. In turn, this causes retrograde throughput, response time and network traffic at the

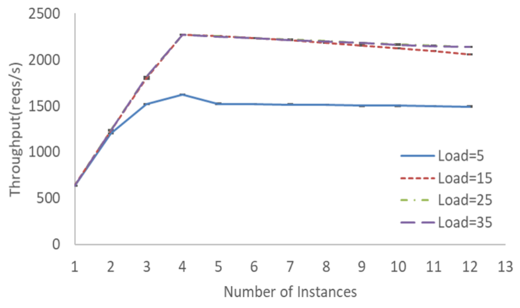


**Figure 5: CPU-intensive App. Throughput (S/MISN)**

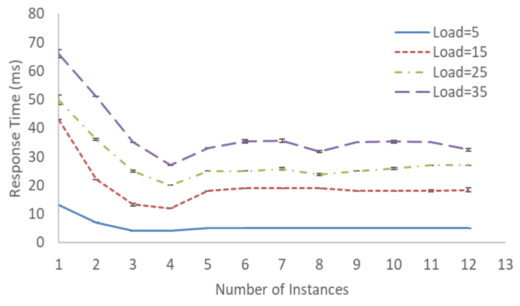


**Figure 6: CPU-intensive App. 95% Resp. Time (S/MISN)**

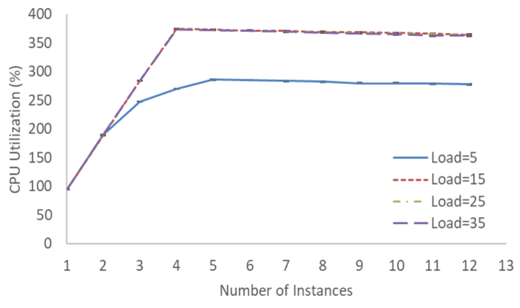


**Figure 7: CPU-intensive App. CPU Utilization (S/MISN)**

point that a specified number of Node.js instances saturate the CPU resource. In addition, the CPU-intensive test application's CPU utilization is slightly higher than the baseline benchmark's. The reason is the CPU-intensive test application processes more computation, running a greater number of loops, than the baseline test application. The response time of the CPU-intensive test application is longer than that of baseline benchmark run due to the CPU-intensive test application taking longer time to process a single request, consequently, it results in lower levels of throughput in comparison to the baseline.

**Network-intensive:** Figures 8, 9 and 10 display throughput, response time and resource usage for the network-intensive test application—we skipped plotting the application's network throughput since the results look similar to those of the throughput graph. This time, we utilized a VM with 8 vCPUs and 4GB of RAM to run the Docker containers. The network-intensive test application

