Unsupervised Learning Approach for Web Application Auto-Decomposition into Microservices

Preprint *in* Journal of Systems and Software · February 2019

DOI: 10.1016/j.jss.2019.02.031

3 authors:

Muhammad Abdullah
University of the Punjab
5 PUBLICATIONS **1** CITATION
SEE PROFILE

Waheed Iqbal
University of the Punjab
30 PUBLICATIONS **376** CITATIONS
SEE PROFILE

Abdelkarim Erradi
Qatar University
57 PUBLICATIONS **226** CITATIONS
SEE PROFILE

Unsupervised Learning Approach for Web Application Auto-Decomposition into Microservices

Muhammad Abdullah[a], Waheed Iqbal[a,b,*], Abdelkarim Erradi[b]

[a]*Punjab University College of Information Technology (PUCIT), University of the Punjab, Lahore, Pakistan*
[b]*Department of Computer Science and Engineering, College of Engineering, Qatar University, Doha, Qatar*

Abstract

Nowadays, large monolithic web applications are manually decomposed into microservices for many reasons including adopting a modern architecture to ease maintenance and increase reusability. However, the existing approaches to refactor a monolithic application do not inherently consider the application scalability and performance. We devise a novel method to automatically decompose a monolithic application into microservices to improve the application scalability and performance. Our proposed decomposition method is based on a black-box approach that uses the application access logs and an unsupervised machine-learning method to auto-decompose the application into microservices mapped to URL partitions having similar performance and resource requirements. In particular, we propose a complete automated system to decompose an application into microservices, deploy the microservices using appropriate resources, and auto-scale the microservices to maintain the desired response time. We evaluate the proposed system using real web applications on a public cloud infrastructure. The experimental evaluation shows an improved performance of the auto-created microservices compared with the monolithic version of the application and the manually created microservices.

Keywords: Application decomposition, Scalability, Microservices, Web applications, Cloud computing

1. Introduction

Web applications are often deployed on the cloud to offer better performance by auto-scaling the allocated resources to meet specific Service Level Objectives (SLO). The application response time is one the most important SLO from the users perspective. It degrades with the increased number of concurrent user requests mainly due to the saturation of hardware resources including CPU, memory, I/O, and bandwidth, software misconfiguration,

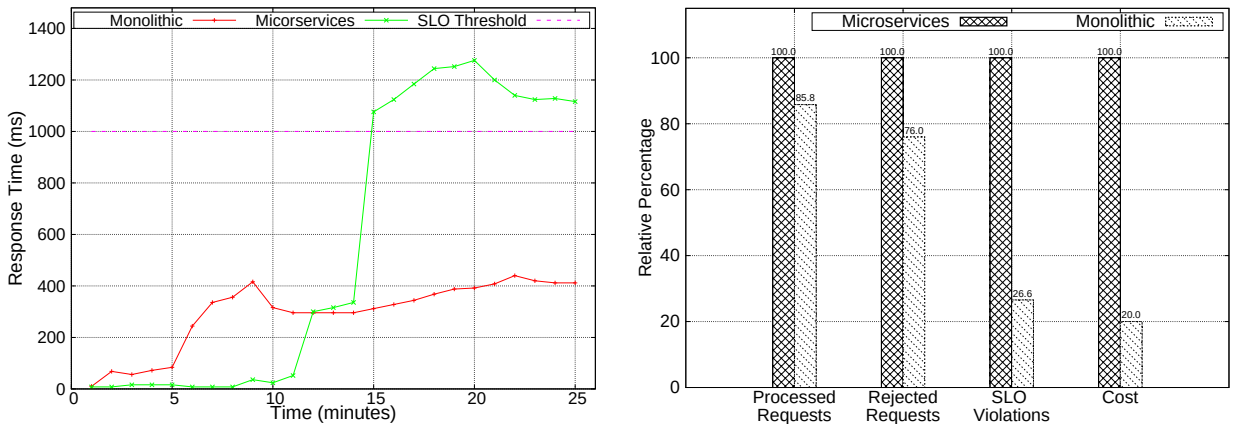
*Corresponding author
Email addresses: phdcsf16m001@pucit.edu.pk (Muhammad Abdullah), waheed.iqbal@qu.edu.qa (Waheed Iqbal), erradi@qu.edu.qa (Abdelkarim Erradi)

and poor implementation. For example, an application requiring complex calculations may cause a bottleneck by saturating CPU resources. Whereas, another application demanding intensive disk operations may introduce a bottleneck due to the saturation of I/O resources. Typically, horizontal scaling is used to maintain the desired response time SLO for applications serving large workloads. Furthermore, the application architecture and the quality of the implementation can significantly improve the application performance and scalability.

One of the most widely used Web application architectures is the monolithic architecture that bundles the user interface, the computation logic, and the data access components into a single executable artifact often deployed as one package on a web server. The web server accepts incoming HTTP requests, executes the request handling logic to produce a response. Over time, monolithic web applications grow large, the complexity of their codebase increases and becomes harder and costly to maintain and scale. Even a small application change requires testing and redeployment of the whole application. Additionally, scaling out a monolithic application to handle a load increase requires deploying the whole package even if only a small subset of its functionality is overloaded.

Recently, the microservices architecture has emerged as an alternative to the monolithic architecture. It decomposes the application into smaller, self-contained, loosely-coupled and replaceable distributed services that implement discrete units of work representing a small subset of related business functions. Such services interact with each other through a well defined API. They can be independently developed, tested, deployed and managed. This allows agile scaling of individual microservices that are heavily used by replicating them on several containers without needing to replicate underutilized services. Moreover, the microservices design aims to increase code modularity and reusability, align services with business capabilities, improve fault isolation, and ease application deployment and scaling. There have been several proposed approaches for functional decomposition of a monolithic web application into microservices and determining the service granularity [1, 2, 3]. However, such manual or semi-automatic white box approaches are mainly based on business domain analysis to decide the service granularity and the service boundaries. They do not consider the impact on the application performance as they are not able to identify the parts of the application that attract high loads and risk of becoming bottlenecks.

To compare the performance of monolithic and microservices architectures, we studied Acme Air [4] open source benchmark web application for a fictitious airline, available in both monolithic and microservices implementations. It comprises five microservices designed and developed by expert software engineers. We analyzed the performance difference between the two architectures. We deployed both implementations on Amazon Web Services (AWS) using `c5.large` instances. The monolithic web application was deployed on one instance and the microservices were deployed on five different instances where each instance contains one microservice. We generated a synthetic workload in increasing fashion to profile the performance of both implementations. We profile the average response time during the study for both implementations and measured performance statistics related to the number of processed requests, the number of rejected requests, and the number of response time SLO violations. The requests that take more than 1,000 milliseconds to produce the output are considered as SLO violations. Figure 1 shows the average response time and



(a) Response Time

(b) Relative Performance Statistics

Figure 1: Comparison of monolithic and microservices implementations of the Acme Air benchmark web application deployed on AWS

the performance statistics for both implementations. We compute the relative percentage for each of the statistics to compare the performance indicators of both implementations. We observed 24% less rejected requests, 73.4% less SLO violations, and 80% less cost for the monolithic implementation compared with the microservices version. However, the microservices implementation processed 14.2% more requests. These results confirmed that the manually created microservices based on software engineering principles do not always yield better performance compared to the monolithic implementation. There are various reasons behind the poor performance of microservices-based applications. In particular, the increased network communications between the individual microservices often through an HTTP resource API increases latency and leads to performance degradation. Another reason is applying software engineering principles to decompose an application into multiple fine-grained services without considering their effects on the application performance. An empirical study with detailed performance analysis by Takanori et al. [5] also concluded that the performance of microservices implementation can be 79.1% lower than the monolithic implementation on the same hardware configuration due to the network virtualization overhead and increased interaction among microservices. A single slow microservice can cause increased end-to-end latency. Although, the microservices architecture has an inherent advantage to more easily monitor and horizontally scale independent microservices. Therefore, we advocate an automatic method to decompose a monolithic application into microservices which can help improve the application performance and scalability compared to manually created microservices designed solely on software engineering principals.

The Cloud offers a wide range of VM types with different configurations including the number of cores, memory, storage, bandwidth, and I/O operations. However, this introduces a challenge to select an appropriate VM to host and deploy the application to maximize the performance with minimal cost [6, 7, 8]. This daunting task to select an appropriate VM type is even harder for microservices-based applications because multiple VMs are required

to deploy the microservices for better performance. A homogeneous type of VMs to deploy microservices would not help in providing better performance with minimal cost as the resource demands for the microservices may vary significantly. Our proposed automatic web application decomposition method helps in identifying appropriate VM types for the microservices.

