FitScale: Scalability of Legacy Applications Through Migration to Cloud

Jinho Hwang(✉), Maja Vukovic, and Nikos Anerousis

IBM T.J. Watson Research Center, New York, USA
{jinho,maja,nikos}@us.ibm.com

Abstract. One of the key benefits of Cloud computing is elasticity, the ability of the system infrastructure to adapt to the workload changes by automatically adjusting the resources on-demand. Horizontal scaling refers to the method of adding or removing resources from the resource pool. As such it is appealing to enterprises who seek to migrate their legacy systems as it requires no application rewrite or refactoring. Vertical scaling approach offers a mechanism to maintain continuous performance while reducing resource cost through reconfiguration of the resource. The challenge is, however, in being able to automatically identify the right size of the target resource such as a VM or a container. Moreover, choice of scalability policies is not intuitive due to application complexity, topology and variability in system performance parameters that need to be considered.

This paper presents a transformation model, FitScale, which provides scalability with minimum price of resources. The paper describes the framework that employs the application functional and operational properties to recommend the target sizing and scalability policies. We evaluate proposed approach in an on-premise and cloud environments, with a dataset of 2023 servers hosting 6737 applications. The experimental results show about 5 times cost reduction with minimum performance impact.

Keywords: Cloud · Migration · Elasticity · Scalability

1 Introduction

Application development in the cloud typically follows the micro-services style, an approach where applications are built from the composition of smaller atomic services, each one running independently in the cloud and communicating through REST APIs. Microservices [15] are designed to represent distinct business functions, and are deployed independently[1].

A question that often comes up is how existing applications, developed in the "traditional" style, and running on previous generation systems can take advantage of cloud platforms, or even become native to the cloud themselves.

[1] http://martinfowler.com/articles/microservices.html.

Q.Z. Sheng et al. (Eds.): ICSOC 2016, LNCS 9936, pp. 123–139, 2016.
DOI: 10.1007/978-3-319-46295-0_8

The latter would imply a significant transformation of the application architecture, including a breakdown of many application sub-components into independently running micro-services. The benefits are many and significant: agility for development and deployment; reduced cost in OS/middleware management, ability for applications to leverage cloud native services.

When considering transformation of legacy applications to Cloud environments there are a number of choices available. On one end of the spectrum is the plain migration (the so called "lift and shift"), which moves the application into one functional entity such as, a server [9] or a container [16] to mimic a micro-service architecture. On the other end of the spectrum is the full transformation to a micro-services architecture (by refactoring and rewriting the application). Given that the latter is both a time and resource-consuming task, a more reasonable approach is to perform a like-to-like migration first (the application is moved in its entirety to one or more virtual machines in the cloud in a way that resembles its original topology). After the migration is completed, the more involved transformation to a micro-services architecture can be performed in increments over a period of time.

As legacy applications are often not built to be scalable in themselves, setting the right scalability after migrating into clouds is still a delicate, time-consuming process. There are a number of challenges in ensuring the scalability of legacy applications during migration to Cloud:

- The root cause of why scalability is required is often unclear. For example, is that due to cpu, memory, or network? If one of them is a bottleneck all the time, then we may buy more of that resource and achieve both horizontal and vertical scalability.
- Auto scaling policies should be defined well to meet performance expectation based on the usage patterns at the source. For example, AWS has its own policy syntax.
- "Like-to-like migration" to Cloud does not take advantage of the benefits provided by Cloud such as elastic scalability.
- In most cases, it is hard to decide whether or not applications should be put in a scaling group.
- It is hard to estimate the properties of initial resources.
- It is challenging to identify the right scaling policy.

In this paper we discuss an approach to migrating applications to the cloud to meet scalability requirements using elastic compute services in the cloud. Elasticity can be achieved through carefully selected policies to meet application demands.

There are three key objectives that drive design of our approach to policy-based scalability: (a) minimize the cost of the resources during scalability, (b) ensure high-performance is maintained, with low-to-no degradation impact and (c) maintain stability of the application through correctly chosen right initial size (thus avoiding the slow-downs during scaling). This in itself is a challenging endeavor as policies are automatically selected and fixed, and any subsequent changes require reconfiguration.

This paper makes the following three contributions:

- TheFitScale framework for application transformation to Cloud with consideration for scaling requirements
- Method for pattern discovery for application topology and performance
- Method for target sizing and scaling policy assignment

2 Background and Motivation

Scaling horizontally (or scaling out/in) means adding more compute nodes to (or removing nodes from) a system. As an example a Web server may be scaled out from one compute node to three. As compute prices have dropped and performance continues to increase, high-performance computing applications such as seismic analysis and biotechnology workloads have adopted low-cost "commodity" systems for tasks that once would have required supercomputers. System architects may configure hundreds of small computers in a cluster to obtain aggregate computing power that often exceeds that of computers based on a single traditional processor. The development of high-performance interconnects such as Gigabit Ethernet, InfiniBand and Myrinet further fueled this model. Such growth has led to demand for software that allows efficient management and maintenance of multiple nodes, as well as hardware such as shared data storage with much higher I/O performance. Size scalability is the maximum number of processors that a system can accommodate [5]. To scale vertically (or scale up/down) means to add resources to (or remove resources from) a single node in a system, typically involving the addition of CPUs or memory to a single computer. Such vertical scaling of existing systems enables them to use virtualization technology more effectively, as it provides more resources for the hosted set of operating system and application modules to share. Taking advantage of such resources is also referred to as "scaling up", such as expanding the number of Apache daemon processes currently running. Application scalability refers to the improved performance of running applications on a scaled-up version of the system.