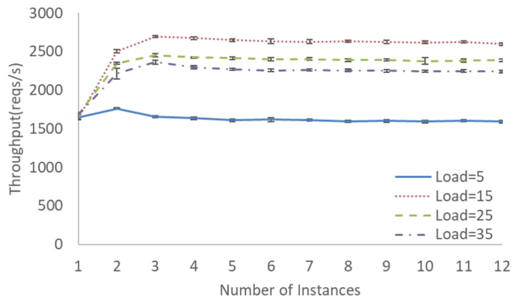


**Figure 8: Network-intensive App. Throughput (S/MISN)**

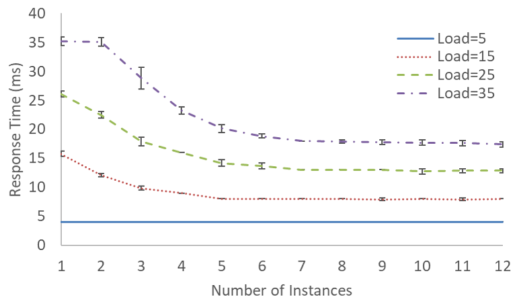


**Figure 9: Network-intensive App. 95% Resp. Time (S/MISN)**

stresses and targets the outgoing network bandwidth by responding to clients with a large data size. Network bandwidth is a critical resource in this case. The host VM has more CPU resources, eliminating any CPU-related bottlenecks and targeting the network bandwidth resource. Throughput is maximized when the workload is 15 parallel clients, whereas, workloads of 25 and 35 degrade the throughput. Additionally, throughput starts to degrade when there are three instances within the workload of 15, 25 and 35. In contrast, at the workload of 5 parallel clients, throughput becomes retrograde at the point where two instances are spawned, because the low workload does not significantly stress the network bandwidth. On the other hand, response time becomes longer with a greater workload. We find that there is a directly proportional relationship between throughput and response time and is consistent with Little's Law, when the workload increased from 5 to 15 parallel clients. Afterwards, the relationship of throughput and response time becomes inversely proportional. The reason is that workloads that exceed 15 client, overload the server and make it unsteady, which causes hardware in the network, such as switches, to become congested, resulting in the server accepting fewer requests. In addition, we find that this case violates Little's Law as the system does not keep a steady condition when it is overloaded.

According to the resource usage diagrams, the CPU utilization of the network-intensive test application is lower than that of baseline and CPU-intensive test applications and does not even exceed half of the maximum possible (800%), while its outgoing network bandwidth is greater. Thus, CPU is not the performance bottleneck in this case: CPU utilization keeps increasing with greater number of instances; there is little CPU resource contention and the CPU resource is not saturated when adding more instances in this

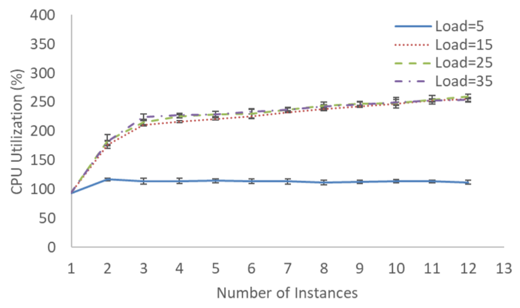


**Figure 10: Network-intensive App. CPU utilization**

system. In addition, performance does not start to degrade when eight instances are created in the system, even though all CPU cores are utilized by a number of instances. This implies the CPU resource suffices to support more than eight instances. However, the network is the performance bottleneck in this case. The network resource is saturated by instances and makes hardware in the network congested. The network congestion limits the data transferred to the client side, which results in retrograde overall performance. The network-intensive test application consumes much more network bandwidth and reaches the peak when approximately three instances are spawned. Afterwards, it accordingly keeps constant, which matches the trend of throughput and also verifies the network being the performance bottleneck. We checked other resource usage data collected by the *Ibenchjs*'s RC component, such as disk I/O throughput and RSS, and found none of them was a bottleneck.

*5.2.1 HS vs. CM.* Next, we performed a comparison between the HS and CM scalability strategies on a VM with 4 vCPUs and 4GB of RAM, utilizing the baseline test application. We collected performance and resource usage metrics within different workloads while scaling out to four instances/worker processes–equal to the number of vCPUs, which resulted in the best performance as discussed earlier. Figures 11, 12, and 13 demonstrate a histogram of maximum throughput, response time and CPU utilization as a function of the various workloads. The red bar represents the CM strategy and blue bar denotes the HS strategy. Our results indicate that when the underlying CPU resource has been fully utilized by multiple Node.js processes, the HS strategy registers significantly higher throughput and shorter response time, outperforming CM. Moreover, the CM strategy utilizes more CPU resources to support its performance than the HS strategy. CM uses Node.js' Cluster module, which uses a master process for managing a set of worker Node.js processes as well as load balancing requests to the round-robin. These tasks are intensive, which apparently caused the additional CPU utilization for CM. Apart from these tasks, the master process also works in parallel with other worker processes to handle the workload. However, the master process handles less workload than worker processes due to its additional task overheads, which degrades the overall performance. In contrast, under the HS strategy, all the instances of the Node.js application do not process the load balancing task and the worker process maintenance as the master process does. A specialized load balancer is installed by Docker Swarm and utilized in the manager node, which does not compete for CPU