In this paper, we propose a novel method to automatically decompose a monolithic web application into microservices to improve the application scalability and performance using a black-box approach based on the application access logs and unsupervised machine learning method. The proposed decomposition method also helps to identify the most appropriate VM types to deploy the microservices. Our solution is based on identifying Uniform Resource Identifiers (URIs) groups of a given monolithic application based on the document size and the response time features of the URI requests from the historical access logs. Then the system automatically packages the identified URI partitions into separate microservices and deploy them using the appropriate type of VMs to maximize the application performance. Finally, an auto-scaling method adaptively scales the resources allocated to the microservices to maintain the desired application response time. The main contributions of this paper include:

i. Propose a new method to automatically decompose a monolithic web application into microservices for better performance and scalability.
ii. Propose a method to dynamically select the appropriate resource type to improve the overall performance of microservices-based applications.
iii. Evaluation of the proposed method using a real benchmark web application deployed on public cloud infrastructure.
iv. Compare the performance and cost of the proposed solution with manually created microservices under dynamic resource provisioning method.

There are a few limitations in this work. Firstly, we mainly focused on the decomposition of the application tier into microservices. We do not address the database decomposition into smaller instances. We simply overprovisioned the database tier to avoid saturation during the experimental evaluation. Secondly, we employed a simple reactive auto-scaling strategy to compare the scalability of the the proposed auto-created microservices and the manually created ones. However, advance autoscaling methods can help to further enhance the performance of the proposed system. We plan to address some of these limitations in future work.

The rest of the paper is organized as follows. Related work is presented in Section 2. The proposed approach for auto-decomposition of monolithic application into microservices is discussed in Section 3. The experimental setup is summarized in Section 4. Results and evaluation are presented in Section 5. Finally, conclusions and future work are discussed in Section 6.

2. Related Work

Microservices architecture has emerged as an alternative to the monolithic web application architecture. Several studies proposed approaches for migration and decomposition

of monolithic web applications into microservices. For example, Almonaies et al. [1] use a semi-automated approach for migrating a monolithic web application to service-oriented architecture using software engineering principles. Alessandra et al. [2] present a technique for identification of microservices from an existing monolithic application. The authors map different sub-systems from a monolithic application on a graph to identify the microservices. Micheal et al. [3] presents a model for decomposition of software into smaller services. The authors map software domain model and use cases on an undirected weighted graph and decompose them into a small cluster of services. Andreas et al. [9] propose a decision support system using a fuzzy method based on existing literature and expert feedback to migrate to microservices. Andrei et al. [10] discuss three challenges, multitenancy, statefulness, and consistency of migrating a legacy application into microservices. The work also provides a solution to address these challenges to successfully migrate the application by appropriate refactoring and adaptation of migration techniques. Armin et al. [11] report their experience of migrating a monolithic software architecture to microservices architecture. The work highlights that microservices architecture introduces new complexities and difficulties although it provides high flexibility for scalability and availability. Davide et al. [12] investigate the motivations for migration to microservices. The work reports that scalability and maintainability are the major factors motivating the move to microservices.

Few studies have been done to compare the different application architectures including microservices. For example, Takanori et al [5] compare the monolithic architecture and microservices architecture. The authors demonstrated that the performance of the microservices implementation is significantly lower than the monolithic implementation. This work also characterized the workload for a microservices based web application. Mario et al. [13] compare the infrastructure cost for a web application with different architectures. The authors compared monolithic, microservices, and serverless architectures and concluded that microservices could significantly reduce the operational cost compared to the monolithic architecture. Bruno et al. [14] also discuss the advantages and disadvantages of microservices architecture over the monolithic architecture. Wilhelm et al. [15] present different properties of microservices architecture based deployment of the real e-commerce web application. The work discusses scalability, reliability, and agility of microservices based web applications.

Other works focused on improving the scalability of cloud-hosted applications using microservices architecture. For example, a recent work by Demetris et al. [16] presents DevOps-as-a-Service for the development, deployment, and scalability of applications developed based on microservices architecture and deployed on a containerized cloud environment. Hasselbring et al. [17] discuss the microservices architecture to obtain scalability and reliability for e-commerce web applications. Hanieh et al. [18] propose a workload pattern prediction for microservices using machine learning methods. The authors use multiclass classification and linear regression algorithms for learning and predicting future workload to scale microservices. Gotin et al. [19] use performance metrics of messaging queues to scale microservices hosted on the cloud for IoT applications. The work concludes that using the state of the messaging queue is better than using CPU utilization metric to trigger the scale operations. However, the work is limited to IoT-based applications developed using the microservices architecture. Manuel et al. [20] present a mathematical model for horizon-

tal scaling of microservices application. The work proposed a model to identify a number of replicas to spawn for a specific microservice for better performance. Moreover, machine learning-based methods [21, 22, 23] to scale web applications can also be used to automatically scale microservices. Thus far only few researchers explored approaches to dynamically identify appropriate resource/VM types to deploy and provision web applications. Hector et al. [24] present a decision tree based system for selecting appropriate resources according to the workload and the customer requirements during auto-scaling. Fbio et al. [25] present a framework based on reactive and proactive approaches for auto-scaling. They use different instance types for different services hosted on the cloud to minimize the cost and SLO violations. Mina et al. [26] present an optimization model for effective selection of VM types during auto-scaling to reduce the overall cost. Ming et al. [27] also present an auto-scaling method using appropriate VM types to minimize the cost under changing workloads.

There is no work which automatically decomposes a legacy monolithic application into microservices and then dynamically provisions appropriate resources to maximize performance with minimal cost. Our work reported in this paper proposes a black-box method using application access logs and an unsupervised machine learning method to decompose a given monolithic application into microservices. The proposed solution also identifies appropriate VM types to host the microservices to maximize the overall application performance.

3. Proposed Approach for Auto-decomposition of Monolithic Application into Microservices

Figure 2 shows the overall proposed approach for auto-decomposition of a web application into microservices. The proposed approach consists of three main components.

- The first component of is the "URI Space Partitioning" which identifies a k discrete number of URI groups with similar resource requirements. We propose a clustering method to identify URIs with similar resource consumption using historical access logs. The details of this component are given in Section 3.1.

- The second component is the "Microservices Deployment" which deploys the identified URI partitions as separate microservices without any human intervention. We propose to deploy the monolithic web application code on k different machines and control the requests through a load balancer that dynamically forwards incoming requests to relevant microservices. We explain the proposed approach in Section 3.2.

- The third component is the "Microservices Auto-scaler" which is responsible for dynamically scaling the resources allocated for each microservice. The details of the proposed auto-scaling method are given in Section 3.3.

3.1. URI Space Partitioning

Web applications contain a set of web pages which can be accessed by a Uniform Resources Identifier (URI). The web servers hosting such applications maintain access logs

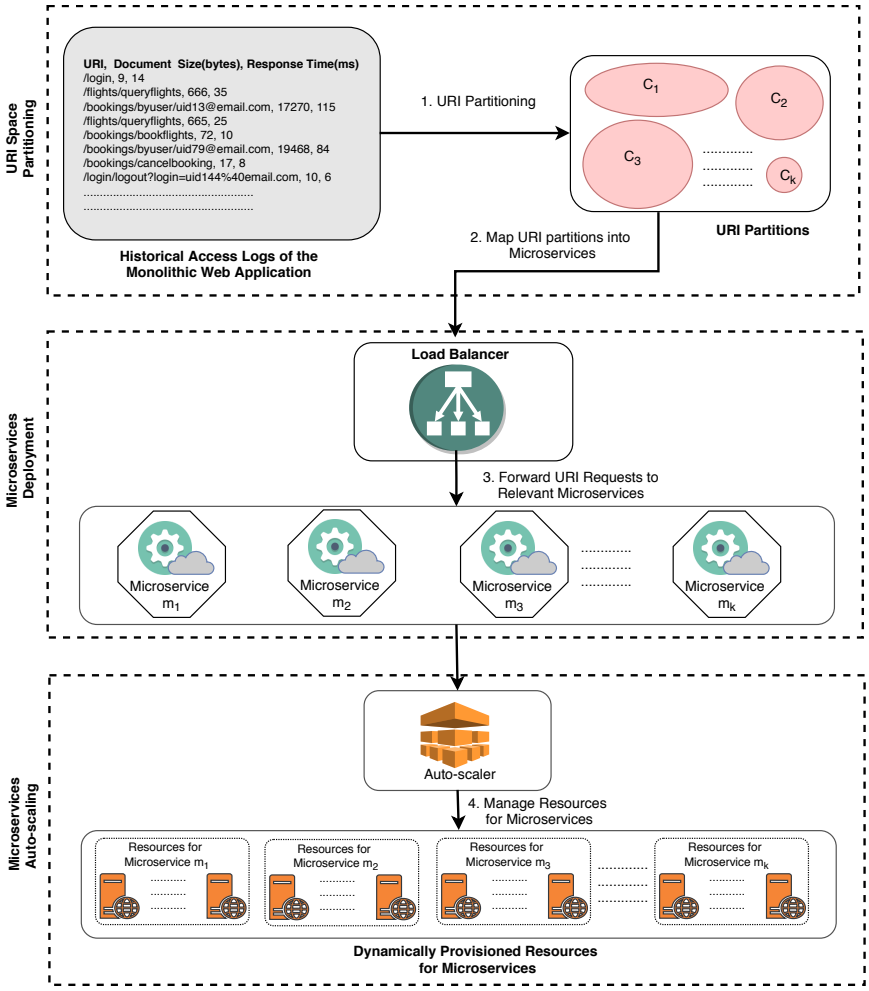


Figure 2: Proposed framework for web application decomposition into microservices using URI space partitioning.The URI partitions are then deployed as separate microservices. Then the auto-scaling controller dynamically provisions resources for microservices.

containing a history of requests for the application URIs. These access logs are used for various purposes including monitoring, traffic pattern analysis, and capacity planning. Each request in the access log has a set of attributes including URI representing the accessed page, response document size, and response time. Where the response document size represents the number of bytes of the response and the response time represents the time consumed to serve the request.