There are tradeoffs between these two models. A larger numbers of computers implies increased management complexity, as well as a more complex programming model and issues such as throughput and latency between nodes; also, some applications do not lend themselves to a distributed model. In the past, the price difference between the two models has favored "scale up" computing for those applications that fit its paradigm, but recent advances in virtualization technology have blurred that advantage, since deploying a new virtual system over a hypervisor (where possible) is often less expensive than actually buying and installing a physical one. Configuring an existing idle system has always been less expensive than buying, installing, and configuring a new one, regardless of the model.

Still, reconfiguration is challenging for a number of reasons: unknown or inaccurate information about source environment, ability to reason and make

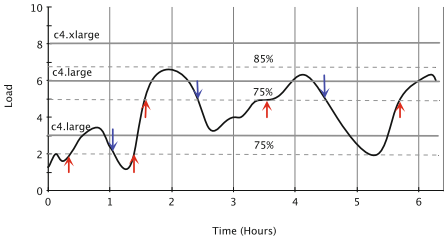


Fig. 1. Two instances have different compute resources and result in different prices based on request patterns (synthetic graph).

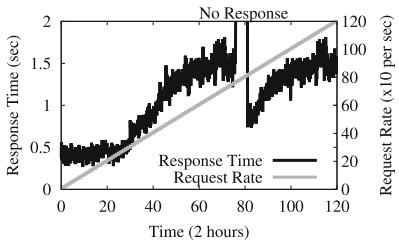


Fig. 2. Scaling up has a provisioning delay while adding an additional application (i.e., virtual machine).

decisions about scalability for multi-tier applications, lack of automatic rule and policy generation.

Figure 1 is a simple experiment that depicts cost benefits of allocating applications with the right size of compute resources. Given the request pattern for 6 h, we show two different deployment cases. An application is deployed to *c4.xlarge* virtual machine capable of 8 loads with 85 % threshold, and *c4.large* virtual machine capable of 3 loads with 75 % threshold. We use AWS prices in the graph for the sake of reader's familarity, but the same scenario can be applied for other cloud providers. *c4.xlarge* costs $0.209 per hour, and *c4.large* costs $0.105 per hour. Note that a partial hour of usage is also considered as a full hour. Upward arrows mean scaling up applications on demand (adding an instance), whereas downward arrows mean scaling down applications.

The reactive horizontal scaling is used, so when loads pass the threshold, a new instance is added. The total number of hours used is 16 for *c4.large*, and 6 for *c4.xlarge*. Thus, the total cost of *c4.large* is $\$0.105 \times 16 = \$1.68$, whereas the total cost of *c4.xlarge* is $\$0.209 \times 6 = \$1.254$. *c4.xlarge* provides 33 % less operational cost in this case. Therefore, this corroborates determining the right size is an important factor.

Figure 2 illustrates a simple web server response time with a linear increase of the request rate. "No response" around 70 min incurs because of the provisioning delay, and zeroed memory without cached data when adding an additional application. Therefore, not only minimizing cost, but also choosing the right thresholds is critical to avoid a service level agreement (SLA) violation.

3 FitScale Framework

The FitScale framework integrates multiple, independent, workflow processes that rely on a common data set. With access to source infrastructure or compiled data, FitScale discovers the necessary data and converts unstructured data into structured data such as matrices or comma separated values. Especially graphs with large data points are summarized to help identify moving patterns in the

analysis step. Once the data is converted into structured data, FitScale seeks usage trends of critical resources which are performance bottlenecks, and creates resource plans such as an initial virtual machine size and number of virtual machines in a pool, and defines scaling policies that are appropriate to the target clouds.

Figure 3 depicts the FitScale architecture for end-to-end migration automation. In Step ① a user needs to collect information such as a server list with credentials, infrastructure API key from on-premise data centers, or to install agents depending on the infrastructure discovery tool[2]. At Step ② a user inserts the collected information to FitScale. Step ③ FitScale launches a discovery process with the provided information. This may take a couple of days or weeks depending on the application usage patterns. If the usage patterns are stationary over a short period of time, the discovery process stops and moves on to the next step. Otherwise, it keeps watching the source infrastructure to find the stable application usage patterns. The information collected in this stage are performance (resource usage), application configurations, running processes, and network connectivity. Step ④ in the analysis step, FitScale seeks metrics such as request patterns, resource usage patterns, variability (a rate of scaling up and down), and dependency that are used in the modeling step. At Step ⑤ together with the target information, FitScale optimizes the scalability with the objective to minimize the cost. At step ⑥ the final outputs are resource plans that decide on the size of instances and scaling policies that define the timing of scalability.

