Performance Analysis of Microservice Design Patterns

Akhan Akbulut
North Carolina State University
Istanbul Kültür University

Harry G. Perros
North Carolina State University

***Abstract*—Microservice-based solutions are currently gaining momentum because they do not have the disadvantages of traditional monolithic architectures. Business interest in microservices is increasing since the microservice architecture brings a lightweight, independent, reuse-oriented, and fast service deployment approach that minimizes infrastructural risks. This approach is at an early stage of its development, and in view of this, it is important to understand the performance of its design patterns. In this article, we obtained performance results related to query response time, efficient hardware usage, hosting costs, and packet-loss rate, for three microservice design patterns practiced in the software industry.**

The service-oriented architecture (SOA) is the most widely used concept in software design. This is because it enables high service reusability, reliability, improved scalability and availability, heterogeneity, and platform independence. SOA is currently evolving to encompass microservices so as to address problems related to the boundaries of the monolithic architecture.[1] Application requirements have given rise to the need for the lightweight and high-scalability option provided by the microservice architecture. Microservices break the logic of an application into a number of independent services that run as separate processes. This approach can lead organizations to the realization of a true SOA that will permit business processes and data to become more available.

Enterprises that have recognized the benefits of microservices have started to replace their existing legacy software with microservice-based solutions. In view of this, it is very critical to accurately determine the application

Digital Object Identifier 10.1109/MIC.2019.2951094
Date of publication 4 November 2019; date of current version 17 January 2020.

architectures for effective use of this relatively new technology. The success of microservice architecture depends on the use of design patterns. A number of patterns have been proposed in MuleSoft's whitepaper,[2] Arun Gupta's blogs,[3] and in some books.[4–6] In this article, we report on the performance results we obtained by implementing various applications using the API Gateway, chain of responsibility, and asynchronous messaging design patterns. The performance results are related to the query response time, efficient hardware usage, hosting costs, and packet-loss rate.

PERFORMANCE ANALYSIS

In our testbed, we used Node JS v.8.11.4 and Python v.4.7.0 for the implementation of the microservices, MongoDB v.4.0 for the data source platform, and Docker Compose v.1.22.0 for the containerization environment. The Docker environment was deployed on an Ubuntu 16.04 LTS Base system in the Virtual Computing Lab, North Carolina State University.[7] Docker containers were restricted by *cgroups* so they do not consume the full resources of the testbed. They only worked as virtually configured hardware.

All experiments use the following three main components: a tester, an analyzer, and web resources implemented in microservices related to the experiment. The tester generates web customer requests that are executed by the microservice environment. In all the experiments, we evaluated the response time of an implemented application, i.e., the time it takes to process a request, by subjecting the application to a maximum load. To this effect, the tester was implemented as a multithreaded structure, which generates a predetermined number of threads. Each thread issues 3000 requests back-to-back in an asynchronous manner, and when all the responses to the requests are received, the thread is deleted.

Each thread records the time at which a request is issued and the time at which the response to the request is received in a text file. All the text files generated in the experiment are processed by the analyzer in order to calculate the mean and confidence interval of the response time, and also the 95th percentile of the response time. The latter metric is a value of the response time such that only 5% of the sample response times are higher. We note that this is a more meaningful metric in Service Level Agreements (SLAs) than the mean, since it provides a statistical upper bound. Other percentiles, such as 99%, can also be used.

Case Study 1—The API Gateway Design Pattern

This pattern consists of a gateway through which a number of different subservices are accessible. An API gateway acts as a single entry point for requests and based on the nature of a request, it invokes an appropriate subservice.[8] In addition to routing and aggregation, an API gateway performs two important tasks, namely, gateway offloading and circuit breaking. Case Study 1 is an implementation of an online theater system using the API gateway pattern on a four-core Intel CPU (2.0 GHz) with 16 GB RAM server. The system consists of four services. The Movies service provides general information, and the Sessions service gives a list of the show times of the movies and the theater name. The tickets service is used to purchase a ticket, and the Reservations service allows a customer to reserve a seat for a show time of a movie.