We use each request URI's document size and response time attributes from the historical access logs to identify partitions with similar characteristics. Each partition represents URIs with similar response time and document size which also demands similar hardware resources including CPU, disk, and network etc. Therefore, identifying similar URIs and mapping them to microservices can improve the application performance and scalability.

The URIs for a specific web application can be clustered into a set of k discrete partitions

$\{c_1, c_2, \ldots, c_k\}$ such that the resource consumption characteristics remain uniform within each partition.To appropriately identify URI partitions, we require historical usage/access logs of the target application which contains a fair distribution of the requests to each URI of the application. As a first step, we preprocess the historical usage logs to obtain response time, document size, and URI attributes of each request.

To identify the URIs groups, we use Scale Weighted Kmeans (SWK) method. Let $\{\mathbf{u}_i = (\texttt{document\_size,response\_time})\}_{i=1}^{n}$ are n observations corresponding to the n access log requests identified after preprocessing. $\mathbf{c}_j = \{w_j, \boldsymbol{\mu}_j, \boldsymbol{\sigma}_j\}$ where $j \in \{1, \cdots k\}$ are the k distributions, where w_j represents relative weight, $\boldsymbol{\mu}_j$ represents average, and $\boldsymbol{\sigma}_j$ represents standard deviation for each of the distribution. We need to minimize the following objective function:

$$\arg\min_{\boldsymbol{\alpha},\boldsymbol{\Gamma}} \left(\sum_{h=1}^{k} \sum_{i=1}^{n} f(\mathbf{u}_i, \boldsymbol{\alpha}_h)(\mathbf{u}_i - \boldsymbol{\alpha}_h)\boldsymbol{\Gamma}(\mathbf{u}_i - \boldsymbol{\alpha}_h)^\top \right), \tag{1}$$

where $\boldsymbol{\alpha}_h$ is a randomly selected center for each partition and $f(\mathbf{u}_i, \boldsymbol{\alpha}_h)$ is a binary function yielding 1 if the Euclidean distance between $\mathbf{u}_i$ and the current $\boldsymbol{\alpha}_h$ is less than all other $\boldsymbol{\alpha}$ otherwise it will yield 0. We introduce matrix $\boldsymbol{\Gamma}$ which is a diagonal matrix used to define a relative importance of each dimension of $\mathbf{u}_i$. We consider that the response time is more important than the document size as it consumes more hardware resources compared to the document size. The objective function given in equation (1) is solved iteratively in a similar fashion to the standard K-means algorithm. Therefore, we named our proposed solution as Scale Weighted Kmeans (SWK) algorithm. We have also applied the SWK in one of our recent work [28] to predict dynamic workload patterns of web applications.

Initially, the URIs are divided into a user-defined similar k partitions by randomly selecting the center $\boldsymbol{\alpha}_h$ of each partition. Then, we iteratively move each u_i to the nearest $\boldsymbol{\alpha}_h$ and also update the centers based on the most recent assignments of all the u_i. The iteration is terminated once we observe that the objective function minimization across two consecutive iterations is significantly similar.

It is important to identify a good value for k to identify appropriate URI partitions. We address this concern by using the elbow method. The elbow method iterates the value of k from 1 to increasing order and computes the residual error for equation (1) for each value of the k. Initially, the error decreases at a significant rate, however, after some values of k the error does not decrease significantly. We use the last value of the k before observing a significant reduction in the residual error.

We observed that a URI can lie among multiple partitions as multiple requests for the same URI can report significant variations in document size and response time due mainly to the query parameters. A URI may return only one record based on a given query parameter, however, the same URI can return a large number of records for a different query parameter. To decompose a web application appropriately, we need to assign each distinct URI to only one partition. Therefore, after we obtain k partitions from the proposed SWK method, we map each URI path to the corresponding partition by performing a majority voting for all URIs. The majority voting for each URI gives us the group where the URI occurs more frequently. This ensures mapping each distinct URI to only one partition.

3.2. Microservices Deployment

Given a k discrete URI partitions of the web application, the system transforms each URI partition into separate microservice and deploys them automatically. Assuming $S = \{c_1, c_2, \cdots, c_k\}$ is a set of discrete URI groups identified by the proposed URI space partitioning method. Then, we sort the S in decreasing order based on the centroid of each URI group. This will ensure that the group consuming more resources are at the beginning of the list. A set of microservices $M = \{m_1, m_2, \cdots, m_k\}$ mapped to the S by a function $g : S \rightarrow M$, where g performs a one to one mapping by assigning all URIs of the specific group s_i to the microservice m_i.

Our aim is to deploy the microservices on cloud resources automatically and seamlessly to offer the same functionality of the monolithic implementation. Therefore, to make this process easy, we deploy copies of the monolithic implementation on k different VMs and then dynamically control the URIs requests from the application's load balancer by redirecting the requests to the corresponding microservices responsible for serving the specific URIs.

In our implementation, we used Nginx as an application endpoint and HTTP load balancer. The application client requests are first received by Nginx which forwards the request to the corresponding microservice responsible of serving the requested URI. Nginx allows managing URI-based request forwarding to a cluster of servers hosting the URIs easily and efficiently. This makes it easier to automate the deployment and scaling of the microservices.

An appropriate VM type selection is important to deliver the desired performance while reducing the cost. There have been several optimization-based solutions to identify the VM type for applications to deploy on the cloud to achieve better performance [6, 29, 30]. However, the complexity of the existing state-of-art VM type selection, the performance varying cloud resources, and the application workload behavior makes it difficult to identify an appropriate VM type for each microservice. We propose a simple and an efficient method for selecting an appropriate VM type to deploy the microservices.

3.2.1. Dynamic VM Type Selection

Given a set $M = \{m_1, m_2, \cdots, m_k\}$ which represents k different microservices sorted based on resource demands as explained in the previous section. Given another set of user selected VM types $V_T = \{(v_{t_1}, v_{t_2}, \cdots, v_{t_z}) \mid v_{t_1} > v_{t_2} \cdots > v_{t_z}\}$ sorted based on VM capacity. The system assigns the VM types V_T to the microservices M by iterating both sets simultaneously to assign VM type v_{t_i} to microservice m_i using the following condition:

$$\varphi(m_i) = \begin{cases} v_{t_i} & \text{if} \quad i \leq z \quad \text{for} \quad \forall i \\ v_{t_z} & \text{otherwise,} \end{cases} \tag{2}$$

where z is a total number of different VM types that can be assigned to the microservices. The system assigns V_T to M using one to one mapping provided that the number of VM types z is greater or equal to the number of microservices k because we have a sorted M and V_T with respect to resource consumption requirements and machine capacity respectively. However, if all of the VM types are assigned and system identifies few remaining microservices then the weakest VM type v_{t_z} will be assigned to all of the remaining microservices.

The intuition behind the proposed VM type assignment method is to assign a powerful VM type to the microservices requiring more resources and relatively less powerful machine to the microservices demanding fewer resources. This becomes possible using the proposed URI space partitioning, explained in Section 3.1, which helps to decompose the web application into similar resource demanding microservices and then we sort the microservices with respect to their resource demands.

Algorithm 1: Microservices auto-scaling method.

