Partial Migration for Re-architecting a Cloud Native Monolithic Application into Microservices and FaaS

Chapter · November 2020

DOI: 10.1007/978-981-15-9671-1_9

4 authors, including:

Deepali Bajaj
University of Delhi
21 PUBLICATIONS **171** CITATIONS
SEE PROFILE

Urmil Bharti
University of Delhi
17 PUBLICATIONS **143** CITATIONS
SEE PROFILE

Anita Goel
Dyal Singh College
67 PUBLICATIONS **413** CITATIONS
SEE PROFILE

Partial Migration for Re-architecting a Cloud Native Monolithic Application into Microservices and FaaS

Deepali Bajaj[1](✉), Urmil Bharti[1], Anita Goel[2], and S. C. Gupta[3]

[1] Department of Computer Science, Shaheed Rajguru College of Applied Sciences for Women, University of Delhi, Delhi, India
deepali.bajaj@rajguru.du.ac.in, ubharti@hotmail.com

[2] Department of Computer Science, Dyal Singh College, University of Delhi, Delhi, India
goel.anita@gmail.com

[3] Department of Computer Science, Indian Institute of Technology, Delhi, India
scgupta@cse.iitd.ac.in

Abstract. Software development paradigm is transitioning from monolithic architecture to microservices and serverless architecture. Keeping monolithic application as a single large unit of scale is a threat for its agility, testability and maintainability. Complete migration of a monolithic application to microservice or serverless architecture poses additional challenges. Design and development of microservices is complex and cumbersome in comparison to monoliths. As number of microservices increase, their management also becomes challenging. Using serverless platforms can offer considerable savings, however it doesn't work for all types of workload patterns. Many a times, it may be more expensive in comparison to dedicated server deployments, particularly when application workload scales significantly. In this paper, we propose partial migration of monolith application into microservices and serverless services. For the purposes of refactoring, we use web access log data of monolith application. Our proposed architecture model is based on unsupervised learning algorithm and aims to optimize the resource utilization and throughput of an application by identifying the modules having different scalability and resource requirements. This facilitates segregation of services of a monolith application into monolith-microservice-serverless. We have applied our proposed approach on a Teachers Feedback monolith application and presented the results.

Keywords: Microservices · Decomposition · Refactoring · K-means clustering algorithm · Web access log mining · Microservices architecture · FaaS · Serverless

1 Introduction

Cloud offers rapid elasticity to commensurate with inward and outward demand and swift setting of virtual servers. Because of these benefits, many business companies from small to medium and large sizes are opting cloud service model for deployment of their web applications. Despite these gains, most of applications deployed on cloud could not reap the complete benefits. As just by dumping the existing legacy applications to a virtualized environment can't be regarded as a true cloud native application. In this scenario, system administrators and developers are responsible for maintenance of software stack on these virtual servers which is a burdensome task for them.

Most of the legacy applications were designed using monolithic architectures. These applications are deployed on cloud web servers as a single codebase consisting of different services/modules. Such applications are convenient to develop, test and deploy till its codebase is small and manageable. As application demand increases or its size grows, maintenance of such monolith applications becomes tricky [1]. In this architectural design, a single defective service can bring down the entire software application.

To resolve these issues of monolithic architecture, a new way of designing software applications emerged which is known as microservices. In microservices architecture, all the business requirements of an application are divided into smaller services. Each service works like an independent component and runs in its own container such as Docker, Amazon EC2 Container Service (ACS), Google Container Engine (GKE), Azure Container Service [2]. Each microservice is designed on single responsibility principle i.e. do just one thing and do it properly [3, 4]. Microservice architectural pattern is free from single point of failure and exercise continuous integration and delivery (CI/CD) practices as each new deployment only affects the microservice being updated and not others [5]. This architecture leverages many advantages like scalability, reliability and agility. Ease of deployment and maintenance, resilience to failure and reduce time to market are also considerable benefits of this architecture. Because of its evolutionary design and inherent advantages, microservice architecture is being embraced by many prime technological companies such as Amazon, Netflix, LinkedIn, SoundCloud, Gilt and eBay [6]. Nevertheless, few drawbacks are also associated with microservices design: (i) since everything is an autonomous service which results in high latency in remote calls between services (ii) testing of microservices based application is very complex (iii) since all services generate their own set of logs so debugging is also painful in comparison to monolithic applications.