All content and methods provided by the online theater system application are accessed from two points: a store front and an API gateway. Each microservice executes a lightweight REST mechanism in order to provide the necessary output to the store front and the API gateway. The API gateway is used by entities that do not need to access the visual elements. For instance, a self-service kiosk ticketing device in a cinema accesses the services via the API gateway.

Figure 1(a) gives the mean response time, as a function of the number of threads. The confidence intervals are also plotted as vertical lines. The 95th percentile and confidence interval of the observed response times for each number of threads are also given in this figure. The confidence intervals were computed using the batch means method, with a batch size equal to 1000 observations.

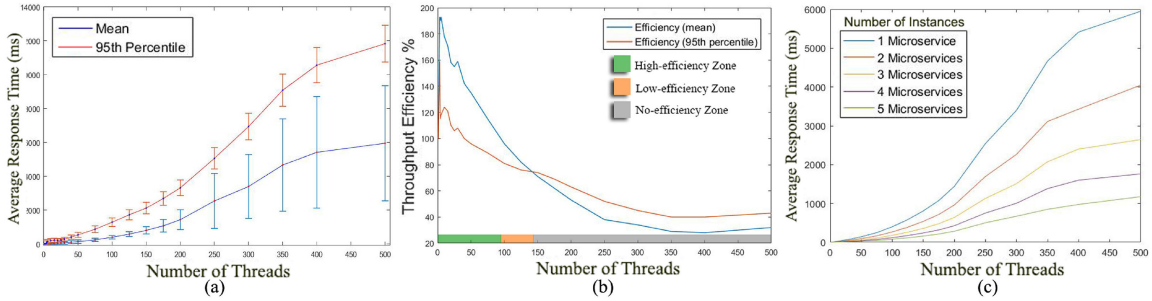

Figure 1. Performance results of Case Study 1. (a) Mean and 95th percentile of the response time versus number of threads. (b) Efficiency curves using the mean and the 95th percentile of the response time. (c) Average service times for different number of instances.

The 95th percentile gives us an idea of the tail of the response time distribution. The further away it is from the mean, the longer the tail is. For instance, for a single thread, the average response time is 3.82 ms, whereas the corresponding 95th percentile is 10.48 ms. Similarly, for 100 and 200 threads, we have a mean response time of 399.89 ms and 1295 ms, respectively, with corresponding 95th percentiles of 2544.32 ms and 5077.85 ms. These numbers indicate a large tail of the probability distribution of the response time, which is due to the fact that we basically submit all the requests from all the threads as soon as possible, thus creating a huge backlog, which in turn causes the response time of the requests toward the end of the queue to increase quite dramatically. We note that the percentile of a performance metric is more meaningful in SLAs than its corresponding mean.

Using the mean response times for the different number of threads, we calculated the following efficiency metric. Let R_i be the mean response time when there are i threads. Then, we define the efficiency metric E_i, as follows:

$$E_i = ((R_1 * i)/R_i) * 100. \quad (1)$$

Also, using the 95th percentile values $R_{0.95,i}$, we calculated a similar efficiency metric using the expression:

$$E_{0.95,i} = ((R_{0.95,1} * i)/R_{0.95,i}) * 100. \quad (2)$$

Figure 1(b) shows the two efficiency plots as a function of the number of threads. We observe that the efficiency based on the mean response time E_i increases from 1 thread to 18 threads, and then from 19 to 94 threads it continuously decreases until at 94 it becomes equal to E_1. Using this plot, we classify the microservice implementation as high, low, and not efficient, as follows. High efficiency is achieved for the range of threads from 1 to 94, since the efficiency is over 100%. Low efficiency is achieved from 95 to 141 threads, since the efficiency is between 100% and 75%. In this case, the service time is slightly longer, but it is still acceptable. Finally, for more than 142 threads, the efficiency is less than 75%, and the service is classified as not efficient.