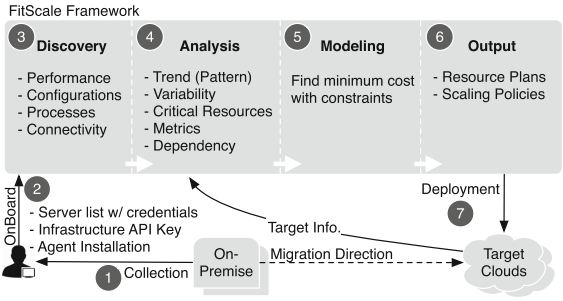


Fig. 3. A FitScale framework concatenates multiple workflow processes to automatically generate scalability plans.

4 Understanding Applications

The first step to understanding application configuration is to define what needs to be discovered. Understanding applications is a complex process, as they run on operating systems with various library dependencies, and operate across multiple network-connected servers. The process of inferencing how applications will

[2] There are agent-based discovery tools [17] and agentless discovery tools (i.e., using scripts) [1].

operate in target Cloud environment relies on information about vertical (local) and horizontal (remote) dependencies, and how resources such as, cpu, memory, disk and network are utilized. In short, to be able to reason about applications we require dependencies and resource usage data. The discovery and analysis phases of the FitScale framework provide these insights about applications.

4.1 Discovery

The discovered data about the source environment needs to be mapped into what are the scalability requirements of the target Cloud. Following are the key data items required:

- resource allocation: cpu, memory, disk, network bandwidth
- resource usage: cpu (%), memory read/write (bytes), disk read/write (bytes), network in/out (bytes)
- network connectivity
- operating system and application configurations

There are number of agent and agent-less tools for system information discovery. In the FitScale framework, we use a script-based approach (agentless) to uncover the necessary configuration and operational data. The shell is the easiest and the most convenient way to program and run. For example, on Linux based systems ./proc directory contains system attributes including process information and system information. Many standardized tools can provide us with key data. *ps* reports a snapshot of the current processes, *df* reports filesystem disk space usage, *nm* lists symbols from object files, *objdump* displays detailed information from object files, *readelf* displays information about ELF object files, *lspci* displays information about PCI buses in the system and devices connected to them, *lsof* provides a list of all open files belonging to all active processes, *ldd* prints the shared libraries required by each program or shared library specified on the command line, *strace* traces system calls and signals, *ltrace* traces library call, and *netstat* prints network connections, routing tables, interface statistics, masquerade connections, and multicast memberships. On Windows-based systems vbscripts can be utilized to collect such data [8].

Servers are connected to each other through network interfaces. A web service may need web applications, databases, file systems, or memory cache servers in the backend. This distributed (micro-service) architecture needs to be taken into consideration to capture the propagation of application processes. The inter-server attributes can help make decisions on simultaneous scalability. To collect the inter-server data, we can look at the network statistics derived from *netstat* information, gather network ports used for communication, and infer performance propagation impact between servers [12].

4.2 Pattern Analysis

Using the data about source infrastructure FitScale provides the following insights:

1. Which resources are bottlenecks? (this helps find key metrics that impact on performance).
2. How variable the processing (request) trend (pattern) is? (the answer helps decide the size of virtual machines).
3. How fast the processing trend increases or decreases? (the answer helps determine the increasing or decreasing rate and timing of virtual machines).
4. Where are request sources from (geographical analysis based on the connections)? (the answer helps decide how/where global scalability is requred).
5. How is a multi-tier application distributed? (the answer helps find how much load propagation impacts when loads increase).

In addition, we can easily derive simple insights. For example, the questions like how many (virtual) CPU cores/memory/disk/network bandwidth are assigned or how many virtual machines are used in the source are rather straightforward computations.

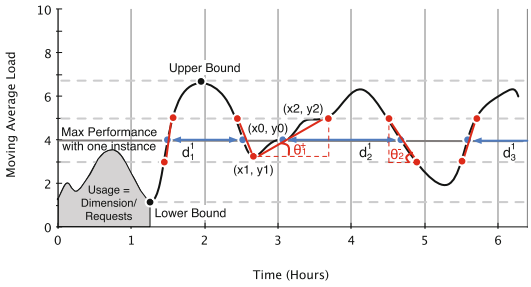


Fig. 4. Sample CPU usage graph (synthetic) to illustrate how to analyze data in FitScale.

The answers are quantified, and translated into scalability decisions to create scalability policies or determine the size of target resources. Figure 4 illustrates a sample graph $y = f(x)$ that shows a certain trend (pattern) along the x-axis and y-axis to explain how the answers to the questions can be made from the data (resource usage) observation. Note that we do not use this graph to formulate an optimization problem, but use its properties to draw useful information. Since the usage graph fluctuates in a very short time period, we use the moving average, a form of average which has been adjusted to allow for smoothing of a time series. Moving average smoothing is a smoothing technique used to make the long term trends of a time series clearer [3]. We use a simple moving average technique denoted as $\frac{1}{n}\sum_{i=0}^{n-1} x_{M-i}$, where n is the averaging window size, and M is current time. The following numbers correspond to the questions earlier.