Recently serverless computing is also gaining traction. A serverless architecture splits an application into a set of even smaller granular services. In this architectural style, developers don't own the burden of setting and administering servers dynamically based on demand which was an overwhelming task in other two service models discussed previously. Sever and infrastructure management is done by cloud service providers. Because of this ease, an application can be developed at a rapid pace with reduced costs. Developers may write the cloud functions packaged as Function-as-a-Service (FaaS) in a variety of languages (i.e. Python, Java, JavaScript, Go) and selects the triggering events for their function. Resource instance type selection, scalability, deployment options, fault tolerance, logging and monitoring is handled by serverless computing platform. Some of the most common serverless platforms are AWS Lambda [7], Microsoft Azure functions [8], Google Cloud Functions [9] and IBM Cloud Functions [10]. Serverless cloud computing execution model also have certain restrictions [11] (i): FaaS functions have limited execution time and when timeout limit is reached, serverless platform abruptly terminates its execution (ii) Functions connect to cloud backend services across a network interface that consumes significant network bandwidth (iii) Maintaining function state across multiple client calls requires writing the current state to slow storage. Looking at these constraints, services that are stateless, isolated, ephemeral, limited in resource requirements and execution time are perfect use cases for FaaS [12].

Thus all the service models have their own strengths and weaknesses that can be exploited for cloud native applications. To make efficient use of these new architectural styles, several migration techniques have been proposed in literature but they all recommend complete migration of an application into microservices or serverless platforms. These migration techniques can be broadly classified as black box and white box. In white box migration technique, static characteristics of an application like design documents, source code files, revision history etc. are used in migration decisions. In black box migration technique, dynamic characteristics of an application like web access log, user usage log, execution traces etc. are the main attributes which guide the migration process [13].

In this paper, we propose a partial migration approach for re-factoring a monolithic application that allows some services of the application to be deployed as microservices, some as FaaS while the rest of the services may remain in monolithic application. Rationale behind this three-in-one architectural pattern is services that are highly scalable and resource intensive can be implemented as microservices. Stateless, ephemeral, short lived modules can be deployed as FaaS. More precisely, the services that are not very popular and consume limited resources can be easily migrated to a serverless platform instead of using a traditional container based implementation. This approach devoid re-factoring a legacy system to run entirely as either microservices or serverless cloud functions. So it would achieve significant gains in terms of resource utilization and better throughput. Our proposed approach of service model selection in terms of s will yield better optimized performance for a web application. Main contributions of the paper are as follow:

1. Proposed a black-box partial migration approach based on web access log for identifying workload patterns of different modules in a cloud native web application. Dimensions considered for workload identification of a module are its scalability and resource requirements.
2. Devised a selection model that will provide guidance for mapping different application modules to suitable service models (monolithic/microservices/serverless).
3. The proposed approach has been applied on a case study of Teachers Feedback Web Application originally developed as a monolithic application.

2 Related Study

It has become harder to maintain legacy monolithic applications because of tight coupling among their internal software components. Modifying a feature in one component causes ripple effects in several other components as well, leading to increased development time and programmers' effort. To address these issues, decomposition of a monolith has been proposed by researchers. Decomposition is a process of re-factoring existing codebase into small and independently deployable modules to improve maintainability and fault tolerance. However monolith application decomposition is a crucial and complex task [14]. Several research papers have been published describing manual, heuristics based, semi automated and automated techniques for this. But all approaches discussed so far, talk about complete migration of monolith to microservices and serverless architectures. Prime objective of discussing their research work is to understand the strategies available in literature that are used for complete migration of large monolithic applications.

Few researchers have done experiments to compare application performance on different service models. Takanori et al. [15] compared the monolithic and microservices implementation of Acme Air, a fictitious airlines web application. They verified that the response time of client request in microservices is significantly increased than the monolithic for two popular language runtimes Java EE and Node.js. Villamizar et al. [16] compared the running cost of a web application in three different deployment options: (i) monolithic architecture (ii) microservice architecture managed by customer and (iii) microservice architecture offered by cloud service providers. Their experimental results show that microservices adoption helps in reducing infrastructure costs when compared with monolithic architectures. Hasselbring et al. [17] studied and presented different aspects of microservice architecture like scalability, reliability, and agility. Costa et al. [18] discusses the pros and cons of microservice over the monolithic architecture. Balalaie et al. [19] discussed their experience report of the migration process of monolithic system to microservices architecture. They observed that microservices architecture offers more flexibility, scalability and availability but brings in new complexities. Adzic et al. [20] examines cost model of migrating two online application to AWS Lambda architecture and they observed significant cost reductions. Manner [21] discussed two conflicting parameters: cost and performance in terms of FaaS adoption.