A slightly different picture emerges when we use the efficiency plot based on the 95th percentile of the response time. High efficiency is achieved from 1 to 83 threads, and low efficiency is achieved from 84 and 141 threads. The high-efficiency range is shorter than the one based on the mean response time, since we use the 95th percentile of the response time, which is an upper bound. Both efficiency plots have the same cutoff of 141 threads, which corresponds to 75% efficiency considered as the lowest acceptable level for service requirements.

Obviously, if we want to decrease the noneffi-cient zone, we need to use more than one instantiation of the microservice when the number of threads exceeds 141. To that effect, Figure 1(c) shows the mean response time of the microservice implementation for k instantiations, where $k = 1$--5. Horizontal scaling allows the average service time to decrease or remain constant as the number of threads increases. A similar set of curves can be obtained using the 95th percentile of the response time. In a monolithic implementation, scalability cannot be performed after a certain level even if the resources are enhanced. Conversely, microservice architectures offer

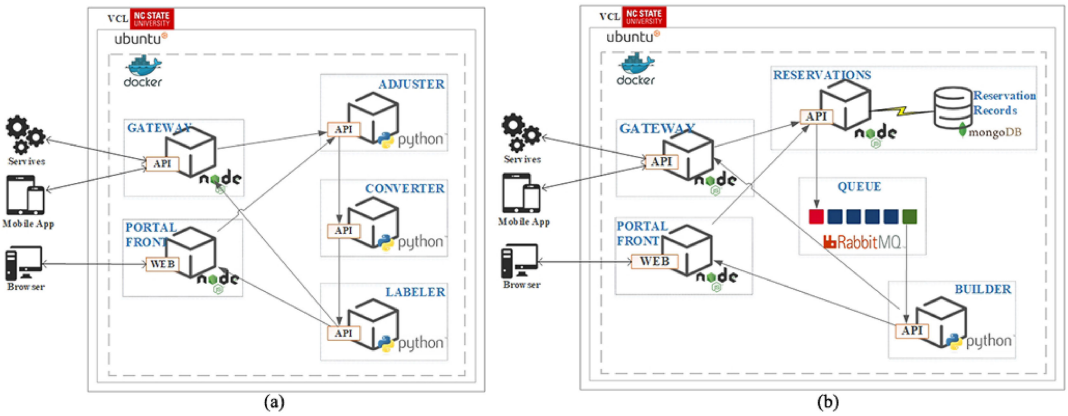

Figure 2. Testbed. (a) Case Study 2. (b) Case Study 3.

scalability at a very high level by replicating microservices.

We note that the efficiency of a microservice implementation depends on the number of coprocessors. Multicore processors are appealing to microservice architectures, but they incur a higher hosting cost. Another important factor is that the misconfiguration of RAM can cause excessive memory swapping, which leads to longer response times.

The case study data reveal that the most ideal architecture for an equally balanced scenario is the API gateway design pattern. It also provides flexibility when we want to manage requests from multiple channels. It facilitates the execution logic of synchronous communications between equally balanced services and provides scale-up artifacts. Separation of concerns along with distributing the load over multiple service instances provides the ultimate black boxed service experience in a polyglot configuration. Polyglot programming benefits from services developed with different programming languages over various stacks. The API gateway design is typically used as a point-of-contact layer among other architectures. In fact, a microservice ecosystem without an API gateway is considered as a bad practice or an antipattern. A similar study to this scenario[9] focused on the what-if analysis and capacity planning of microservices. It is also possible to predict the user load[10] using these analyses and monitoring metrics. Finally, version control can be performed with this approach.[11]

Case Study 2—The Chain of Responsibility Design Pattern

In the second case study, we implemented an image-editing application using the chain of responsibility pattern. This design pattern consists of a collection of subservices designed to work together in order to process a request. The subservices are linked together sequentially, so that one subservice's output becomes input to the next subservice. In this scenario, users submit a colored image as an input and request to have it converted to black and white. The application is triggered by first delivering requests from the web channel or API channel to the Adjuster microservice. This microservice, which is the first node of the chain, checks the size of the submitted image and resizes it if it is bigger than 1920 × 1080. The modified image is transferred to the Converter microservice in a byte-array form for the next operation. The second node is responsible for converting the image into a grayscale form.