Input: Time interval in seconds (ξ),CPU utilization threshold (τ_{cpu}), a set of initially allocated VMs to microservices ($V_{m_1}, V_{m_2}, \cdots, V_{m_k}$)

Output: updated $V_{m_1}, V_{m_2}, \cdots, V_{m_k}$

```
1  t ← 1;
2  while true do
3  |  Wait for ξ seconds;
4  |  for i= 1 to k do
5  |  |  ρ_i ← Extract average CPU utilization for all VMs allocated to m_i^th
   |  |   microservice;
6  |  |  if ρ_i > τ_cpu then
7  |  |  |  v_{t_i} ← φ(m_i); get VM type for m_i^th microservice using Equation 2
8  |  |  |  v'_i ← instantiate a new VM for m_i^th microservice of v_{t_i} type  ;
9  |  |  |  V_{m_i} ← V_{m_i} ∪ {v'_i};
10 |  |  |  update and reload Nginx (loadbalancer) configurations;
11 |  |  end
12 |  end
13 |  t ← t + 1;
14 end
```

3.3. Microservices Auto-Scaling

Our main contribution is not to devise a new auto-scaling methods. Any auto-scaling method can be used to automatically provision the resources to the proposed decomposed services. To show the proposed system working and effectiveness, we used an implementation similar to [31] by extending it to work with multiple microservices of a web application. In this section, we explain the method used to auto-scale the decomposed microservices dynamically.

Algorithm 1 shows the pseudocode for the proposed auto-scaling method used in our experimental evaluation to show the effect of automatic application decomposition on the performance and scalablity. The algorithm accepts a time interval in seconds (ξ) to monitor the application and gather necessary statistics, a user-defined CPU utilization threshold τ_{cpu} for initiating a scaling decision, and a set of initially provisioned VMs for k different micorservices ($V_{m_1}, V_{m_2}, \cdots, V_{m_k}$). In each iteration, the algorithm waits for (ξ) seconds and then calculate the average CPU utilization ρ_i for all the machines allocated to a specific i^{th}

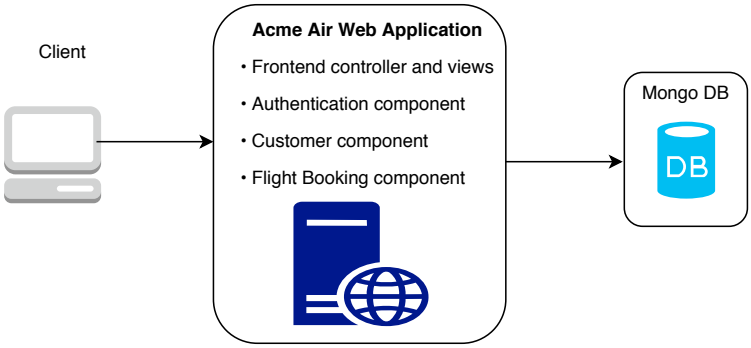


Figure 3: Deployment architecture of the monolithic implementation of the Acme Air benchmark web application.

microservice. This step repeats k times to identify the saturating microservices. If average CPU utilization ρ_i for VMs allocated to any microservice is greater than the user defined CPU utilization threshold τ_{cpu} then the algorithm performs a horizontal scaling for this microservice by dynamically instantiating and adding another VM to of specific type to that microservice and update the corresponding V_{m_i}. This step dynamically updates the Nginx configuration to reflect the newly added machine to a specific microservice and it is used to dynamically load-balance the incoming user requests.

A smaller value for (ξ) can be used to react quickly to add resources, however, a larger value can slow down the reaction time and can be used to ignore a short term brustiness in the workload. A higher observation of web server CPU utilization shows a possibility of server saturation which will affect the application performance drastically. In our experimental evaluation, we used $\xi = 60$ seconds and CPU utilization threshold $\tau_{cpu} = 80\%$. We allocate only one VM to each of the microservice initially and then the proposed auto-scaler dynamically scale the resources to the microservices.

4. Experimental Setup and Design

This section presents the experimental benchmark application, the testbed infrastructure, the workload generation method, and the details of the experiments conducted to evaluate the proposed application decomposition and the auto-scaling method.

4.1. Benchmark Application

We used Acme Air [4] open source benchmark web application developed with both monolithic and microservices architectures. Figure 3 illustrates the deployment of Acme Air monolithic implementation on Apache server and the database on MongoDB. Acme Air microservices implementation has five different microservices as shown in Figure 4. This decomposition is done manually by software engineers. Multiple implementations of Acme Air application are available using different programming languages. We used the Java-based microservices implementation in the experimental evaluation.

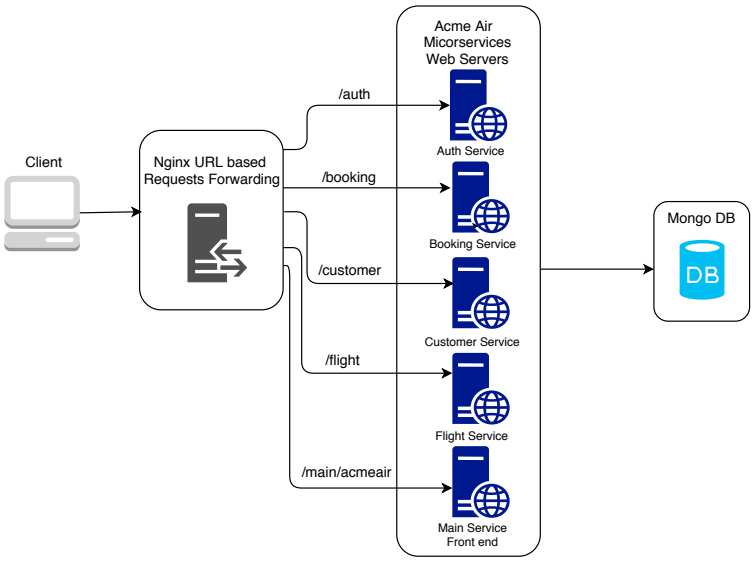


Figure 4: Deployment architecture of the microservices implementation of the Acme Air benchmark web application.

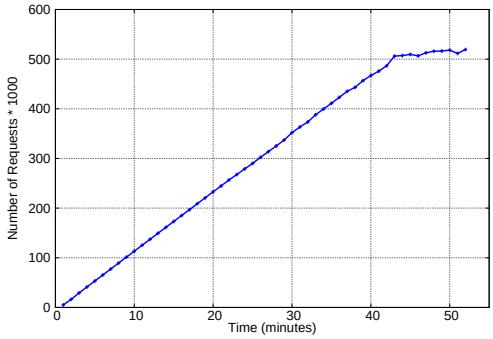


Figure 5: Linearly increasing workload used in the experimental evaluation.

4.2. Workload Generation

We used Apache JMeter [32] to generate a synthetic workload for the Acme Air benchmark web application. We used a publicly available workload scenario [33] for the benchmark application. The default scenario includes user login, query flights for different routes, book a trip, list passed flight bookings, and cancel a flight. We generate a linearly increasing workload by emulating a specific number of concurrent user sessions. The workload generator starts with 200 requests per second and increases it after every minute in a step-up fashion with a fixed rate of 200 requests per second up to a maximum of $8,600$ requests per second. We generated this workload for 52 minutes. Figure 5 shows the number of requests generated per minute for all of the experiments. JMeter reaches $8,600$ requests per second in 43 minutes, generating $516 \times 1,000$ requests per minute, and then for the remaining 9 minutes, JMeter does not increase the requests. The linearly increasing workload is commonly used to evaluate the application performance with an increasing number of user requests.

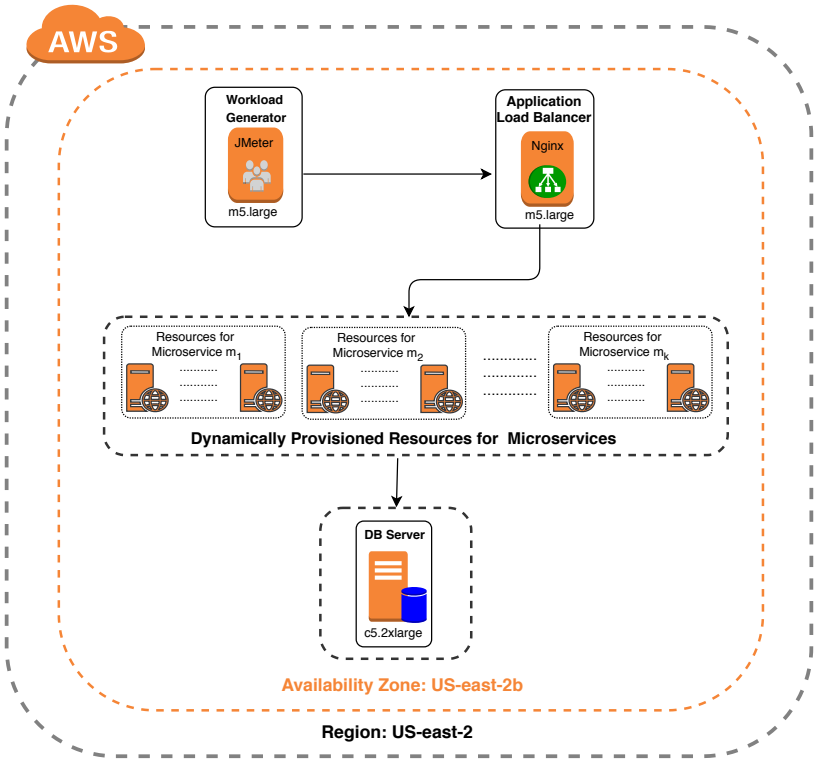


Figure 6: Testbed cloud infrastructure used for the experimental evaluation.