Some researchers observed that slicing of monolithic application done on the basis of business domain may not find highly loaded parts of the system [13, 22]. The load predictions done by the software architects may not be enough for more complex use cases and should be supplemented with users' access records for the web pages. Mining web usage logs to dig out useful details regarding user behavior has been investigated in research. Muhammad et al. [13] devised a novel approach to automatically decompose a monolithic application into microservices using web access logs of the application for improved user experience and scalability. They also proposed a dynamic method for the selection of the suitable virtual machine to deploy microservices so as to enhance the performance of microservices. Mustafa et al. [22] suggested black-box based technique that extends utilization of web usage mining techniques and considered non functional requirements like performance and scalability for comparison against monolithic implementation.

All the existing approaches explain about complete migration of monoliths to microservices or serverless architectures using either black box or white box techniques. We did not come across any research work that discusses partial migration of monolithic application to microservices and serverless frameworks. In order to address this research gap, we are proposing a partial migration approach that will refactor an existing monolithic codebase to suitable service model. Our idea is to implement scalable and resource intensive components as microservices and less scalable components with short execution time as FaaS. Components with moderate scalability and resource requirements may remain as monolith. With the best of our knowledge, this is the first and novel approach for re-factoring monolithic systems into microservices and FaaS.

3 Proposed Partial Migration Approach for Re-architecting a Monolithic Application into Microservices and FaaS

Having identified various inadequacies in monolith, microservices and serverless platforms, our partial migration approach identifies services and suggests deploying them on best suited service model. Figure 1 shows the complete view of the partial migration approach for re-factoring a monolithic application to microservices and FaaS.

Our approach is based on mining web server access logs of a monolithic web application to predict the scalability and resource requirements of its various components. This will give an insight about highly loaded services, small independent services that are less scalable (with restricted resources requisite) and services with consistent workload (with predicted resources requisite). Our migration approach involves three main steps:

1) Preprocessing of Web Access Log
2) Segregation of URIs
3) Identification of service models

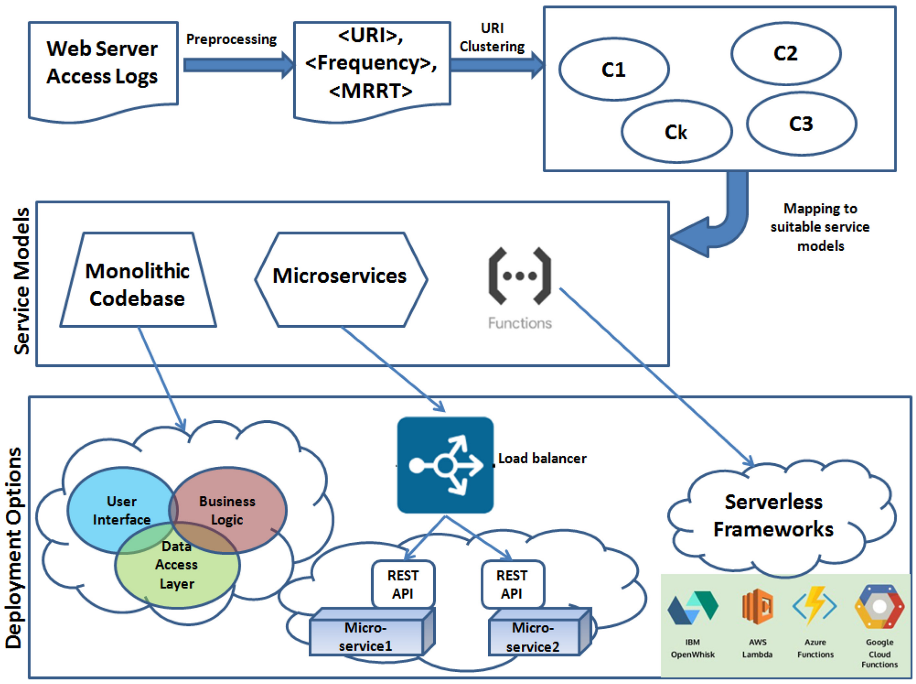

Fig. 1. Partial migration approach for refactoring a monolithic application to Microservices and FaaS

3.1 Preprocessing of Web Access Log

Our approach commences with the collection of web server access log of a monolith web application. The log data must be sufficient enough to reflect the overall application usage statistics. To achieve this, log data should be collected for one complete application execution cycle. This step will ensure inclusion of logs of all services requested by clients so that log mining will obtain unbiased results.

Common format of web server access log file includes Client IP, requested resource URI, time stamp values, HTTP status code, size of document object returned to the client (in bytes) [23]. In addition to default parameters, web servers can be configured to retrieve other log parameters values as well. In our approach, we have used one such configurable parameter i.e. request service time. Value of request response time indicates the time spent to serve a client request (in microseconds) and is also known as request response time (RRT).