The Converter microservice passes this image to the Labeler node, which applies a watermark onto the image. Finally, the resized, grayscaled, and watermarked image is returned to the thread that initiated the request. All the services in this scenario were implemented using the Python OpenCV image-editing libraries. A diagrammatic representation of the testbed is shown in Figure 2(a).

In this case study, we compared the CPU and RAM requirements of the microservice implementation against a composite implementation

Table 1. Docker container performance evaluation on composite versus chain of responsibility pattern.

Threads	Composite	Chain of Responsibility			Hosting Diff.
10	2560 MB vRAM - 2* 4.0Ghz vCPU	$^{\alpha}$256 MB - 800Mhz	$^{\beta}$1280 MB - 4.0Ghz	$^{\gamma}$256 MB - 800Mhz	30%
		Total: 1792 MB vRAM - 2* 2.8Ghz vCPU			
20	5376 MB vRAM - 4* 4.0Ghz vCPU	$^{\alpha}$524 MB - 1.8Ghz	$^{\beta}$2620 MB - 3* 3.0Ghz	$^{\gamma}$524 MB - 1.8Ghz	21.25%
		Total: 3668 MB vRAM - 3* 4.2Ghz vCPU			
50	13312 MB vRAM - 10* 4.4Ghz vCPU	$^{\alpha}$1344 MB - 4.2Ghz	$^{\beta}$6720 MB - 5* 4.2Ghz	$^{\gamma}$1344 MB - 4.2Ghz	33.18%
		Total: 9408 MB vRAM - 7* 4.2Ghz vCPU			

$^{\alpha}$Adjuster microservice, $^{\beta}$Converter microservice, and $^{\gamma}$Labeler microservice

for three different sets of threads. The results obtained are given in Table 1. The column labeled "Chain of Responsibility" lists the RAM and CPU requirements per microservice and also the total for all three microservices for 10, 20, and 50 threads. The column labeled "Composite" lists the RAM and CPU requirements for the composite implementation for the same number of threads. The CPU and RAM for the composite implementation were obtained experimentally by varying them until the throughput of the composite implementation became equal to that of the microservice implementation. The third column labeled "Hosting Diff" lists the percentage by which the hosting cost will increase if we used the composite implementation based on hosting prices from Amazon web services.[12] In all considered scenarios, the CPU's utilization was 10%–40%.

The same application may require different RAM and CPU for different design pattern implementations. Therefore, the design pattern is an important decision that has a direct impact on hosting costs. Table 1 shows that the cost difference for a different number of users varies between 21.25% and 33.18%. For enterprise-level applications, this difference can amount to thousands of dollars. Even if a private cloud is used for hosting microservices, using an accurate design pattern for an application contributes to green computing.

A *megaservice*[13] is a service that provides many functionalities. It is an antipattern, and it should be decomposed into multiple separated microservices. When a multiple application logic developed via a single code-base is scaled up, all parts are scaled up even those that do not require additional resources. However, if each application logic is implemented as a separate microservice, scale-ups are made for only the needed entities, so that less resource is consumed. The ideal microservice design pattern for functions that process consecutively shared data is the chain of responsibility design pattern. In Case Study 2, we observed that this architecture allows us to scale backend services independently, with a gain on the hardware usage of around 30% compared to the megaservice equivalent structure. The most important issue to guard against in this architecture is the possibility of excessively long chains that may lead to a spaghetti structure.

Case Study 3—The Asynchronous Messaging Design Pattern

The synchronous communication provided by the REST mechanism is simple and well known, easy to test, and firewall friendly. However, it is not ideal for some scenarios because of the blocking of the clients. In view of this, asynchronous messaging and event-driven communications can be implemented to propagate changes between microservices. The asynchronous messaging design pattern is preferred when there is a large volume of data that needs to be processed and also when no immediate response is expected. In this case study, we retained the Reservations microservice from the first case study and we added a Builder node that generates pdf documents. The Reservations microservice transfers user information, such as name, surname, movie name, show time, and seat number, to the Builder, which generates a