1. To find which resources are bottlenecks, we look at the overall resource usage of CPU, memory, disk, and network when user requests arrive in applications. Given a fixed period of time T with the number of requests per second r_i, each resource usage is calculated as $u(t) = \frac{\sum_{i=0}^{T} m_i}{\sum_{i=0}^{T} r_i}$, where m is a resource

usage (%) per second (i.e., measurement unit), and t is a resource type. In other words, $u(t)$ is an average resource usage (%) per request, and this is a metric to decide which resources are potential bottlenecks.

2. The variability can be interpreted as a rate of scaling up and down, so the high variability means there are more adding/removing servers. The variability of the graph does not need to be statistically analyzed since we only need to know how much the graph is fluctuating based on the maximum performance line of a target virtual machine [4]. The variablity of this graph f is $v(f,t) = \frac{|\{x|f(x)\overset{+}{=}l(t)\}|}{H}$, where $l(t)$ is the maximum load of the instance type t, an operator $\overset{+}{=}$ represents an increasing intersection when load increases, H is the total number of monitoring hours. In Fig. 4, the middle line (at load 4) represents the maximum performance line with one instance type, and the total time H is 6. Therefore, $v(f,t) = \frac{3}{6}$, meaning that in a time unit (1 h), the resource usage surpasses the maximum performance line 0.5 times.

3. The speed of increasing or descreasing load decides the change rate of the number of virtual machines and also the timing of when to scale up or down. We measure the moving speed by calculating the slope of the change between critical ranges. The critical ranges mean the lines above/below of the maximum performance line and left/right of the crossing-point. In Fig. 4, the critical ranges are defined as 25 % above (load 5) and below (load 3) of the maximum performance line (load 4), and 0.5 h left/right. The slope of slope triangles (red line with θ) shows the speed of increasing loads. Therefore, the average speed is $s(f,t) = \frac{\sum_{i=0}^{n} sin(\theta)}{v(f,t)}$, where n is the number of crossings with $|\{x|f(x) \overset{+}{=} l(t)\}|$, and $sin(\theta) = \frac{opposite}{hypotenuse} = \frac{y_2 - y_1}{\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}}$. Likewise, the decreasing loads can be calculated the same way.

4. The geographical distribution of request sources provides a useful information on how/where the scalability should happen. For example, some vendors allow to scale up the number of virtual machines across multiple data centers based on the geographical requirements. In this case, we can locate additional servers close to where requests are generated. The source network addresses of requests provide geographical distribution.

5. A multi-tier (distributed) application consists of applications that communicate each other to process a request and generate a response. For example, a web server talks to a database server to retrieve information. This directional dependency is important because the scalability is done in just one tier, meaning the scaling up of the web server does not necessarily lead to the scaling up of the database. Therefore, we have to consider the impact of other tiers when we design scalability policies. The application dependency is the same as the network dependency, thus the network connectivity between applications is captured.

Note that FitScale takes heed of dependency among connected servers because often the connected servers need to scale together and the backend server (not scaling elastically) should be able to accommodate maximum loads

from the frontend servers (scaling elastically). While the most of applications are formed in a multi-tier architecture (for example, service oriented architecture, and micro-services architecture), considering request flows in a scaling group should not be obviated in order not to countervail the scalability configurations.

5 Understanding Clouds

Cloud providers offer different ways to scale applications elastically, and therefore it is important to understand the differences in scalability models. This section describes scalability offerings by major cloud providers and how users are charged.

5.1 Scalability

The first step towards elastic scalability is to monitor instances at the platform level. Most cloud providers provide average metrics for every 5 min, and optionally more fine-grained monitoring with additional charges. CloudWatch, AzureWatch, and Nimsoft are examples. The monitored metrics are checked against policies in every monitoring period. Once one of user-defined policies is triggered from the monitoring engine, the actions defined by the policies initiate scalability processes (add or remove instances). When adding instances, cloud providers automatically attach the new instances to a load balancer to forward requests to them.

Table 1 shows some scalability examples of major cloud providers. There are two ways to provision instances when thresholds are surpassed: on-demand and pool-based. The on-demand provisioning adds instances by replicating a running instance when needed. This approach is cost efficient and does not have (state) synchronization problem, but is expected to be slow. On the other hand, the pool-based provisioning pre-previsions instances and uses them when needed (usually keep them in stopped mode). This approach expects fast addition of instances, but renders waste of resources and may result in synchronization problem because it is replicated when an application is initially created.

5.2 Scalability Cost

Cloud providers charge users either hourly or monthly for running instances, called the pay-as-you-go model. Prices for the scalability are varied. In fact, users are not charged by the scalability function itself, but by the number of used instances. A unit of charging a running instance for scalability is only per-hour, and a partial hour of usage is also charged the same as a full hour. Intuitively, frequent scaling up and down can charge more than just running more instances without elastic scalability for the same period of time. Therefore, it is imperative to find an optimal scaling model that can provide a minimum price and sustain performance.