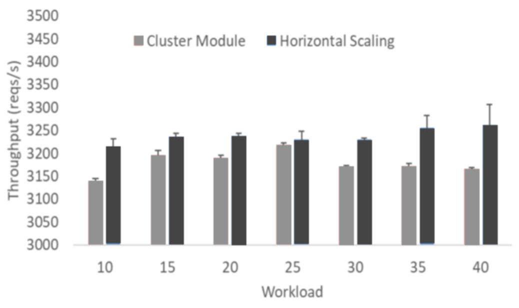


**Figure 11: Baseline App. Throughput of HS vs. CM**

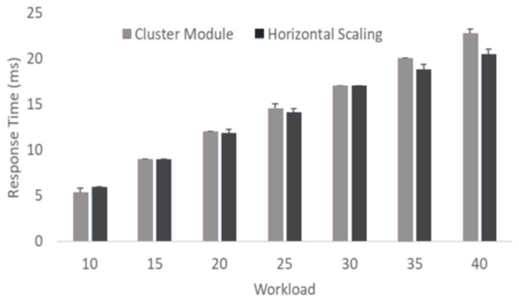


**Figure 12: Baseline App. 95% Resp. Time of HS vs. CM**

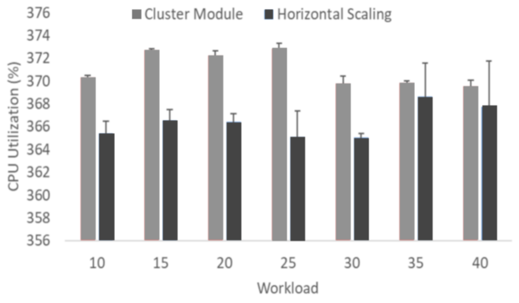


**Figure 13: Baseline App. CPU Utilization of HS vs. CM**

resources with instances deployed in worker nodes. Consequently, it ensures all instances deployed in worker nodes focus on handling concurrent workload from clients without processing additional tasks. Therefore, the native cloud scaling HS scaling strategy—in the embodiment of Docker Swarm and its load balancer—outperforms the CM scaling strategy of Cluster, i.e. forking more Node.js instances and using one of them as a manager. Nevertheless, CM does improve the system's performance, albeit not as well as HS.

*5.2.2 UP Strategy.* To investigate the common hypothesis that Node.js does not scale up well by adding resources to a fixed number of instances, we applied the UP strategy on the CPU-intensive and network-intensive test applications. A single worker node was utilized for these experiments with 8 CPUs and 8GB of RAM; exactly one Node.js instance was started and was progressively given from 1 to 8 vCPUs.

We display the results of the CPU-intensive application in Figure 14 and of the Network-intensive application in Figure 15, both as

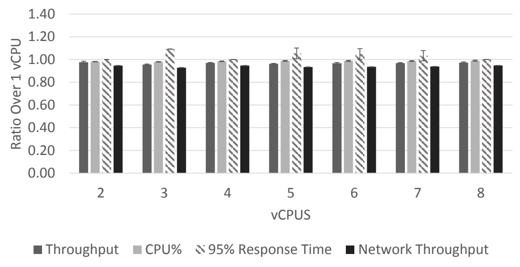


**Figure 14: UP Results of CPU-Intensive Application**

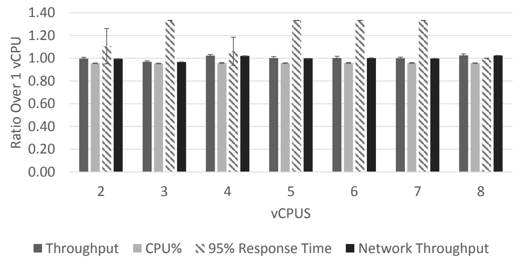


**Figure 15: UP Results of Net-Intensive Application**

ratios over the 1-vCPU case. As expected, throughput and response time of both applications remained the same—or even deteriorated due to increased OS and CPU contention—as more vCPUs were added. The single-threaded nature of Node.js was not able to utilize the presence of extra cores; the CPU utilization floated around 100%, i.e. exactly one core fully utilized, for all cases. Consequently, scaling Node.js by adding more cores is not an effective strategy.

## 5.3 MIMN Pattern Analysis

We extend the number of hosts from a single to multiple nodes to conduct a set of experiments using the CPU-intensive and the network-intensive test applications. We collected throughput, response time and resource usage to investigate the scalability effects and determine the performance bottleneck on each case.