ticket in pdf format. The case study was implemented using the asynchronous messaging design pattern, as shown in Figure 2(b). A queue structure is introduced between the Reservation and the Builder microservices using RabbitMQ since the service time of a request in the Builder is much longer than that in the Reservation. A queue in RabbitMQ is an ordered collection of messages that are enqueued and consumed in a FIFO manner. The scenario ends with the transfer of the pdf-formatted ticket to the client. The reason we implemented RabbitMQ instead of any other message broker products is that it provides the state of the messages (consumed, rejected, acknowledged, etc.) in the ecosystem. Most of the message brokers are stateless and assume that the consumer keeps track of what has been consumed. Furthermore, RabbitMQ supports several protocols such as MQTT and STOMP for processing messages in addition to AMQP.

The goal of Case Study 3 is to demonstrate the usefulness of the asynchronous messaging design pattern for transaction-based scenarios involving processes with disparate service times. The capacity of the queue implemented using RabbitMQ is limited since it is allocated a fixed amount of memory. In view of this, messages sent from the Reservations subprocess to the Builder subprocess during the time that the queue is full are lost. In view of this, the queue size has to be fixed so as to minimize the queue overflow. When we inspect the results for the asynchronous messaging implementation for two different configurations, we observed that packet losses are minimized with lower configurations. In the first configuration, we allocated 8.192 GB vRAM and four cores of 3.5 GHz vCPUs and measured the packet loss in the case where a single thread generated 1000 requests back-to-back. We note that the packet loss was 26% of all packets sent to the queue by the Reservations subprocess. Subsequently, we slowly increased the vRAM and vCPU allocation until no packet loss was observed. This occurred when we have allocated 122.288 GB vRAM and six 4 GHz vCPUs.

We compared Case Study 3 to an equivalent composite implementation of the Reservations and Builder microservices. Like the composite structure in Case Study 2, the megaservice that houses both Reservations and Builder microservices' application logic should be scaled-up according to the needs of the Builder part, since it performs more complex computations. This causes an additional unwanted resource allocation to Reservations part as well. The input buffer of each subprocess is also limited due to the finite amount of memory allocated at configuration time. For this implementation, we kept increasing both the vRAM and vCPU allocation until no packets were lost at the Builder. The resulting configuration is 131.073 GB vRAM and ten 4.0 GHz vCPUs. We observe that the asynchronous messaging design pattern requires less memory and CPU in order to achieve zero loss.

An important constraint of data centers is energy consumption. An improper software architecture may increase the CPU utilization, thus increasing the energy consumption. Barroso and Holzle[14] observed in 2007 that processors in data centers operate mostly within a utilization range of 10%–50%. Today, for competitive data centers, this figure is up to 60%. The use of Dynamic Voltage and Frequency Scaling power management mechanisms lead to significant energy reductions (up to 40%) and power savings (up to 20%)[15] for the same utilization levels. In our experiments, we observed that the presence of a messaging system can help to maintain an optimal CPU utilization. In Figure 3, we show the CPU utilization with and without the messaging queue for the first 100 s. In the graph on the left, we observe a 100% utilization for the case when there is no messaging queue in front of the Builder. In the graph on the right, we see that the CPU settles down to around 50%. This was achieved by configuring appropriately the egress rate of RabbitMQ via its management plugin. Operating at 100% utilization causes an unwanted energy consumption.

Apart from orchestrating an efficient operation of consumer services, the messaging system can handle requests from various channels with different operating logic. Similar to the FIFO approach, priority queues can be defined, which allow different priorities according to different user types. In practice, RabbitMQ implements a queue over a single processor. This is