We specifically designed the workload to finish within an hour to minimize the experimental cost for using cloud resources.

4.3. Testbed Cloud Infrastructure

We performed all experiments on the Amazon Web Services (AWS) public cloud. Figure 6 shows the testbed cloud infrastructure used in our experimental evaluation. AWS offer cloud services in many geographical regions, we performed all experiments on AWS datacenter in `us-east-2` region. We used a dedicated EC2 `m5.large` instance to generate the application workload. Another EC2 `m5.large` instance is used as an application endpoint and load balancer. The proposed system dynamically adjusts the pool of EC2 instances allocated to each of the microservices. Initially each pool contains one VM of type `c5.large`. Our proposed auto-scaling method controls the number of VMs in these dynamically provisioned VM pools. A dedicated EC2 instance of type `c5.2xlarge` is used to host the MongoDB database server for the Acme Air benchmark application.

The VMs allocated to the workload generator, the load balancer, and the database server are over-provisioned to avoid any saturation on these tiers during our experimental evaluation and to be able to focus the study on the scalability and the performance evaluation of the application tier. We performed 9 iterations with different combinations of VM types to identify the appropriate VM types for the load balancer, the workload generation, and the database to avoid saturation at these tiers for the linearly increasing workload.

Table 1: Summary of conducted experiments

Experiment	Description
Experiment 1: Monolithic Implementation (baseline)	Dynamic resource provisioning for the monolithic implementation of the benchmark application using homogeneous VMs of type `c5.large`.
Experiment 2: Manually Created Microservices	Dynamically resource provisioning for the manually created microservices using the proposed auto-scaling method with homogeneous VMs of type `c5.large`.
Experiment 3: Auto-created Microservices	Dynamic resource provisioning using the microservices created by the proposed method with homogeneous VMs of type `c5.large`.
Experiment 4: Auto-created Microservices with Dynamic VM Type Selection	Dynamic resource provisioning using the microservices created by the proposed method with dynamic VM types to the microservices.

4.4. Experimental Details

We performed and reported four experiments to study the scalability and performance of the benchmark application under various loads using four different deployment strategies. Table 1 summarizes the conducted experiments. Experiment 1 is used as a baseline, experiment 2 is used to evaluate manually created microservices whereas experiments 3 and 4 are conducted using the proposed system.

In **Experiment 1**, we profile the performance of a baseline monolithic implementation of the benchmark application. Dynamic resource provisioning method is used to scale-out the web server tier whenever the average CPU utilization of the allocated VMs saturate. The system uses homogeneous VMs of type `c5.large` to deploy and scale the application.

In **Experiment 2**, we profile the performance of manually created microservices of the benchmark application. We deployed these microservices on the testbed cloud infrastructure and dynamically scale-out the resources allocated to the microservices using the proposed auto-scaling method. In this experiment, homogeneous VMs of type `c5.large` are used to deploy and scale the microservices.

In **Experiment 3**, we deploy the microservices created using the proposed decomposition method on the testbed cloud infrastructure and study the application performance under the proposed auto-scaling method. In this experiment, we employ homogeneous VMs of type `c5.large` to deploy and scale the microservices.

In **Experiment 4**, we deploy the microservices created using the proposed decomposition method on the testbed cloud infrastructure and study the application performance using the proposed auto-scaling and VM type selection methods. In this experiment, the system automatically identifies the VM type using the proposed dynamic VM type selection method explained in Section 3.2.1, to deploy and scale the microservices to meet the desired performance requirements.

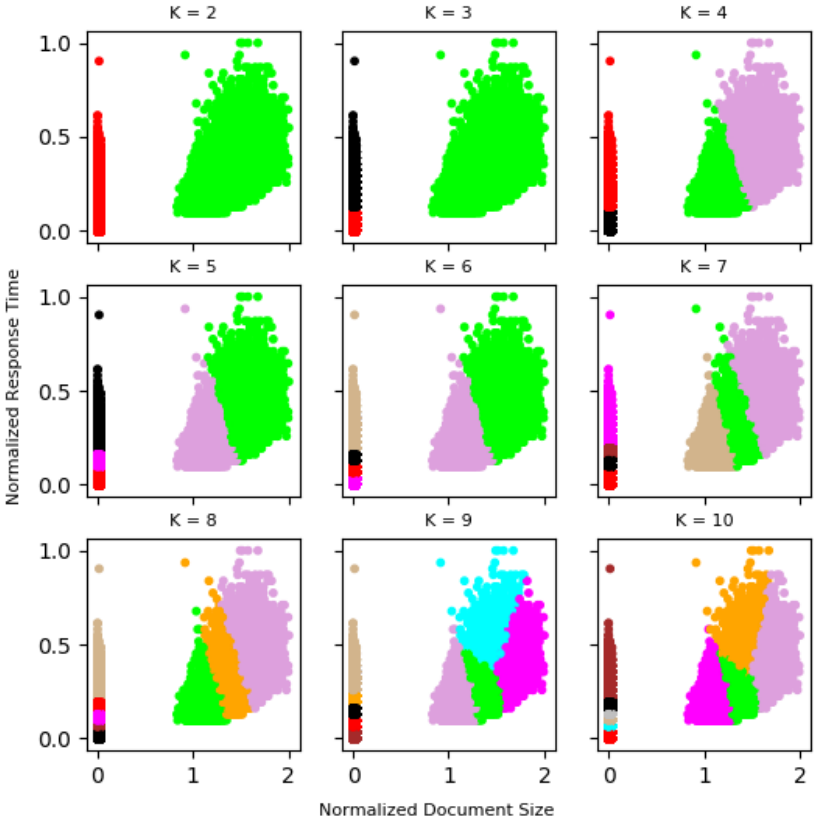


Figure 7: URI space partitioning for different values of k.

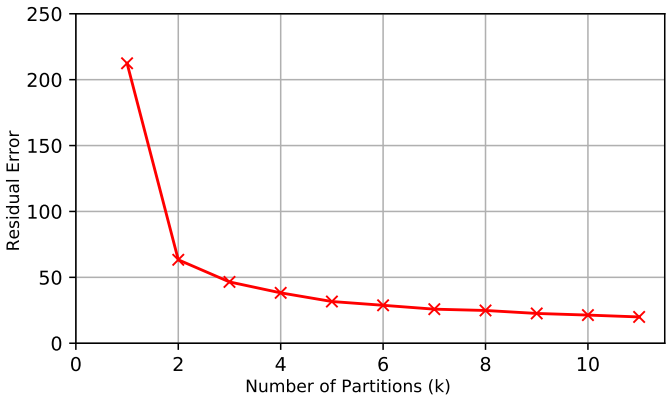


Figure 8: Variation of residual error with increasing number of partitions using the proposed SWK method to find the optimal value of k.

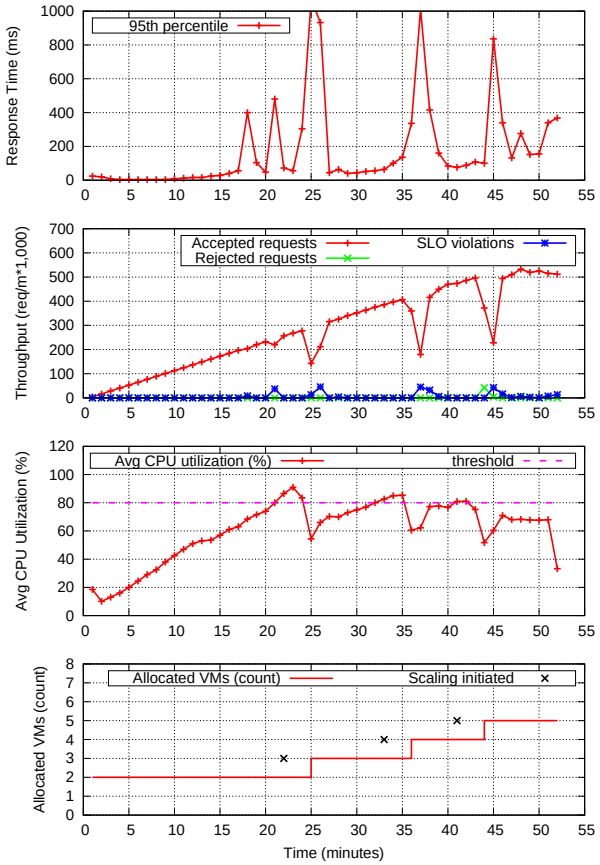


Figure 9: Experiment 1 results showing 95^{th} percentile of response time, throughput, average CPU utilization, and the number the dynamically allocated VMs for the monolithic implementation.