**CPU-intensive:** Figure 16 illustrates throughput of the CPU-intensive test application for different workloads when experiments ran on various numbers of nodes (one to four). The results demonstrate that throughput scales according to the varying number of nodes. However, it does not scale linearly with respect to the workloads, according to the different cluster sizes. For example, considering when running within one-node cluster, 2275 requests per second were satisfied, but within four nodes, 7527 requests/second. Throughput within four nodes under linear scaling would be four-time that of one node, and the expected value should had been 9100 requests/second. However, the measured value is 7527 requests/second, which is approximately only three times greater than that of one node. Table 4 lists the maximum throughput for different cluster sizes from one to four nodes under a maximum supported workload, and accordingly, the number of instances created. Non-linear scalability effects are revealed from these results.

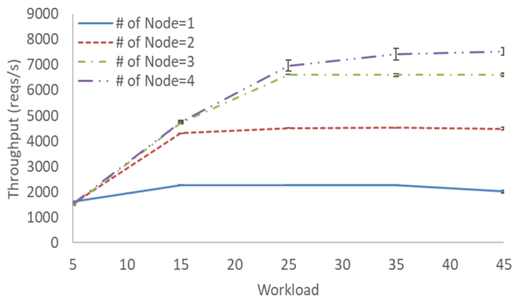


**Figure 16: CPU-intensive App. Throughput (MIMN)**

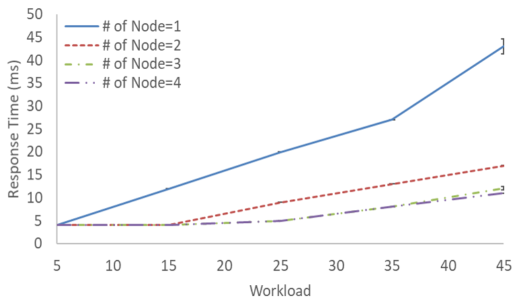


**Figure 17: CPU-intensive App. 95% Resp. Time (MIMN)**

**Table 4: Maximum throughput of CPU-intensive test application achieved for various cluster sizes**

| #Nodes | Max Thr. (reqs/s) | #Inst. at Max Thr. |
|---|---|---|
| 1 | 2274.73±1.64 | 4 |
| 2 | 4522.90±5.53 | 8 |
| 3 | 6615.73±1.36 | 12 |
| 4 | 7527.21±157.26 | 14 |

Figure 17 illustrates the response time of the CPU-intensive test application. These results indicate that response time keeps constant with varying workloads until increasing towards exponential trend at a particular workload due to the system overloading. Additionally, response time starts to increase at the same point where throughput begins to become retrograde—as expected by Little's Law. In addition, when we add more nodes to the cluster, response time becomes shorter within a fixed workload. However, response time also follows a non-linear scalability pattern, because the benefits of achieving short response time become smaller, instead of accordingly bigger, with adding more nodes in the cluster.

We also measured and collected CPU resource usage (Figure 18), and incoming/outgoing network bandwidth (omitted for brevity as it looks similar to the throughput and CPU% graphs) to identify performance bottlenecks. Considering the characteristics of the CPU-intensive test application, the CPU resource is the performance bottleneck, which is verified by Figure 16. The CPU utilization grows until reaching a peak under a particular workload. From Table 4, we can see the number of instances created to support the maximum workload and achievable throughput. For example, when we create 12 instances running on the three-node cluster, we can see the system achieves the maximum throughput (6615.73

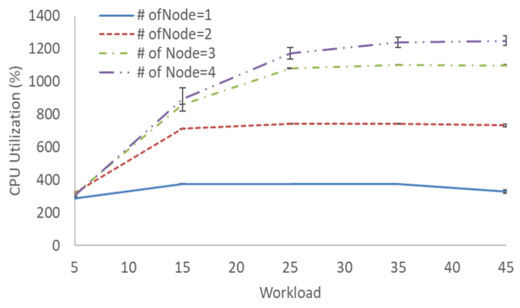


**Figure 18: CPU-intensive App. Total Cluster CPU Utilization (MIMN)**

requests/second on average). The three-node cluster was equipped with 12 vCPUs; therefore, all these 12 instances saturate the CPU and the utilization reaches the maximum capacity of computation. Consequently, it achieves the best performance. For this reason, the network incoming and outgoing bandwidth begins to decline at the same point where the CPU utilization begins to decrease.