Table 1. Cloud providers have different scalability metrics and provisioning strategy. The queue length metric scales based on how many messages are waiting in the queue.

Cloud provider	Scaling metrics	Actions	Provisioning strategy
SoftLayer (IBM)	CPU percentage, Private network incoming/outgoing (Mbps), Public network incoming/outgoing (Mbps)	Add/remove/set scale group by quantity/percentage	On-demand
AWS (Amazon)	CPU utilization (%), Disk reads (Bytes), Disk read operations (Operations), Disk writes (Bytes), Disk write operations (Operations), Network in/out (Bytes)	Add/remove/set scale group by quantity/percentage	On-demand
GCP (Google)	CPU utilization (%), HTTP(S) load balancing serving capacity (%), Cloud Monitoring metrics (80 metrics in https://cloud.google.com/monitoring/api/metrics) also network load balancing (for other protocols such as SMTP) is applicable preprovisioned managed instance group	Add/remove/set scale group by quantity/percentage	Pool-based
Azure (Microsoft)	CPU (%), Queue length	Add/remove/set scale group by quantity/percentage	Pool-based

6 Scaling Model

The two main objectives are (1) finding the right resource size that minimizes cost and guarantees performance and (2) creating policies based on the observation from the discovery and pattern analysis in order to scale at the right moment with the right size. Specifically, the outputs of the model are:

- Instance size: using an auto scaling function usually does not involve any pricing unless augmenting monitoring capabilities, but the instance (resource) size is directly related to the final cost, so it is important to define the right size.
- Maximum number of instances: instances are either pre-provisioned in the resource pool or provisioning on the fly with the maximum capacity definition.
- Scaling policy: depending on the performance (cpu, memory, disk, network), we need to scale up/down the service. The properties that need to be found are threshold (%), watching time (minutes) above the threshold, and scaling size unit (# of instances or % of the scaling group).

We have input information from the discovery and the pattern analysis from Sect. 4: bottleneck resources, upper-bound/lower-bound loads, load variability, the speed of increasing/descreasing, geographical distribution, and dependency.

To achieve the two main goals mentioned earlier, we need to find the minimum cost while guaranteeing performance, which can be translated into the question of how we set up a scalability configuration at the target cloud because the size of

instance and the scalability policies directly affect the operational expenditure. First of all, we need to define how we calculate the cost. The cost function is

$$C(f,t) = \sum_{i=0}^{M(f,t)} \sum_{j=0}^{|d^i|} p(t) * d^i_j(t) + p(t) * H, \tag{1}$$

where f is the observed function for a particular bottlenek resource type (for example, cpu, memory, disk, or network) with the number of data points D, t is the target instance type (for example, c4.large with cpu, memory, disk, network), $M(f,t)$ is the maximum number of instances of type t used in the function f, $|d^i|$ is the number of segments for d^i, $p(t)$ is the price of the type t, and H is the total number of monitoring hours. The time complexity to find the cost for each instance type is bounded by the data points of the graph because it goes through all data points to find crossing points. Thus, the time complexity is $O(D)$. The objective function to minimize cost for scalability is:

$$\begin{aligned} &\underset{t \in T}{\text{argmin}}\ C(f,t) \\ &\text{s.t } f \in F, \text{ and} \\ &M(f,t) \times L(t) \geq U(f), \end{aligned} \tag{2}$$

where the cost function $C(f,t)$ is defined in Eq. (1), T is the list of all instance types, F is the list of observed functions with resource types, $L(t)$ is the load of the instance type t, and $U(f)$ is the uppper bound. The time complexity to find the best t is $O(|T| \cdot D)$.

Now that we know the size of the target instance with the type t from Eq. (2), and the maximum number of instances, we need to define policies. As shown in Table 1, scalability policies need threshold (%), watching time (minutes) above the threshold, and scaling size unit (# of instances or % of the scaling group). Furthermore, we can configure the availability zone for geographical distribution. The threshold determines when we start triggering the resource usage alarm to further make decisions on scaling up and down. It is often defined as a percentage of the total resource capacity. For example, if CPU utilization goes over 80 %, we can count down to see whether the load sustains for pre-defined sustaining time.

We derive the threshold according to how fast loads reach the maximum capacity of the instance and how long it takes to provision an instance. While the increasing speed is defined in Sect. 4.2 with a parameter θ, the average (increasing) time to reach the maximum performance line from the threshold is noted as $\tau^+ = \frac{\sum_{i=0}^{|\theta^+|} hypotenus \times cos(\theta_i^+)}{|\theta^+|}$, where $|\theta^+|$ is the total number of increasing loads, and *hypotenus* is the slope of the slope triangle, $\sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$. Therefore, the average (increasing) time should be within boundary of the provisioning time. The condition is noted as

$$\tau^+ > B(t), \tag{3}$$

where $B(t)$ is provisioning time (or booting time for the pool-based provisioning strategy) of the instance type t. However, since in reality, some workloads may increase too fast to meet this condition, we predictively provision instances even when small increase incurs. This means that FitScale sets a very low threshold and a short watching time. There are prior arts that use the predictive and autonomous scalability [6,11], so we do not focus on the predictive methods in this paper.