5. Experimental Result

5.1. URI Space Partitioning Evaluation

URI space partitioning method identifies URIs partitions using the SWK method explained in Section 3.1. The vector $\mathbf{u}_i$ in Equation (1) is initialized as $\mathbf{u}_i$=[`Document_Size Response_Time`]$^\top$ where each dimension is scale normalized between 0 and 1. The optimal weight matrix $\mathbf{\Gamma}$ in Equation (1) is a 2×2 diagonal matrix and found to be [2 0; 0 1] for the Acme Air benchmark web application. Figure 7 shows the URI space partitions using the proposed SWK method for varying value of k from k =2 to k =10. For all values of k, the boundaries of the groups are distinguishable without any overlapping. However, the small values of k separates the group better compared to larger values of k.

Figure 8 shows the variation of the residual error with increasing number of partitions to identify the optimal value of k. We used the sum of square difference as the residual error to evaluate the URI space partitioning method. We can clearly observe that the error decrease is not significant when k is greater than elbow point 2 Therefore, we consider 2 as the optimal value of k for the Acme Air benchmark web application.

5.2. Experiment 1: Monolithic Implementation

Figure 9 shows 95^{th} percentile response time, average CPU utilization, number of VMs allocated to the application, and throughput during experiment 1. We show the throughput for accepted requests, rejected requests, and response time SLO violations. The accepted requests represent the total number of processed requests, the rejected requests show the number of requests the system could not process, and the SLO violations represent the number of requests having a response time higher than the desired one. We consider $1,000$ ms as a maximum acceptable response time during all of the experiments. The dynamic addition of VMs to the application for maintaining the desired performance is shown in the last graph of Figure 9. We indicate the time when the system initiates a scaling operation and then show an increased VMs count when the VM starts serving requests.

The proposed auto-scaling method detects the system saturation appropriately. However, during the time required to launch the VM the system performance degrades. For example, whenever the average CPU utilization for the allocated VMs saturates, the throughput decreases and the response time increases which yield SLO violations and the system starts rejecting some requests. Once a new VM joins the cluster and starts serving requests, the desired performance is quickly restored. For example, the first scaling decision is observed at 22^{nd} time interval and the VM starts serving requests at 25^{th} time interval. We observe a system performance degradation between these time intervals. However, after 25^{th} time interval the performance of the system is restored until the next saturation.

5.3. Experiment 2: Manually Created Microservices

Figure 10 depicts the application performance under dynamic resource provisioning of manually created microservices during experiment 2. We show 95^{th} percentile response time, throughput, average CPU utilization, and dynamic addition of VMs to the application during this experiment.

The manually-created microservices implementation of the benchmark web application consists of five microservices. Initially we deployed each microservice in its own VM. The auto-scaling method dynamically provisions more VMs to saturating microservices due to increased workload. The first scale operation occurs at the 11^{th} time interval as the system initiated a new VM to add to the saturating microservice. The VM starts serving requests at 14^{th} time interval and the system performance is restored immediately. Another saturation occurred at 20^{th} time interval which caused a system performance degradation. However, the proposed auto-scaling method restored the desired performance by provisioning an additional VM. Finally, a saturation is observed at 31^{st} time interval and the system reacted to restore the application performance gracefully.

This experiment shows that microservice 1 (Booking Service) saturated several times during the experiment whereas all other microservices were underutilized and did not experience any saturation. This highlights the need to consider the performance impact of microservices decomposition manually created based solely on software engineering principles. The Booking Service implements operations that are heavily accessed and required higher resource demands to serve the workload compared to other microservices. The results

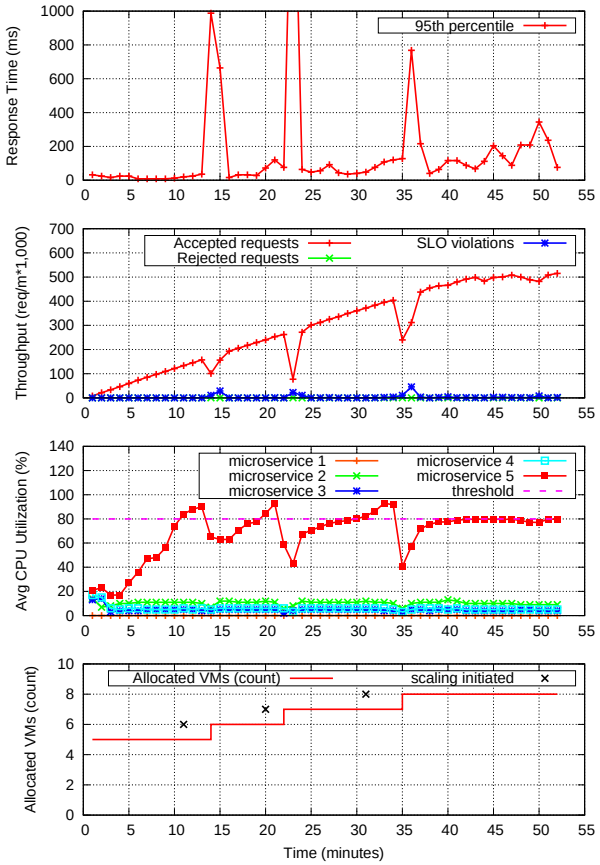


Figure 10: Experiment 2 results showing 95^{th} percentile of response time, throughput, average CPU utilization, and dynamic allocation of VMs to the manually created microservices.

in this experiment validate a need to decompose an application into microservices based on resource demands and then assign appropriate VM types for better performance.

5.4. Experiment 3: Auto-created Microservices

Figure 11 shows the application performance under dynamic resource provisioning of auto-created microservices during experiment 3. The figure depicts the 95^{th} percentile response time, throughput, average CPU utilization, and provisioning of VMs to the microservices during this experiment.

Our proposed method identified two microservices for the Acme Air benchmark web application. Each microservice is deployed to its own VM. Then, the proposed auto-scaling method dynamically allocates additional VMs to the microservices in case of CPU saturation to maintain the desired performance. The first scaling operation occurred at the 15^{th} time interval and the system initiated a new VM to allocate to the saturating microservice. The VM starts serving requests at 19^{th} time interval and the system performance is restored immediately. The second scaling operation occurred at the 20^{th} time interval and the desired performance is restored by provisioning an additional VM. Finally, a saturation at time

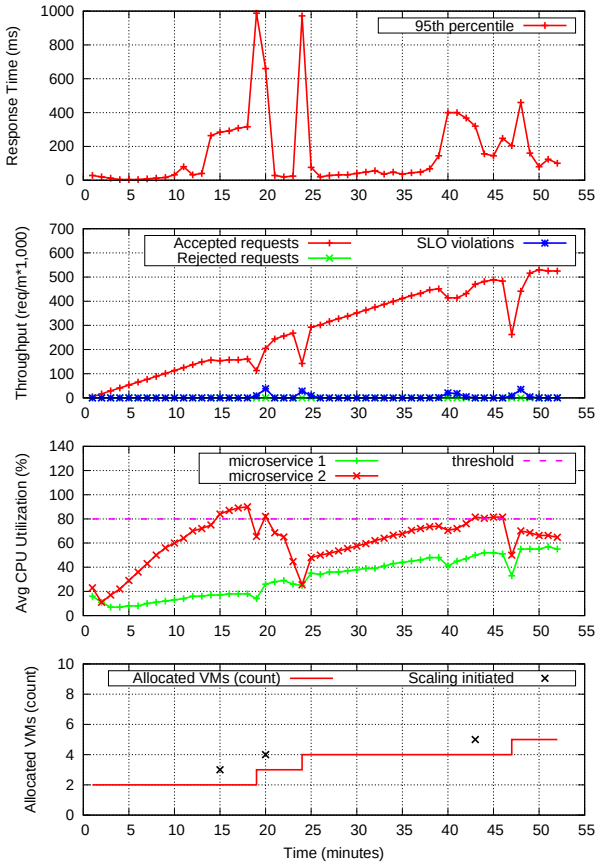


Figure 11: Experiment 3 results showing 95^{th} percentile of response time, throughput, average CPU utilization, and dynamic allocation of VMs to the auto-created microservices.

interval 43^{th} is observed and an additional VM was provisioned to restore the application performance gracefully.

This experiment shows that the proposed method decomposed the application into fewer microservices that require less number of VMs to deploy and also utilize the allocated resources more efficiently. This experiment only used 5 VMs to serve the workload whereas the manually-created microservices (Experiment 2) used 8 VMs. The reduced number of microservices and allocated VMs greatly reduces the interaction between the microservices and brings down the operational cost.