Pre-process the collected log data by extracting out only two desired fields i.e. URIs and Request Response Time. The URI frequency from pre-processed logs will reflect scalability requirements and request response time will indicate hardware resource requirements like CPU, disk, network etc. to fulfill a client request.

3.2 Segregation of URIs

From processed log data, segregate all distinct URIs and calculated their frequency of occurrence. Frequency of a URI corresponds to its popularity and thus indicates scalability requirement. Next task is to calculate Mean Request Response Time (MRRT) for each distinct URI. Figure 2 shows sample URIs, their frequency of occurrence and MRRT values.

URIs	Frequency	MRTT
/FeedbackApp/login/	6866	325
/FeedbackApp/	12756	90
/FeedbackApp/feedback/	28716	618
/FeedbackApp/logout/	2016	78
/FeedbackApp/change_password/	4126	92
/FeedbackApp/teacher_analytics/	975	108
/FeedbackApp/teacher_view/	525	515
/FeedbackApp/feedbackStatus/	1020	87
/FeedbackApp/feedbackStatus_view/	510	378
/FeedbackApp/principle_analytics/	413	113
/FeedbackApp/top_five_teachers/	12	973
/FeedbackApp/analytics/	186	606
/FeedbackApp/feedback/VH	6	193
/FeedbackApp/feedback/VH/Chatbot	3	185

Fig. 2. Sample URIs, their frequencies and MRRT values

Our migration approach apply mining technique on processed data to find out similar URI clusters that can be migrated to suitable service models. We propose a URI clustering technique on web access log taking into consideration two clustering parameters (i) mean request response time and (ii) frequency of invocation of each request. We apply URI clustering by partitioning URIs into K discrete clusters $\{C_1, C_2, \ldots, C_K\}$ such that the resource and scalability requirements remain uniform within each cluster. Precisely, each partition corresponds to URIs having comparable request response time and frequency of occurrence.

To identify the URIs clusters, we use unsupervised K-means clustering algorithm wherein URIs are partitioned into pre-defined disjoint non-overlapping K clusters. This is a hard clustering technique in which each data point belongs to only one cluster. Our approach also demands that each URI should be a member of only one cluster so we selected K-means clustering algorithm. This algorithm aims to make all the inter-cluster points as close as possible and at the same time, keeping different clusters as distant as possible. Data points corresponding to URIs are assigned to a cluster so that Euclidean distance (sum of squared distance between data points and clusters' centroid) are minimized [24]. Euclidean Distance can be represented in Eq. 1:

$$\text{Euclidean Distance} = \sqrt{(X_F - F_1)^2 + (X_{MRTT} - MRRT_1)^2} \qquad (1)$$

where X_F: Observation value of variable Frequency, X_{MRRT}: Observation value of variable MRRT, F_1: Centroid of cluster 1 for variable Frequency, $MRRT_1$: Centroid of cluster 1 for variable MRRT.

As the scale of measurement of clustering parameters affects Euclidean Distance, so variable standardization becomes essential to avoid biased clustering. Standardization is a method wherein all the parameters are centered around zero and have nearly unit variance. For this we subtracted each dataset value by mean and then divide with standard deviation as shown in Eq. 2.

$$\hat{\chi} = \frac{(\mathrm{X} - \mu)}{\sigma} \qquad (2)$$

An elementary step for any unsupervised learning algorithm is to find the appropriate number of clusters into which data must be clustered. The most common method used for K-means algorithm is Elbow Method. We also used Elbow method to calculate the appropriate number of clusters K. We plot WSS value (within-cluster sums of squares) and number of clusters K. The location on the plot that shows a knee bend is considered as an indicator of optimal value of K [25]. WSS is a measure of compactness of a cluster and can be calculated as sum of squared distance between all data points xi of a cluster j and the centroid of cluster j, as shown in Eq. 3.

$$\mathrm{WSS(K)} = \sum\nolimits_{j=1}^{K} \sum\nolimits_{x_j \in clusterj} \|Xi - \bar{X}j\|^2 \qquad (3)$$

where $\bar{X}j$ is mean in cluster j.

Once we get an optimal value of K, we can implement K-mean clustering to get K number of clusters. URIs that is grouped in one cluster has similarity in terms of scalability and resource requirements.

3.3 Identification of Service Models

In this step, identified clusters are mapped to monolith, microservices and FaaS service models. In general, highly scalable and modifiable components of a monolith are good candidates for microservices. At the same time isolated, ephemeral and independent components are good candidates for FaaS in serverless service model.

Frequency of a URI is proportionate to its scalability need and MRRT value reflects resource usage to serve clients request. We suggest that URIs having high frequencies can be deployed as microservices. Further, URIs with moderate frequency and high resource usage can also be considered for microservices deployment. Similarly URIs having low frequencies and moderate to low resource requirements can be deployed as FaaS. Rest of the URIs can remain in monolithic deployment. Table 1 provides pragmatic guidance about suitable service models.