**Network-intensive:** Figure 19 illustrates the maximum throughput of the network-intensive benchmark for workloads varying from 5 to 45 parallel clients, within different sizes of the cluster. Studying Figure 19 reveals that throughput does not scale well with a greater number of nodes for each launched workload. Thus, the benefits of obtaining a maximum throughput are smaller with extending the cluster size—scaling out. The system achieves its maximum throughput with a workload of 15 parallel clients. When workloads that exceeded 15 were launched, throughput became retrograde. Figure 20 shows the minimum response time as a function of workloads. From Figure 20, no obvious scalability effect of response time with a greater number of nodes for different launched workloads emerges; the relationship between throughput, response time, and the workload is similar with that of S/MIMN scalability pattern. For smaller workloads, throughput and response time are consistent with Little's Law being directly proportional to the workload: when workloads increase, both throughput and response time are greater. Once the workload exceeded 15 clients, throughput was inversely proportional to response time with greater workloads. The reason is the system becomes saturated with more than 15 clients, which leads into performance bottlenecks.

Figure 22 shows the maximum CPU utilization as a function of various workloads when running in varying sizes of our Docker Swarm cluster-cloud. Figure 22 indicates that the CPU resource is not the performance bottleneck, because this system has a rather low CPU utilization for each launched workload, when it runs in different cluster sizes—the maximum CPU utilization was at most 200% out of 2000%. Figure 23 presents the maximum network bandwidth and reveals a similar scalability effect with that of throughput. We find a network bandwidth bottleneck at the workload of 15 clients, which matches the throughput trend. Beyond the workload of 15, the network bandwidth starts to decline. We also checked other computing resource usage data and the result is the same as the S/MIMN scalability pattern: disk I/O and RAM are not performance bottlenecks. Node.js itself cannot be the performance bottleneck either. Moreover, we have confirmed that this test application does

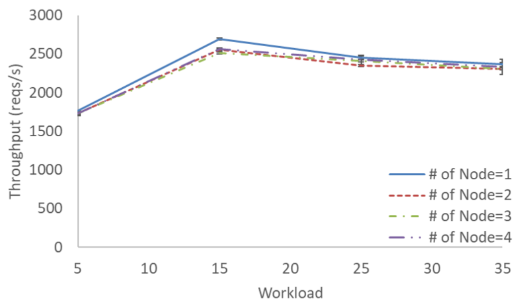


**Figure 19: Network-intensive App. Throughput (MIMN)**

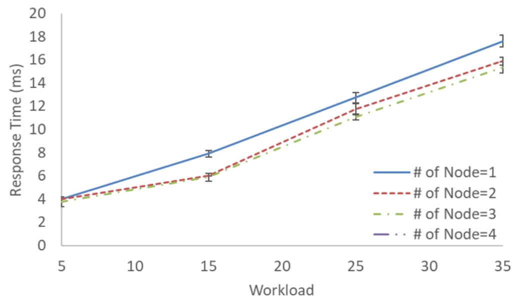
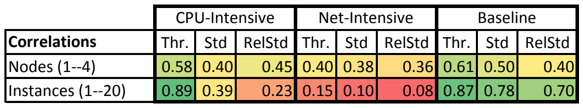


**Figure 20: Network-intensive App. 95% Resp. Time (MIMN)**



| | CPU-Intensive | | | Net-Intensive | | | Baseline | | |
|---|---|---|---|---|---|---|---|---|---|
| **Correlations** | Thr. | Std | RelStd | Thr. | Std | RelStd | Thr. | Std | RelStd |
| Nodes (1--4) | 0.58 | 0.40 | 0.45 | 0.40 | 0.38 | 0.36 | 0.61 | 0.50 | 0.40 |
| Instances (1--20) | 0.89 | 0.39 | 0.23 | 0.15 | 0.10 | 0.08 | 0.87 | 0.78 | 0.70 |

**Figure 21: Color-Coded Throughput Correlations (MIMN)**

not have contention of the CPU resources, but network-related contention. This observation is crucial as Node.js is best suited for handling network I/O due to its non-blocking and asynchronous nature—this design pattern allows other requests from clients to be served while waiting for an I/O operation to complete.