When the threshold is set, we need to configure the watching time starting when loads cross upward the threshold. The watching time is reset when loads cross downward the threshold. The watching time should also satisfy Eq. (3) in order to trigger to spin up more instances for increasing loads.

To scale up/down, we define a scaling size as either the number of instances or the percentage of the scaling group. We use the average number of expected instances from the pattern analysis. We count the added number of instances appended to the scaling group every time loads increase, then find the expected (average) number of instances added each time, and use that number as a scaling size.

If users are international, data centers distributed across globe can help serve requests better when applications are deployed close to users. Availability zones can be used to define geographical locations where the scalability is required. FitScale simply recommends potential data centers that can be included in the availability zones based on the network address of users.

7 Evaluation

Note that we do not intend to compare cloud platform performance of cloud providers since there are many prior arts to compare performance [10,13]. Instead, we focus on how transformation with appropriate scalability can help applications make use of scalability provided by clouds. We first case study applications of a real enterprise (on-premise) data center to analyze discovered data (Sect. 7.1). Then, we run experiments on the real cloud provider with the outcome from FitScale (Sect. 7.2). Lastly, we perform cost and performance analysis (Sect. 7.3).

7.1 Case Study: Legacy Data Center

We study a real (on-premise) data center with 2023 servers (physical and virtual), running 6737 applications to see some of pointers made in previous sections. Figure 5 depicts a cumulative density function of normalized resource usage to show how resources are used. CPU seems to be the most under-utilized resource and only about average 6 % is used. This is mainly because the CPU is a time sharing resource, which can go down to 0 % when it is not used. We have observed that CPU is heavily used in a certain period of time during day and night (up to 100 %). Disk is also under-utilized because users often reserve space for future uses. However, memory is well utilized because lots of cache data are stored both from applications and from operating systems.

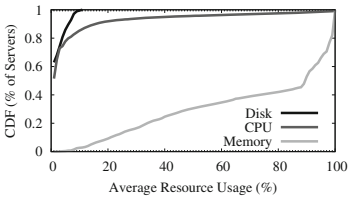


Fig. 5. CDF with resource usage shows CPU/disk is under-utilized, but memory is well utilized.

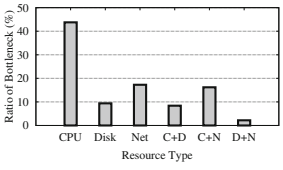
Table 2. 5 min average resource usage statistics of a machine (AVG = average, STD = standard deviation).

Resource	AVG	STD	MIN	MAX
CPU (%)	6.598	17.653	0	100
Network in (KB)	111851	172954	0	869880
Network out (KB)	89543	166628	0	829791
Disk read (KB)	53814	2149716	0	502722620
Disk write (KB)	53209	895461	0	395700840

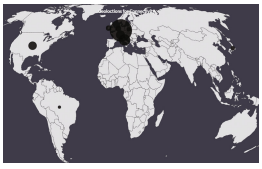
Additionally, Table 2 shows some resource metrics (used in scalability policies) and some statistical results. As mentioned, CPU has low average number, but maximum usage shows that CPU is used heavily in a certain time period. Network and disk are also consistently used (average), and saturated in a certain period of time (maximum).

As previously shown in Table 1, cloud providers have different scaling metrics that can be used as a monitoring/triggering metric. It is worthwhile to identify what resource type is the most bottlenecked in the real data center. Figure 6(a) illustrates ratios of bottleneck resource types. Even though average CPU is unterutilized as depicted in Fig. 5, more than 43 % of servers have CPU as a bottleneck. The next highest bottleneck is the network as 17 %, and only 9 % of servers have disk as a bottleneck. The combination of resource types show some insights that C+N (CPU+Net) has high correlation, meaning when the network traffic increases, CPU load increases together. The rest of servers (about 30 %) not shown in any of bottleneck do not have any load changes.

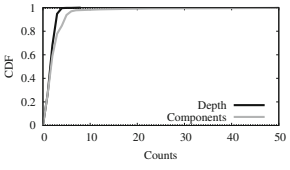
While global companies maintain on-premise data centers across continents, it is important to decide how to scale out across geo-distributed locations. The geo-locations can be configured with scalability policies, and it is important to see where requests come from. Geographical distributions show where requests originate, which can determine the need for geographical scalability. Figure 6(b) depicts the datasets have most of requests from European countries (Italy, German), and some US.



(a) Ratios of bottlenecked resource.



(b) Geographical distributions.



(c) Multi-tier apps with components with depth.

Fig. 6. Experimental results from legacy data center.

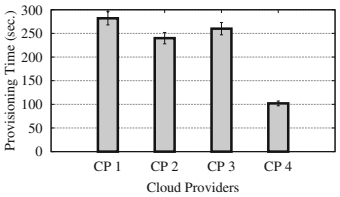


Fig. 7. Provisioning time of VMs in cloud providers (CP = Cloud Provider).