Table 1. Service model selection on the basis of scalability and resource usage.

Scalability need	Resource usage	Service model
High	High	Microservice
High	Moderate	Microservice
High	Low	Microservice
Moderate	High	Microservice
Moderate	Moderate	Monolith
Moderate	Low	*Monolith/FaaS
Low	High	Monolith
Low	Moderate	FaaS
Low	Low	FaaS

* Dependent on Cloud Service Provider threshold limits for execution time and resource usage.

4 Case Study

In this section, we will present our case study of Teachers Feedback Web Application which is developed to collect teachers' feedback in a university environment. We have applied our proposed methodology on this system to prove our results. This case study can be considered as a proof of concept (PoC) for our partial migration approach.

4.1 Teachers Feedback Web Application

We have validated our approach using a case study "Teachers Feedback Web Application". In order to fine tune the process of teaching-learning and for establishing a positive learning environment, constituent colleges of the University of Delhi employs a feedback mechanism. In this spirit, we developed Teachers Feedback Web Application (TFWA) to automate the feedback process. TFWA is developed on Model View Controller (MVC) architectural pattern which is a popular web application design pattern. It is developed using Django framework 3.0.2 which is a Python-based free and open-source web framework [26]. Django framework officially supports many database backends like PostgreSQL, MariaDB, MySQL, SQLite. In our implementation we have used SQLite as database backend which is a default option supported by Django. For our PoC we deployed TFWA on Apache HTTP Server 2.4.41 (https).

Different actors of this system are Student, Administrator, Teacher In-Charge and Principal. Students use TFWA to record their feedback at the end of every semester for all teachers. Feedback is collected for all subjects taught by either same or different teachers in a course. Feedback is collected on fifteen different questions where a student can mark her response on a scale of 1 to 5. A special use case has also been implemented in which a visually impaired student can also provide her feedback using a chatbot facility that is being developed using conversation technology via audio messages. Teacher In-Charge of every department can track the status of the feedback process to ensure the participation of all the students. TIC can view and analyze the feedback data from different perspectives i.e. teachers-wise feedback summary/details, subject-wise feedback summary/details and course-wise feedback summary/details.

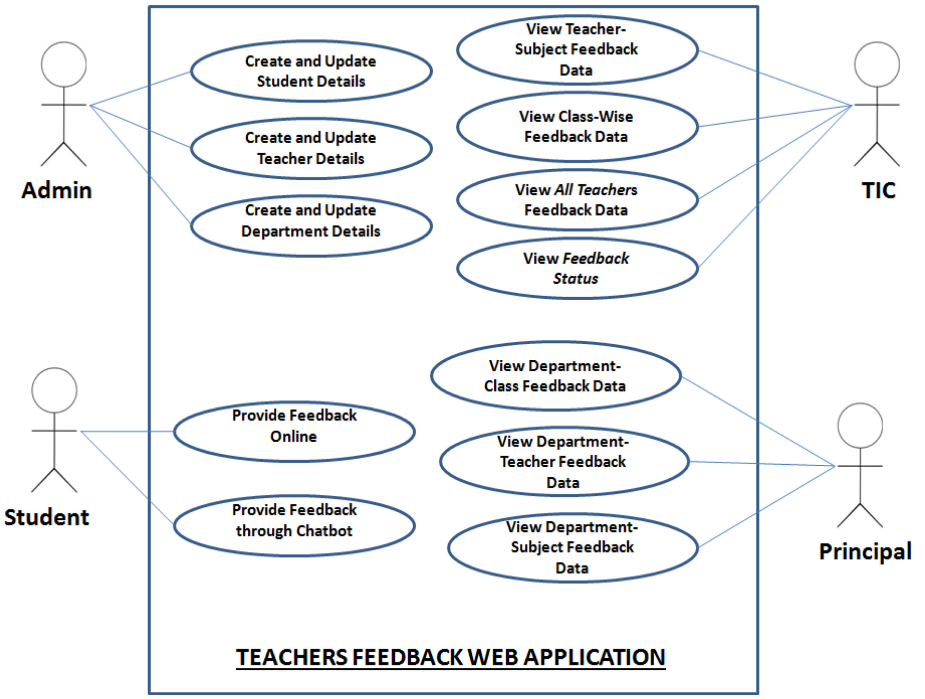
Admin
Create and Update Student Details
Create and Update Teacher Details
Create and Update Department Details
View Teacher-Subject Feedback Data
View Class-Wise Feedback Data
View *All Teachers* Feedback Data
View *Feedback Status*
TIC
Student
Provide Feedback Online
Provide Feedback through Chatbot
View Department-Class Feedback Data
View Department-Teacher Feedback Data
View Department-Subject Feedback Data
Principal
TEACHERS FEEDBACK WEB APPLICATION