**On Bottlenecks:** We present the correlations per testing application between the number of nodes and instances, and the average, standard deviation and relative standard deviation of throughput (Figure 21). The results show that the CPU-intensive and baseline applications benefited more by adding instances; whereas, the network-intensive, by adding cluster nodes. Furthermore, the network-intensive application was virtually not affected by adding instances, as the near-zero correlations indicate. Regarding the system's quality of service, which is best highlighted by lower absolute and relative standard deviation values, strong connections appear particularly for the baseline application by adding more instances.

Considering all of our experimental data, we make the following predictions regarding causes of Node.js performance bottlenecks: 1) A poor network configuration of Linux TCP/IP connection in nodes has a significant influence on network performance and in turn affects the scalability of the whole system. 2) Concurrent requests from clients are dispatched by a built-in load balancer to different Docker containers hosting web servers over the overlay network.

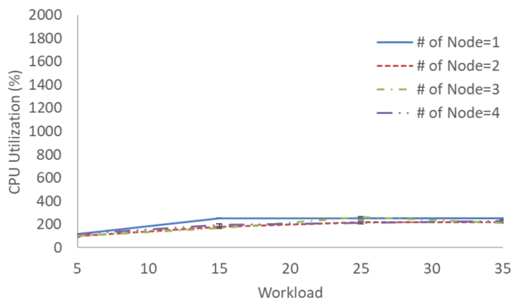


**Figure 22: Network-intensive App. Cluster CPU% (MIMN)**

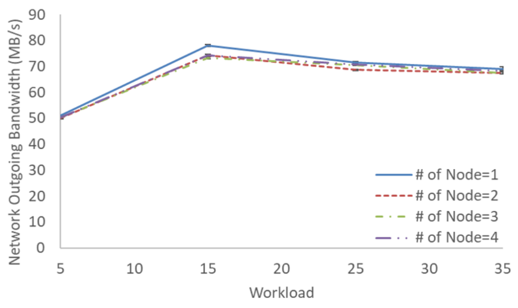


**Figure 23: Network-intensive App. Total Cluster Outgoing Network Throughput (MIMN)**

The internal load balancer can be a bottleneck and will not be able to serve beyond those concurrent requests from clients. We use the load balancer that is integrated in the manager node. When creating more Docker containers, extending the cluster size, or increasing workloads, the load balancer will be overloaded and degrade the network performance. 3) Hardware in the network, such as switches, network interface cards (NICs) and links can affect the network performance with a greater workload. For example, transferring the data consumes much network bandwidth and in turn leads to exceeding the maximum capacity of links between the client and the server. The NIC and the multi-port switch can also overload and produce congestion with an increased workload, which results in retrograde network performance.

## 5.4 Regression Analysis

Finally, we perform a regression analysis to fit USL and Amdahl's Law scalability models in the term of software scalability for the CPU- and network-intensive test applications. We calculate model coefficients, which can identify scalability characteristics. Figure 24 and Figure 25 respectively illustrate fitting the USL and Amdahl's Law model on the maximum throughput of the CPU- and network-intensive applications against workloads under a fixed hardware configuration. Dots denote measured data points; solid lines denote the USL and Amdahl's Law model fitted to sample data. Both plots demonstrate the effects of diminishing returns in terms of achieving better performance for greater numbers of workloads. We calculated their associated coefficient values of the USL and Amdahl's Law model by performing regression and summarize them in Table 5.