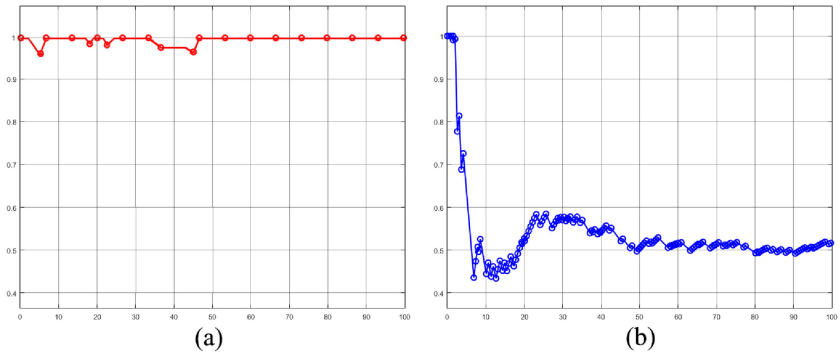

Figure 3. Builder microservice utilization for the API gateway and asynchronous messaging patterns. (a) Utilization without queue. (b) Utilization with queue.

because real-life scenarios are implemented in multiprocessor environments with multiple instances of consumers and queues. It should be noted that in general having a single queue in the ecosystem is considered an antipattern, and it has a negative effect on resource utilization. If the queue service is insufficient and cannot be mirrored, it may be possible to respond fairly to requests from all the channels using a round-robin approach. The benefits of using RabbitMQ is also documented by Hong *et al.*[16] In Case Study 3, we observed that, with the use of a message queue like RabbitMQ, the most important gain is redundancy via persistence. Such messaging systems allow requests to be stored in the system until they are processed by appropriate microservices. They are also useful in the case where microservices do not have access to adequate CPU and RAM resources. This architecture allows a more energy efficient ecosystem for hosting non-time-critical services. Also, the use of a queue allows non-real-time operations to be batched and executed together. For example, it is more efficient to batch 100 database commands and execute them all together rather than executing them one at a time. In addition, different queueing models can be used so as to meet business needs.

LESSONS LEARNED

The impact of adopting a microservice architecture cannot be seen immediately. In order to make a complete evolution, practitioners should carry out the following steps. The unit testing action is the first step for verifying that the services work as intended. This API functionality test covers overall application functionality in the service and completely examine the API with multiple test cases. Using automated testing frameworks such as *NUnit* or *JUnit*, we can increase the depth and scope of the tests and also inspect the different functional pieces of the application. In addition, load tests should be performed periodically to assess the application topology. Load tests are useful to find the bottlenecks in the ecosystem and help to configure the accurate number of running instances of the entities. It is important to use SLA and also have some knowledge of the user behavior so as to characterize better the traffic amounts and usage patterns, which enables us to perform more realistic tests. However, almost all load-testing products (e.g., *JMeter*) may produce false negatives due to caching and session management configurations. Software architects can also benefit from simulators like *Hoverfly* or *Vagrant* to evaluate different configurations without stopping or interfering the ecosystem's traffic. Running such simulations is necessary in order to expose the nonscalable modules of the ecosystem and prevent meltdowns associated with high amounts of user traffic in live usage. Finally, testing the resiliency of the microservices uncloaks the potential infrastructure failures. Netflix's open-source application *Chaos Monkey* is a good alternative to observe the destructive behavior of underlying resources. No matter which architecture is preferred for the application, the entire ecosystem should be orchestrated with a system like *Kubernetes*, *Mesosphere* or *Docker Swarm* along with an

infrastructure-level monitoring system. Scalability and performance are two different metrics, but they are intimately entwined. Therefore, monitoring the system will give us valuable and critical insights. Nevertheless, a favorable scalability and performance are not sufficient criteria for adopting a microservice architecture, unless we gain agility in the development team, and the deployment infrastructure can become fully automated.

CONCLUSIONS

Successful microservice implementations in enterprises, such as Netflix and Spotify, provide a motivation for other organizations to adopt this technology. However, the needs of the organization should be taken into consideration while choosing a microservice architecture. In general, there is no single microservice pattern that is better than the others. Rather, each design pattern performs better in different scenarios. Complex architectures come with long-term development cycles and additional license expenses for third-party applications. In addition, the employment of more qualified developer and test personnel in the team is another factor that increases the total cost. However, it should not be forgotten that these architectures increase productivity and drive down costs because they are energy efficient in the long term. The easiest way to evaluate the success of microservices is to ensure that they meet or exceed monolithic premigration performance. Microservice architectures are still immature, and therefore, best practices of their use are critical to their successful adaptation and incorporation in the future of SOA.