Fig. 3. Use case diagram for teachers feedback web application

Principal of the college can also view and analyze feedback data of the whole college from different perspectives i.e. department-wise feedback summary/details, teachers-wise feedback summary/details, subject-wise feedback summary/details and course-wise feedback summary/details. Principal can also view the top five teachers, as per student feedback, of the college. Administrator of the application can populate students, teachers, departments, courses and principal data into the system by import facility provided in the application. Administrator can also insert or update any of these entities if any correction is required after import operation. Figure 3 shows use case diagram for TFWA.

4.2 Methodology in Practice

Based on the system requirements of TFWA presented above, in this section we have discussed the application of our proposed approach for partial migration as PoC for our case study.

We collected web server access logs of one complete execution cycle of Teacher Feedback Web Application. These logs have feedback from all the students (for all teachers and their respective subjects) and multiple views of feedback data by TIC and principal and hence the entire URI space is collected as per the real usage of application. Next step was to preprocess the web server access logs to retrieve frequency of each URI and its MRRT. Since both parameters differ in their unit and scale which may unduly influence the migration process so we performed variable standardization to transform data to a common scale. This scaling will not distort the differences in original ranges of values. This standardization preprocessing has been discussed in Subsect. 3.2.

To apply K-means clustering technique on our URI space, we need to get an idea about the optimal number of clusters (K) for the K-means algorithm. We addressed this concern by applying the Elbow method in which we iterate the value of K and calculate within-cluster sums of squares (WSS) as shown in Eq. 3. We clearly noticed that WSS value drops sharply when K is smaller than elbow point 3 and decreases slightly afterwards. Therefore, we opted 3 as an optimal value for K in TFWA. Figure 4 shows variation of WSS with an increasing number of K. URI space distribution is shown in Fig. 5 after applying K-mean clustering with $K = 3$.

Boundaries of the clusters are distinguishable without any overlap. This demonstrates that identified clusters can be implemented in different service models as suggested in Table 1. Figure 5 shows service model selection for TFWA where each data point represents an application URI (a set of sample URIs is given in Fig. 2 for reference).

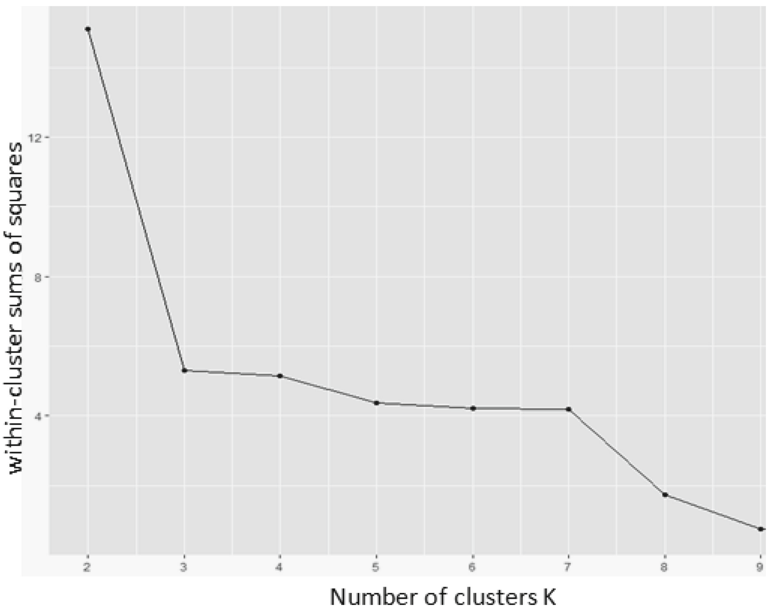

Fig. 4. Elbow method - WSS vs. number of clusters

4.3 Results

In the previous subsections, we have applied our proposed methodology on TFWA. As an outcome, we have identified possible refactoring for microservices, FaaS and monolith deployments. Our findings are as follow:

Microservice Potential Candidates: URIs with high frequency indicate high scalability requirements so can be deployed as microservice architectures. As shown in Fig. 5 cluster C1 contains all the URIs involved in capturing feedback of teachers. Since, *Capture Feedback* functionality of TFWA is a highly scalable component as thousands of students give their feedback in parallel for all teachers taught them in the current semester. So it is a judicious decision to re-factor this component from monolithic code base and implement it as a separate microservice.