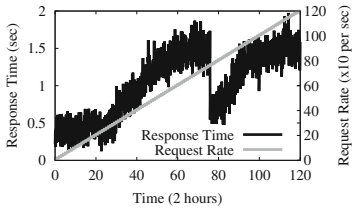


Fig. 8. Scaling experiment in the real cloud.

Business applications run as a multi-tier application spanning multiple servers with functional separation. In this multi-tier application, the high load may be propagated along server dependencies, thus it is important to figure out topological dependencies among servers. Figure 6(c) shows a statistical summary with CDF to show the number of components (instances) and depths (layers) of multi-tier applications. 98 % of multi-tier applications have less than 10 servers, and its depth is less than 7. The average number of components is 3, and the maximum is 105 with depth 1 (this was a monitoring server). The average number of depth is 2 and the maximum was 8.

7.2 Study in the Wild

As explained in Sect. 4, provisioning time may countervail benefits of elastic scalability due to slow responsiveness, and this is shown in Fig. 2. Therefore, FitScale takes into consideration the provisioning time when making policies. Figure 7 shows provisioning time of cloud providers, which spins up a small virtual machine with 2 CPUs, 4 GB memory, and 25 GB local disk. This corroborates the provisioning time is an important factor to consider in order to avoid any performance violation.

We repeat the same experiment as in Fig. 2 to see whether FitScale creates a good scalability policy for increasing loads. A simple web server with a database access in the backend is deployed in the cloud, and httperf is used to generate HTTP requests [14]. Figure 8 illustrates how well FitScale can create a scalability policy based on the source observation. The main difference between the two experiments is the level of threshold, which differentiates the provisioning time of each case.

7.3 Study in Laboratory

To expand the experiments to more diverse topologies than just two nodes, we take common topologies from the observation of the on-premise data center. Since 85 % of multi-tier applications consists of less than 5 servers as shown in Fig. 6(c), we consider up to 5 servers in each multi-tier application.

Figure 9 shows 5 common topologies observed from the on-premise data center. Black circles are able to scale horizontally and white circles are only able to scale vertically, meaning only resource size (# of CPUs, memory, etc.) can be adjusted due to application limitations.

We take multi-tier applications' patterns (topology, resource allocation/usage) from the on-premise data center dataset (Sect. 7.1) to simulate the scalability. The experiment is done in the Xen hypervisor in the local machine with 24 cores (Xeon CPU X5650 2.67 GHz) and 32 GB memory. The sample request pattern (12 h) in Fig. 10(a) is from the dataset, and it is already smoothed with 5 min moving average. We compare FitScale with like-to-like (LTL) case, and aim to validate cost benefits and performance violation.

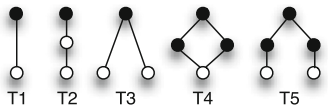
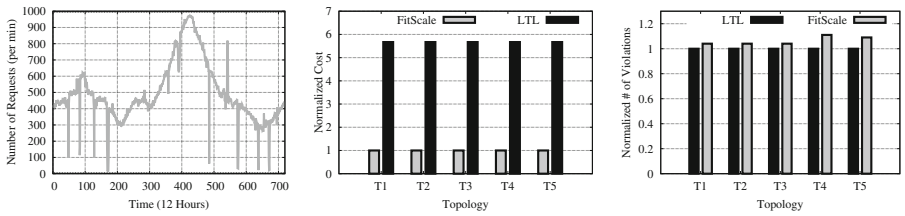


Fig. 9. Five common topologies of multi-tier applications.

FitScale picks virtual machines based on the minimum cost as shown in Eq. (1). Figure 10(b) shows a ralative cost of each topology that sets FitScale as a baseline. In all cases, LTL needs 5 times more operational expenditure and this continues to increase over time (note that we only look at 12 h). Reducing instance costs could result in a performance violation, but as shown in Fig. 10(c), the violation is minimum. We observe that performance violations occur because of initial memory caching (application behavior), not because of scalability policies.



(a) Request graph (12 hours). (b) Cost benefits. (c) Performance violations.

Fig. 10. Performance evaluation with sample traffic by comparing with like-to-like (LTL) migration.

8 Related Works

Gallant et al. [5] survey relate work on scalability and propose a classification of elasticity methods based on four characteristics: scope (infrastructure or application), policy (manual or automatic, reactive or predictive), purpose (performance, cost, energy, capacity) and method (replication, redimensioning or migration). In the context of application scalability they note PaaS platforms, such as Aneka [2], where new container instances are executed to handle increase in the demand. In contrast, in Microsoft Azure, user defines resources used by

applications. Hasan et al. [7] reasons that the current autoscaling policies consider resources from three separate domains, compute, storage and network, are acquired or released on-demand without regard to each other. Moreover, network resources are typically not auto-scaled. They propose a mechanism for an integrated auto-scaling system overcoming the above mentioned limitations. Vaquero et al. [18] present the survey of scalability techniques from PaaS and IaaS perspective. They observe that the Cloud benefits are centered around scalability of resources, and this is chiefly achieved by employing a set of service provider defined rules (that may be customized). Most of the prior art focuses on auto-scaling mechanisms, and our work addresses the gap in transforming the application to a scalable functional unit (such as container).