5.5. Experiment 4: Auto-created Microservices with Dynamic VM Type Selection

Figure 12 shows the results obtained during experiment 4. To show the performance of the application under dynamic resource provisioning of microservices and dynamic VM type selection, we draw 95^{th} percentile response time, throughput, average CPU utilization and dynamic addition of VMs to the application during this experiment.

Our proposed solution decomposed the benchmark monolithic web application into two microservices. Each microservice is deployed to its own VM. The system automatically

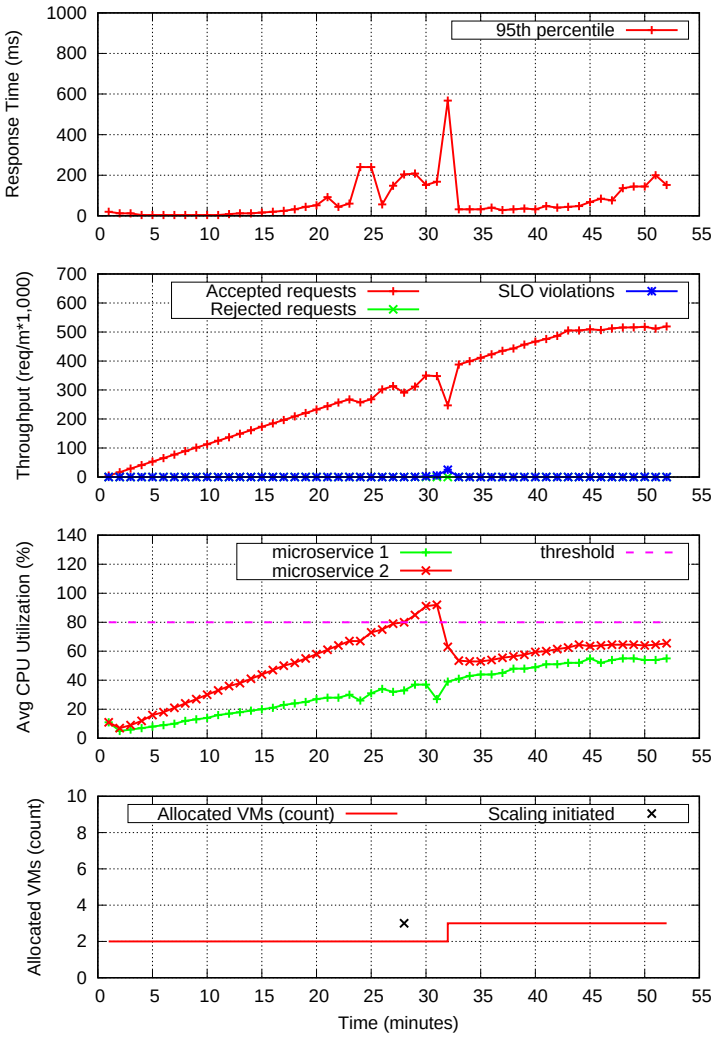


Figure 12: Experiment 4 results showing 95^{th} percentile of response time, throughput, average CPU utilization, and dynamic allocation of VMs to the auto-created microservices with dynamic VM type selection.

allocated a VM of type `c5.large` to microservice 1 and `c5.xlarge` to microservice 2 using the proposed VM type selection method explained in Section 3.2.1. The auto-scaling method provisioned more VMs of the specific type to the microservice experiencing CPU saturation. We only observed one scaling operation at 28^{th} time interval and the system dynamically added a new VM of type `c5.xlarge` to microservice 2. This experiment showed only one saturation and used three VMs in total to serve the workload. This yields minimal SLO violations and rejections compared to the manually-created microservices (Experiment 2).

The experimental result using the proposed auto-decomposition and VM type selection methods yield excellent results because the URIs demanding similar resources are packaged into microservices, and then appropriate VM type is assigned to each microservice based on the intensity of resource demands. This yields better performance compared to manually-

Table 2: Experimental results summary. Total processed requests, total rejected requests, total response time SLO violations, percentage of requests with response time SLO violations, total allocated VMs, and total cost for experiments 1, 2, 3, and 4.

Experiment	**Processed** (Total)	**Rejected** (Total)	**SLO Violations** (Total)	**SLO Violations** (%)	**VMs Used** (Total)	**Cost ($)**
Exp 1 (Baseline)	14,542,572	48,040	333,430	2.29	5	0.425
Exp 2	15,164,462	2,004	171,314	1.13	8	0.680
Exp 3	14,612,509	1,487	178,889	1.22	5	0.425
Exp 4	15,342,106	1,231	36,258	0.24	3	0.425

created microservices.

5.6. Experimental Summary

To compare the performance results obtained in all experiments, we calculate the total processed requests, total rejected requests, total response time SLO violations, total VM instances used, and the total cost for the VMs used. The comparison is shown in Table 2. Ideally, the best method should yield maximum completions, minimum rejections, minimum response time SLO violations, and minimum cost. Experiment 4 outperforms all other experiments and turns out to be the best as it yields the maximum completions and the minimum SLO violations and cost.

We consider Experiment 1 as a baseline method and compare the results of all other experiments with it. Figure 13 shows the relative percentage of processed requests, rejected requests, response time SLO violations, total VMs used, and the total cost for Experiments 2, 3, and 4 compared to Experiment 1 (baseline). We observed 4.3%, 0.5%, and 5.5% higher processed requests relative to the baseline for Experiment 2, 3, and 4 respectively. We also observed 95.8%, 96.9%, and 97.4% percent less rejected requests compared to the baseline for Experiments 2, 3, and 4 respectively. The response time SLO violations are reduced by 48.6%, 46.3%, and 89.12% for Experiments 2, 3 and 4 respectively compared to the baseline. The cost is increased by 60% for Experiment 2 compared to the baseline. However, the cost of Experiment 3 and 4 is similar to the baseline. Overall, Experiment 4 using the proposed auto-decomposed microservices with dynamic VM type selection method significantly outperformed the baseline.

5.7. Discussion

A typical monolithic application is tightly coupled and scaling it out requires deploying the entire application to maintain the desired performance even if only a small subset of its functionality is under heavy load. Requests for specific resource demanding services can cause saturation of the whole application even on a small number of concurrent requests. Thus, the autoscaling as shown in Experiment 1 had to scale frequently to maintain the monolithic application desired performance. This results in more SLO violations and lower overall application performance. Whereas, the manually-created microservices designed based on software engineering principles do not consider the impact on the overall

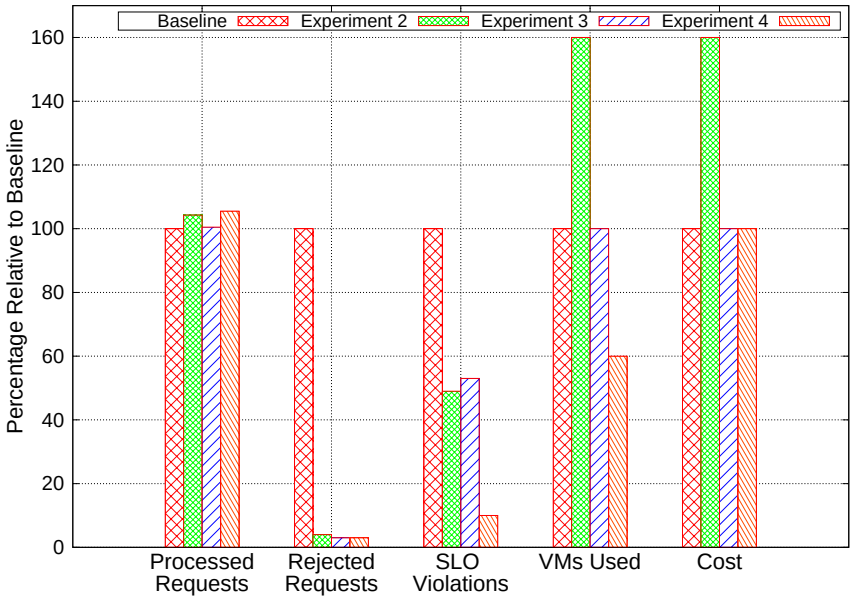


Figure 13: Comparison of the proposed methods with the baseline.

application performance. The Acme air manually created microservices run independently and require more interactions with each other over HTTP which incurs higher latency and communication overhead.

The proposed auto-decomposition method packages as a microservice the URIs requiring similar hardware resources to serve the user requests and then assigns appropriate VM type to each microservice based on the intensity of resource demands. The proposed solution for the Acme air monolithic application identified only two microservices thus reduced the latency caused by inter-service communication between microservices. Moreover, This helps to increase the performance of the microservices over monolithic and manually-created microservices. Furthermore, identifying appropriate VM type during the automatic scaling helps to significantly reduce the SLO violations and rejections.