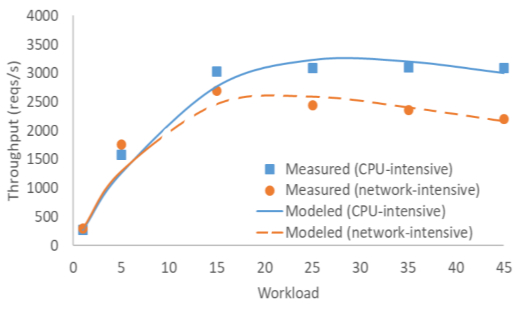


**Figure 24: USL Model of CPU- and Network-intensive App. Max Throughput**

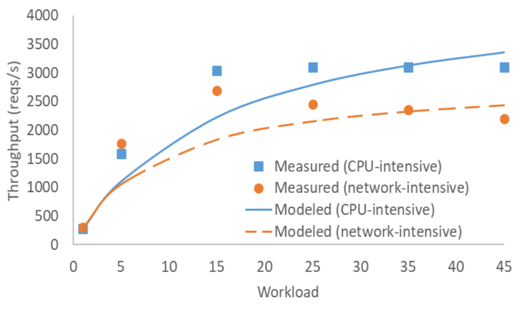


**Figure 25: Amdahl's Law Model of CPU- and Network-intensive App. Max Throughput**

**Table 5: Coefficient Values of Scalability Models**

| Coefficients | CPU-intensive | | | Network-intensive | | |
|---|---|---|---|---|---|---|
| | $\sigma_c$ | $\lambda_c$ | $R_c^2$ | $\sigma_n$ | $\lambda_n$ | $R_n^2$ |
| **USL** | 0.017 | 0.0012 | 98.76% | 0.0282 | 0.002 | 99.24% |
| **Amdahl' Law** | 0.0614 | N/A | 93.15% | 0.1023 | N/A | 93.43% |

Figures 24 and 25 denote that performance scales not linearly but sub-linearly, when higher workloads are launched. This is determined by the non-zero coefficient values; in other words the contention (referred to as $\sigma$) and the coherency (referred to as $\lambda$) affect the scalability of the system. Moreover, USL provides a good fit for both types of test applications due to a high value of R-square, whereas the Amdahl's Law model has a worse fit. Both types of test applications create multiple instances to support maximum throughput, causing significant communication overheads within the system. However, Amdahl's Law does not consider the interprocess communication overhead, which results in worse fitting than the USL model. Furthermore, all coefficient values of both models for the network-intensive test application are greater than those of the CPU-intensive test application. Thus, the network-intensive test application has had a higher contention and coherency than the CPU-intensive application.

## 6 CONCLUSION

We introduced *Ibenchjs*, a scalability-oriented benchmark framework for Node.js, and a set of sample resource-intensive test applications. These applications stress different types of computing resources, which we confirmed experimentally. *Ibenchjs* leverages them—although, it can be easily extended to include any Node.js application—to perform scalability benchmarks. Several suitable metrics, throughput, response time, and resource consumption were measured to enable a scalability analysis by investigating their effect depending on the system's configurations.

To eliminate performance interference present in multitenant clouds, we built locally a private and isolated cloud with six physical hosts using Docker Swarm and Vagrant stacks, which provided a scalable, distributed and multicore environment for our scalability experiments. We defined two types of scalability patterns depending on the deployment of containerized applications: S/MISN refers to deploying single or multiple application instances in a single node; MIMN refers to deploying multiple instances in multiple nodes. Based on these scalability patterns, we initialized *Ibenchjs* and conducted scalability experiments using the CPU-, network-intensive and baseline test applications in our private cloud.

We find: 1) Scaling up Node.js by increasing the number of CPUs is ineffective. 2) For CPU-intensive and baseline test applications, the CPU resource is critical; adding more application server instances does not necessarily improve the performance, because these additional instances saturate the CPU. When we scaled the cluster by adding more nodes, we observed sub-linear effects; therefore, the benefit of adding a greater number of nodes becomes smaller due to a heavy network communication overhead between nodes. 3) Regarding network-intensive test applications, we found the network resource is critical. With launched workloads and the number of instances increasing in a single node, more network bandwidth was consumed until reaching a maximum capacity and becoming a bottleneck. When we performed the scalability analysis based on the MIMN scalability pattern, we found the performance of the whole cluster does not have a proportional improvement by adding more nodes. Based on these, we made several predictions in aspects of hardware in the network, Docker Swarm overlay network, and network configuration of nodes. We concluded that building a good network environment—including hardware, topology configuration of TCP/IP connection—is necessary to achieve a performance improvement, instead of just adding more nodes or instances of applications. 4) We compared our CPU- and network-intensive applications via a scalability-model regression analysis. We obtained non-zero coefficients and concluded that the scalability of both test applications is sub-linear as the workload increases due to contention and coherency. Finally, 5) we compared two scalability strategies applied to Node.js, native cloud horizontal scaling (HS) and Node.js Cluster module forking (CM) on the S/MISN pattern: when the underlying CPU resource has been fully utilized by multiple Node.js processes, HS significantly outperformed CM. Cluster's master process performs load balancing and worker-process maintenance, tasks that fine-tuned PaaS-based solutions of HS are explicitly designed to perform.