FaaS Potential Candidates: URIs with very less frequency and low resource requirements can be mapped to serverless service model. It means services associated with these URIs can be implemented as Function-as-a-Service in a serverless platform. As shown in Fig. 5, cluster C3 contains all the URIs that occurred least number of time in log data. Data points of this cluster represent the URIs of chatbot service. In our application, *ChatBot* functionality to simulate human conversation for providing feedback using voice-based interfaces (speech-to-text and text-to-speech) that is required only for visually impaired students is a good candidate for FaaS.

Other system capabilities of TFWA like *User Authorization*, *Admin Functions* like maintaining department, course, teacher and students details, *Tracking Feedback Status* and various *Feedback Analytics* options available for TIC and Principal can be

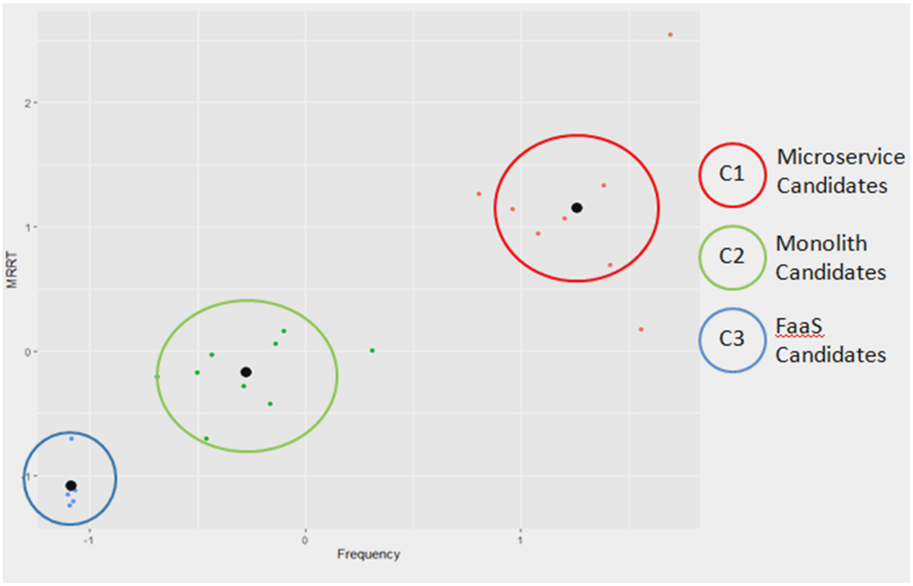

Fig. 5. Identified Clusters and their mapping to service models

considered as monolith candidates as these services have moderate scalability and moderate resource usage requirements. All the URIs of these application capabilities are shown in cluster C2 as shown in Fig. 5.

So our case study establishes our results and validates our proposed approach of partial migration.

Conclusion

In recent years, web applications that require scaling for thousands or millions of concurrent users are becoming common. Traditional enterprise web applications are generally developed as monolith applications, which become hard to develop, test, scale and maintain with time. To alleviate these problems, Microservice and serverless architectures are evolving. Microservices enable creating an application as a collection of autonomous services which interact with each other through lightweight mechanisms like REST API or message bus etc. In a serverless execution model, cloud provider dynamically controls allocation and provisioning of servers. Serverless applications run in stateless compute containers that are event-triggered, ephemeral, and fully managed by the cloud provider itself. Pricing is determined by number of executions at run time and not by pre-provisioned compute capacity as in IaaS solutions.

While re-factoring a monolith application for better scalability and resource utilization, some components of the application can be reorganized as microservices and some as FaaS. Identifying different components of the application that suits to microservice architecture and for FaaS implementation is an unanswered question. To address this issue, we have given an approach that provides pragmatic guidance to system architects to reorganize their monolithic code base into microservices, FaaS and monolith. We applied our proposed technique on a case study "Teachers Feedback Web Application" and proved our approach. Our partial migration procedure uses web server access logs of one complete application execution cycle and then applied unsupervised learning algorithm on these logs to map components that best suits monolith-microservices-FaaS service patterns.

Future Work
In our future work, we will implement above identified services in microservices and serverless architecture to gauge empirically improvement in performance in terms of response time and cost reduction. We are also planning to perform static and dynamic code analysis to identify independent and autonomous services to strengthen the basis of our approach and assess more deeply the benefits of partial migration.