The proposed approach to decompose an application into microservices considers the historical application access logs and identifies URIs partitions based on the response time and the document size. Such partitions demand similar system resources and they are mapped to microservices to ease scaling their deployment while minimizing the inter-service communication overhead. Then it dynamically assigns the appropriate VM type to the identified microservices based on the resource usage intensity. Therefore, the proposed method is application independent and can be used to decompose a given monolithic application into microservices to yield better performance.

6. Conclusion and Future Work

Building an automated solution to improve the performance of a legacy monolithic web application is a challenging task. Existing software engineering based decomposition ap-

proaches do not take performance considerations into account in the development life cycle. Thus yielding microservices that are likely to perform worse than the monolithic version due to the increased inter-services communication. This paper has first demonstrated the need to consider performance aspects through an empirical study to analyze the performance difference between the two architectures using the monolithic and the microservices implementations of Acme Air open source benchmark application. Then the paper proposed a novel method to automatically decompose a monolithic application into microservices while improving the application performance and scalability. Our proposed decomposition method is based on a black-box approach that mines the application access logs using a clustering method to discover URL partitions having similar performance and resource requirements. Such partitions are mapped to microservices. Then it decides the appropriate VM type to host the identified microservices based on the resource usage intensity. The experimental results using a testbed on a public cloud infrastructure show that the proposed solution yields a significant improvement of the application performance and scalability compared to the manually created microservices while reducing the overall operational cost.

Currently, we are extending the solution to incorporate advanced predictive auto-scaling methods to further reduce SLO violations. We are also exploring the possibility of further reducing the operational cost of the proposed system by deploying the auto-created microservices to containers rather than VMs. Extending the proposed approach to enable the automatic decomposition of the database layer into smaller instances is another interesting future work direction. Furthermore, the proposed approach could be extended to optimize the granularity of microservices by considering and reducing the inter-services communication between microservices and thereby preventing performance degradation.

Acknowledgement

This work was made possible by NPRP grant # 7-481-1-088 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

References

[1] A. A. Almonaies, M. H. Alalfi, J. R. Cordy, T. R. Dean, Towards a framework for migrating web applications to web services, in: Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research, CASCON '11, IBM Corp., Riverton, NJ, USA, 2011, pp. 229–241.

[2] A. Levcovitz, R. Terra, M. T. Valente, Towards a technique for extracting microservices from monolithic enterprise systems, CoRR abs/1605.03175. arXiv:1605.03175.

[3] M. Gysel, L. Kölbener, W. Giersche, O. Zimmermann, Service cutter: A systematic approach to service decomposition, in: M. Aiello, E. B. Johnsen, S. Dustdar, I. Georgievski (Eds.), Service-Oriented and Cloud Computing, Springer International Publishing, Cham, 2016, pp. 185–200.

[4] Acme air benchmark web application, https://github.com/blueperf (2018).

[5] T. Ueda, T. Nakaike, M. Ohara, Workload characterization for microservices, in: IEEE International Symposium on Workload Characterization (IISWC), 2016, pp. 1–10.

[6] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, M. Zhang, Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics., in: NSDI, Vol. 2, 2017, pp. 4–2.

[7] C.-J. Hsu, V. Nair, T. Menzies, V. W. Freeh, Scout: An experienced guide to find the best cloud configuration, arXiv preprint arXiv:1803.01296.
[8] W. Iqbal, M. N. Dailey, D. Carrera, Low cost quality aware multi-tier application hosting on the amazon cloud, in: International Conference on Future Internet of Things and Cloud (FiCloud), IEEE, 2014, pp. 202–209.
[9] A. Christoforou, M. Garriga, A. S. Andreou, L. Baresi, Supporting the decision of migrating to microservices through multi-layer fuzzy cognitive maps, in: M. Maximilien, A. Vallecillo, J. Wang, M. Oriol (Eds.), Service-Oriented Computing, Springer International Publishing, Cham, 2017, pp. 471–480.
[10] A. Furda, C. Fidge, O. Zimmermann, W. Kelly, A. Barros, Migrating enterprise legacy source code to microservices: On multitenancy, statefulness, and data consistency, IEEE Software 35 (3) (2018) 63–72.
[11] A. Balalaie, A. Heydarnoori, P. Jamshidi, Migrating to cloud-native architectures using microservices: an experience report, in: European Conference on Service-Oriented and Cloud Computing, Springer, 2015, pp. 201–215.
[12] D. Taibi, V. Lenarduzzi, C. Pahl, Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation, IEEE Cloud Computing 4 (5) (2018) 22–32.
[13] M. Villamizar, O. Garcs, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, M. Lang, Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures, in: 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2016, pp. 179–182.
[14] B. Costa, P. F. Pires, F. C. Delicato, P. Merson, Evaluating rest architectures approach, tooling and guidelines, Journal of Systems and Software 112 (2016) 156 – 180.
[15] W. Hasselbring, G. Steinacker, Microservice architectures for scalability, agility and reliability in e-commerce, in: IEEE International Conference on Software Architecture Workshops (ICSAW), 2017, pp. 243–246.
[16] D. Trihinas, A. Tryfonos, M. D. Dikaiakos, G. Pallis, Devops as a service: Pushing the boundaries of microservice adoption, IEEE Internet Computing 22 (3) (2018) 65–71.
[17] W. Hasselbring, G. Steinacker, Microservice architectures for scalability, agility and reliability in e-commerce, in: IEEE International Conference on Software Architecture Workshops (ICSAW), IEEE, 2017, pp. 243–246.
[18] H. Alipour, Y. Liu, Online machine learning for cloud resource provisioning of microservice backend systems, in: Big Data (Big Data), 2017 IEEE International Conference on, IEEE, 2017, pp. 2433–2441.
[19] M. Gotin, F. Lösch, R. Heinrich, R. Reussner, Investigating performance metrics for scaling microservices in cloudiot-environments, in: Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering, ACM, 2018, pp. 157–167.
[20] M. R. López, J. Spillner, Towards quantifiable boundaries for elastic horizontal scaling of microservices, in: Companion Proceedings of the10th International Conference on Utility and Cloud Computing, ACM, 2017, pp. 35–40.
[21] T. Gerald, J. Nicholas K., D. Rajarshi, B. Mohamed N., A hybrid reinforcement learning approach to autonomic resource allocation, in: IEEE International Conference on Autonomic Computing, 2006, pp. 65–73.
[22] L. Yazdanov, C. Fetzer, Lightweight automatic resource scaling for multi-tier web applications, in: 7th IEEE International Conference on Cloud Computing (Cloud'2014), IEEE, 2014, pp. 466–473.
[23] X. Bu, J. Rao, C.-Z. Xu, A reinforcement learning approach to online web systems auto-configuration, in: Proceedings of 29th IEEE International Conference on Distributed Computing Systems, ICDCS '09, IEEE Computer Society, 2009, pp. 2–11.
[24] H. Fernandez, G. Pierre, T. Kielmann, Autoscaling web applications in heterogeneous cloud infrastructures, in: IEEE International Conference on Cloud Engineering, 2014, pp. 195–204.
[25] F. J. A. Morais, F. V. Brasileiro, R. V. Lopes, R. A. Santos, W. Satterfield, L. Rosa, Autoflex: Service agnostic auto-scaling framework for iaas deployment models, in: 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, 2013, pp. 42–49.

[26] M. Sedaghat, F. Hernandez-Rodriguez, E. Elmroth, A virtual machine re-packing approach to the horizontal vs. vertical elasticity trade-off for cloud autoscaling, in: Proceedings of the ACM Cloud and Autonomic Computing Conference, CAC '13, ACM, New York, NY, USA, 2013, pp. 6:1–6:10.
[27] M. Mao, J. Li, M. Humphrey, Cloud auto-scaling with deadline and budget constraints, in: 11th IEEE/ACM International Conference on Grid Computing, 2010, pp. 41–48.
[28] W. Iqbal, A. Erradi, A. Mahmood, Dynamic workload patterns prediction for proactive auto-scaling of web applications, Journal of Network and Computer Applications 124 (2018) 94–107.
[29] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, R. H. Katz, Selecting the best vm across multiple public clouds: A data-driven performance modeling approach, in: Proceedings of the Symposium on Cloud Computing, ACM, 2017, pp. 452–465.
[30] C.-J. Hsu, V. Nair, T. Menzies, V. Freeh, Micky: A cheaper alternative for selecting cloud instances, arXiv preprint arXiv:1803.05587.
[31] W. Iqbal, M. N. Dailey, D. Carrera, P. Janecek, Adaptive resource provisioning for read intensive multi-tier applications in the cloud, Future Generation Computer Systems 27 (6) (2011) 871–879.
[32] Apache jmeter, `https://jmeter.apache.org/`.
[33] Acme air workload driver, `https://github.com/blueperf/acmeair-driver`.