Future work includes extending our scalability investigation to the remaining resource-intensive applications. Moreover, several more complicated real-world Node.js test applications will also work with *Ibenchjs* to investigate scalability in the cloud. Additionally, our predictions on the scaling failure of network-intensive test applications and narrow down network bottlenecks can be further experimentally tested. Finally, *Ibenchjs* can be extended to run in other PaaS clouds, such as Cloud Foundry.

## ACKNOWLEDGMENTS

This research was conducted within the Centre for Advanced Studies – Atlantic, Faculty of Computer Science, University of New Brunswick. The authors are grateful for the colleagues and facilities of CAS Atlantic in supporting our research. The authors would like to acknowledge the funding support of the Natural Sciences and Engineering Research Council of Canada (NSERC), 501197-16. Furthermore, we would also like to thank the New Brunswick Innovation Foundation for contributing to this project.

## REFERENCES

[1] 2018. Apache JMeter. Retrieved May, 2018 from http://jmeter.apache.org/
[2] 2018. Docker. Retrieved May, 2018 from https://www.docker.com/
[3] 2018. Docker Swarm Overview. Retrieved May, 2018 from https://docs.docker.com/swarm/overview/
[4] 2018. Node.js. Retrieved May, 2018 from https://nodejs.org/en/
[5] 2018. Node.js Express Module. Retrieved May, 2018 from https://expressjs.com/
[6] 2018. Node.js v9.4.0 Documentation Cluster Module. Retrieved May, 2018 from https://nodejs.org/api/cluster.html#cluster_cluster_schedulingpolicy
[7] Stavros Aronis, Nikolaos Papaspyrou, Katerina Roukounaki, Konstantinos Sagonas, Yiannis Tsiouris, and Ioannis E Venetis. 2012. A scalability benchmark suite for Erlang/OTP. In *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang workshop*. ACM, 33–42.
[8] Inderveer Chana and Nidhi Jain Kansal. 2012. Cloud load balancing techniques: A step towards green computing. *International Journal of Computer Science Issues (IJCSI)* 9, 1 (2012), 238.
[9] Neil J Gunther. 2007. *What is guerrilla capacity planning?* Springer.
[10] Neil J Gunther. 2008. A general theory of computational scalability based on rational functions. *arXiv preprint arXiv:0808.1431* (2008).
[11] Neil J Gunther and Raj Foreword By-Jain. 2000. *The practical performance analyst.* Authors Choice Press.
[12] Kai Lei, Yining Ma, and Zhi Tan. 2014. Performance comparison and evaluation of web development technologies in php, python, and node. js. In *Computational Science and Engineering (CSE), 2014 IEEE 17th International Conference on.* IEEE, 661–668.
[13] J. D. Little and S. C Graves. 2008. *Little's law.* Springer, Boston, MA. 81–100 pages.
[14] Peter Mell, Tim Grance, et al. 2011. The NIST definition of cloud computing. *Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg* (2011).
[15] Panagiotis Patros, Dayal Dilli, Kenneth B Kent, and Michael Dawson. 2017. Dynamically Compiled Artifact Sharing for Clouds. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on.* IEEE, 290–300.
[16] Panagiotis Patros, Kenneth B Kent, and Michael Dawson. 2017. SLO request modeling, reordering and scaling. In *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering.* IBM Corp., 180–191.
[17] Panagiotis Patros, Stephen A MacKay, Kenneth B Kent, and Michael Dawson. 2016. Investigating resource interference and scaling on multitenant PaaS clouds. In *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering.* IBM Corp., 166–177.
[18] B Schwarz. 2015. Practical Scalability Analysis with the Universal Scalability Law.
[19] Wei-Tek Tsai, Yu Huang, and Qihong Shao. 2011. Testing the scalability of SaaS applications. In *Service-Oriented Computing and Applications (SOCA), 2011 IEEE International Conference on.* IEEE, 1–4.
[20] Lloyd G Williams and Connie U Smith. 2004. Web Application Scalability: A Model-Based Approach.. In *Int. CMG Conference.* 215–226.
[21] Jiapeng Zhu. 2018. *A Scalability-oriented Benchmark Suite for Node.js in the Cloud.* Master's thesis. University of New Brunswick, Fredericton, NB, Canada.