REFERENCES

1. O. Zimmermann, "Microservices tenets," *Comput. Sci.-Res. Develop.*, vol. 32, no. 3/4, pp. 301–310, 2017.
2. I. MuleSoft, "Whitepaper: The top six microservices patterns," Oct. 2018. [Online]. Available: https://www.mulesoft.com/lp/whitepaper/api/top-microservices-patterns
3. A. Gupta, "Microservice design patterns," Oct. 2015. [Online]. Available: http://blog.arungupta.me/microservice-design-patterns/
4. I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*. Newton, MA, USA: O'Reilly Media, 2016.
5. S. Newman, *Building Microservices: Designing Fine-Grained Systems*. Newton, MA, USA: O'Reilly Media, 2015.
6. C. Richardson, *Microservice patterns*, Shelter Island: Manning Publications, 2018.
7. H. E. Schaffer, S. F. Averitt, M. I. Hoit, A. Peeler, E. D. Sills, and M. A. Vouk, "NCSU's virtual computing lab: A cloud computing solution," *Computer*, vol. 42, no. 7, pp. 94–97, Jul. 2009.
8. V. F. Pacheco, *Microservice Patterns and Best Practices: Explore Patterns Like CQRS and Event Sourcing to Create Scalable, Maintainable, and Testable Microservices*. Birmingham, U.K.: Packt Publishing, 2018.
9. H. Khazaei, C. Barna, N. Beigi-Mohammadi, and M. Litoiu, "Efficiency analysis of provisioning microservices," in *Proc. IEEE Int. Conf. Cloud Comput. Technol. Sci.*, 2016, pp. 261–268.
10. L. Bao, C. Wu, X. Bu, N. Ren, and M. Shen, "Performance modeling and workflow scheduling of microservice-based applications in clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 9, pp. 2114–2129, Sep. 2019.
11. A. Akbulut and H. G. Perros, "Software versioning with microservices through the API gateway design pattern," in *Proc. IEEE, 9th Int. Conf. Adv. Comput. Inf. Technol.*, 2019, pp. 289–292.
12. "AWS fargate pricing," Sep. 2018. [Online]. Available: https://aws.amazon.com/fargate/pricing/
13. R. Shoup, "Craft conf 2015, day 3—From the monolith to microservices: Lessons from Google and eBay," 2015. [Online]. Available: https://codeandtalk.com/v/craft-2017/61479577
14. L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *Computer*, vol. 40, no. 12, pp. 33–37, 2007.
15. J. Leverich, M. Monchiero, V. Talwar, P. Ranganathan, and C. Kozyrakis, "Power management of datacenter workloads using per-core power gating," *IEEE Comput. Architecture Lett.*, vol. 8, no. 2, pp. 48–51, Feb. 2009.
16. X. J. Hong, H. S. Yang, and Y. H. Kim, "Performance analysis of restful API and RabbitMQ for microservice web application," in *Proc. IEEE, Int. Conf. Inf. Commun. Technol. Convergence*, 2018, pp. 257–259.

Akhan Akbulut is currently a Postdoctoral Researcher with the Computer Science Department, North Carolina State University. His research interests focus on the design and performance optimization of software-intensive systems, Internet architectures, and broadening participation in cloud computing research. He received the P.h.D degree in computer engineering from Istanbul University, Istanbul, Turkey. He is a Member of the IEEE. Contact him at aakbulu@ncsu.edu.

Harry G. Perros is currently a Professor of Computer Science, an Alumni Distinguished Graduate Professor, and the co-founder and Program Coordinator of the Master of Science degree in Computer Networks program at North Carolina State University. He has published extensively in the area of performance modeling of computer and communication systems, and has organized several national and international conferences. He has also published six print books and an e-book. His current research interests are in the areas of resource allocation under QoS, IoT analytics, and queueing theory. He is a Fellow of the IEEE. Contact him at hp@ncsu.edu.