9 Conclusion

We looked at the problem of workload migration from a legacy environment to the cloud, and specifically the challenge of automatically identifying the right level of scalability, while at the same time minimizing the operational expenditure. This paper introduces the FitScale framework, develops a method for pattern discovery in application topologies and performance, and develops a method for target sizing and scaling policy assignment. FitScale reasons about functional and operational properties of applications, and derives the target sizing recommendation, coupled with the scalability policies. We evaluated FitScale in an on-premise data center with a dataset of 2023 servers/6737 applications. The experimental results show about 5 times cost reduction with minimum performance impact. As the scalability needs to continue to adapt to changing demands, our future work will focus on identifying how to predict future demands and deliver adequate recommendations.

References

1. Bai, K., Ge, N., Jamjoom, H., Jan, E., Renganarayana, L., Zhang, X.: What to discover before migrating to the cloud. In: 2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), Ghent, Belgium, May 27–31, 2013, pp. 320–327 (2013)
2. Calheiros, R.N., Vecchiola, C., Karunamoorthy, D., Buyya, R.: The aneka platform and qos-driven resource provisioning for elastic applications on hybrid clouds. Future Gener. Comput. Syst. **28**(6), 861–870 (2012). http://dx.doi.org/10.1016/j.future.2011.07.005
3. Casella, G., Berger, R.L.: Statistical Inference, vol. 2. Duxbury Press, Pacific Grove (2002)
4. Chiang, R.C., Hwang, J., Huang, H.H., Wood, T.: Matrix: achieving predictable virtual machine performance in the clouds. In: 11th International Conference on Autonomic Computing (ICAC 14), USENIX Association, Philadelphia, PA, pp. 45–56 (2014). https://www.usenix.org/conference/icac14/technical-sessions/presentation/chiang

5. Galante, G., de Bona, L.C.E.: A survey on cloud computing elasticity. In: Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing, UCC 2012, pp. 263–270 (2012). http://dx.doi.org/10.1109/UCC.2012.30
6. Gandhi, A., Dube, P., Karve, A., Kochut, A., Zhang, L.: Adaptive, model-driven autoscaling for cloud applications. In: 11th International Conference on Autonomic Computing (ICAC 14), USENIX Association, Philadelphia, PA, pp. 57–64 (2014). https://www.usenix.org/conference/icac14/technical-sessions/presentation/gandhi
7. Hasan, M.Z., Magana, E., Clemm, A., Tucker, L., Gudreddi, S.L.D.: Integrated and autonomic cloud resource scaling. In: 2012 IEEE Network Operations and Management Symposium (NOMS), pp. 1327–1334. IEEE (2012)
8. Hwang, J.: Towards beneficial transformation of enterprise workloads to hybrid clouds. IEEE Trans. Netw. Serv. Manag. **PP**(99), 1 (2016)
9. Hwang, J., Huang, Y.W., Vukovic, M., Anerousis, N.: Enterprise-scale cloud migration orchestrator. In: 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM), pp. 1002–1007, May 2015
10. Hwang, J., Zeng, S., Wu, F., Wood, T.: A component-based performance comparison of four hypervisors. In: 2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), pp. 269–276, May 2013
11. Hwang, J., Wood, T.: Adaptive performance-aware distributed memory caching. In: Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13), USENIX, San Jose, CA, pp. 33–43 (2013). https://www.usenix.org/conference/icac13/technical-sessions/presentation/hwang
12. Jermyn, J., Hwang, J., Bai, K., Vukovic, M., Anerousis, N., Stolfo, S.: Improving readiness for enterprise migration to the cloud. In: Proceedings of the Middleware Industry Track, pp. 5:1–5:7. Industry papers, ACM, New York (2014). http://doi.acm.org/10.1145/2676727.2676732
13. Li, A., Yang, X., Kandula, S., Zhang, M.: Cloudcmp: comparing public cloud providers. In: Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement, pp. 1–14. ACM (2010)
14. Mosberger, D., Jin, T.: httperfa tool for measuring web server performance. ACM SIGMETRICS Perform. Eval. Rev. **26**(3), 31–37 (1998)
15. Newman, S.: Building Microservices. O'Reilly Media Inc., Sebastopol (2015)
16. Slominski, A., Muthusamy, V., Khalaf, R.: Building a multi-tenant cloud service from legacy code with docker containers. In: 2015 IEEE International Conference on Cloud Engineering (IC2E), pp. 394–396, March 2015
17. Tivoli-Application-Dependency-Discovery-Manager (2016). http://www-03.ibm.com/software/products/en/tivoliapplicationdependencydiscoverymanager
18. Vaquero, L.M., Rodero-Merino, L., Buyya, R.: Dynamically scaling applications in the cloud. ACM SIGCOMM Comput. Commun. Rev. **41**(1), 45–52 (2011)