References

1. Iqbal, W., Erradi, A., Mahmood, A.: Dynamic workload patterns prediction for proactive auto-scaling of web applications. J. Netw. Comput. Appl. **124**, 94–107 (2018). https://doi.org/10.1016/j.jnca.2018.09.023
2. Zimmermann, O.: Microservices tenets agile approach to service development and deployment. Comput. Sci. Res. Dev. **32**(3-4), 301–310 (2017). https://doi.org/10.1007/s00450-016-0337-0
3. Fowler, M., Lewis, J.: Microservices a definition of this new architectural term, p. 22 (2014). http//martinfowler.com/articles/microservices.html
4. Mazlami, G., Cito, J., Leitner, P.: Extraction of microservices from monolithic software architectures. In: Proceedings of the 2017 IEEE 24th International Conference on Web Services, ICWS 2017, pp. 524–531 (2017). https://doi.org/10.1109/icws.2017.61
5. Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Gil, S.: Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud (Evaluando el Patrón de Arquitectura Monolítica y de Micro Servicios Para Desplegar Aplicaciones en la Nube). In: 10th Computing Colombian Conference, pp. 583–590 (2015). https://doi.org/10.1109/columbiancc.2015.7333476
6. Garriga, M.: Towards a taxonomy of microservices architectures. In: Cerone, A., Roveri, M. (eds.) SEFM 2017. LNCS, vol. 10729, pp. 203–218. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-74781-1_15
7. https://aws.amazon.com/lambda/
8. https://azure.microsoft.com/en-gb/services/functions/
9. https://cloud.google.com/functions/
10. https://developer.ibm.com/api/view/cloudfunctions-prod:cloud-functions:title-Cloud_Functions#Overview
11. Hellerstein, J.M., et al.: Serverless computing: one step forward, two steps back. In: The 9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, vol. 3 (2019)
12. Spillner, J., Mateos, C., Monge, D.A.: FaaSter, better, cheaper: the prospect of serverless scientific computing and HPC. In: Mocskos, E., Nesmachnow, S. (eds.) CARLA 2017. CCIS, vol. 796, pp. 154–168. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73353-1_11

13. Abdullah, M., Iqbal, W., Erradi, A.: Unsupervised learning approach for web application auto-decomposition into microservices. J. Syst. Softw. **151**, 243–257 (2019). https://doi.org/10.1016/j.jss.2019.02.031
14. Fritzsch, J., Bogner, J., Zimmermann, A., Wagner, S.: From monolith to microservices: a classification of refactoring approaches. In: Bruel, J.-M., Mazzara, M., Meyer, B. (eds.) DEVOPS 2018. LNCS, vol. 11350, pp. 128–141. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-06019-0_10
15. Ueda, T., Nakaike, T., Ohara, M.: Workload characterization for microservices. In: Proceedings of the 2016 IEEE International Symposium on Workload Characterization, IISWC 2016, pp. 85–94 (2016). https://doi.org/10.1109/iiswc.2016.7581269
16. Villamizar, M., et al.: Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures. In: Proceedings of the 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016, May 2016, pp. 179–182 (2016). https://doi.org/10.1109/ccgrid.2016.37
17. Hasselbring, W., Steinacker, G.: Microservice architectures for scalability, agility and reliability in e-commerce. In: 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017, pp. 243–246 (2017)
18. Costa, B., Pires, P.F., Delicato, F.C., Merson, P.: Evaluating REST architectures—approach, tooling and guidelines. J. Syst. Softw. **112**, 156–180 (2016)
19. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Migrating to cloud-native architectures using microservices: an experience report. In: Celesti, A., Leitner, P. (eds.) ESOCC Workshops 2015. CCIS, vol. 567, pp. 201–215. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33313-7_15
20. Adzic, G., Chatley, R.: Serverless computing: economic and architectural impact, pp. 884–889 (2017). https://doi.org/10.1145/3106237.3117767
21. Manner, J.: Towards performance and cost simulation in function as a service. In: Proceedings of ZEUS 2019 (2019)
22. Mustafa, O., Marx Gómez, J.: Optimizing economics of microservices by planning for granularity level - Experience Report. In: 2017 Programming Technology for the Future Web, April 2017, p. 6 (2017)
23. Aragon, H., Braganza, S., Boza, E.F., Parrales, J., Abad, C.L.: Workload characterization of a software-as-a-service web application implemented with a microservices architecture. In: Web Conference 2019 - Companion World Wide Web Conference WWW 2019, pp. 746–750 (2019). https://doi.org/10.1145/3308560.3316466
24. Hussain, T., Asghar, S., Masood, N.: Web usage mining: a survey on preprocessing of web log file. In: 2010 International Conference on Information and Emerging Technologies, pp. 1–6 (2010)
25. Bholowalia, P., Kumar, A.: EBK-means: a clustering technique based on elbow method and k-means in WSN. Int. J. Comput. Appl. **105**(9), 17–24 (2014)
26. Forcier, J., Bissex, P., Chun, W.J.: Python Web Development with Django. Addison-Wesley Professional, Boston (2008)