A Performance-Adaptive and Time-Monitored Autonomous Ticket Booking Service in Cloud

Liu, H.; Oudejans, M.; Xin, R.; Grosso, P.; Zhao, Z.

DOI
10.1109/ISIE51358.2023.10228152

Publication date
2023

Document Version
Final published version

Published in
2023 IEEE 32nd International Symposium on Industrial Electronics (ISIE)

Citation for published version (APA):
Liu, H., Oudejans, M., Xin, R., Grosso, P., & Zhao, Z. (2023). A Performance-Adaptive and Time-Monitored Autonomous Ticket Booking Service in Cloud. In *2023 IEEE 32nd International Symposium on Industrial Electronics (ISIE): 19-21 June, 2023, Helsinki-Espoo, Finland : proceedings* (pp. 940-945). IEEE. https://doi.org/10.1109/ISIE51358.2023.10228152

A Performance-Adaptive and Time-Monitored Autonomous Ticket Booking Service in Cloud

Hongyun Liu*‡
Maarten Oudejans§ Ruyue Xin* Paola Grosso* Zhiming Zhao*†
*Multiscale Networked Systems (MNS), University of Amsterdam, Amsterdam, the Netherlands
†LifeWatch ERIC Virtual Lab and Innovation Center, Amsterdam, the Nederlands
‡Graduate School of Informatics, University of Amsterdam, Amsterdam, the Nederlands
§University of Amsterdam, Amsterdam, the Nederlands
Email: {h.liu|r.xin|p.grosso|z.zhao}@uva.nl, maarten.oudejans@gmail.com

2023 IEEE 32nd International Symposium on Industrial Electronics (ISIE) | 979-8-3503-9971-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ISIE51358.2023.10228152

***Abstract*—With the increasing demand for autonoumous ticket booking services for all kinds of events, user numbers and service performance requirements increase dramatically. Platforms such as BASH, Ticketmaster, Eventbrite, and SeatGeek, often outsource such services to third parties, but the operation cost is often high. Therefore the performance for time-critical requirements is without monitoring. Furthermore, the cloud platform is becoming a popular solution for event opganizing platforms due to its mighty computation power. In order to address these issues, this study presents a cloud-based, performance-adaptive, and time-monitored autonomous ticket booking system, utilizing a paradigm for booking as a service that can be readily included into cloud platforms. Together with the implementation itself, we also provide a variety of flexible service settings that may serve a specified number of concurrent users while consuming the least amount of resources. We carry out thorough implementations to verify our methodology.**

***Index Terms*—time-monitored, cloud platform, autonomous ticket booking, performance-adaptive**

I. Introduction

With the popularity and convenience of the online ticket booking service [7], there have been increasing numbers of platforms which are in need of adaptive autonomous ticket booking solutions. Third-party providers like Ticketmaster, Eventbrite, and SeatGeek are pricey proprietary solutions. These services hold client data while they are operating, jeopardizing privacy and data control. Because a platform depends on those services to provide economic value, any potential failure of those third-party services might have a direct impact on the user experience of its users. The consequent performance degradation harms time-critical requests, and the corresponding solution can be costly. Using an unlimited percentage-based pricing system is a frequent approach; nevertheless, this has substantial costs [13] [9] [2]. Moreover, because all of these services are provided by other companies, users are transferred from the platform to another setting to purchase tickets. Consequently, ticket booking as a time-critical service experiences performance degradation. To that end, this effort attempts to build a solution that can immediately adapt to the current services of a platform, allowing the platform total control, and provide an autonomous Booking-as-a-Service system for platforms that handle these issues. Furthermore, this work aims to support a large concurrent user number and solve these problems. The best practice nowadays for handling high user volumes is to dynamically scale the service's resources in accordance with the data on the actual workload. Regarding the demands of high computational power and scalability, a cloud-based platform becomes an ideal solution. In this work, we investigate the ideal integration and adaptive configuration for a cloud-based ticket booking platform and conduct a comprehensive analysis.

A. Challenges

There are certain challenges for an autonomous ticket booking platform: 1). Large user number; 2). The conflicts control between tickets availability and purchasing behavior; 3). Faulty purchases; 3). Service availability: for tickets booking service, the service up-time needs to reach 99%.

B. Contributions

The contribution of this work is as follows:

1) We propose an autonomous ticket booking service system that can be directly integrated with a ticket selling platform.
2) We propose a performance-adaptive ticket booking system which offers ticket booking service to numerous users, meeting the time-critical requirement in the meantime.
3) We conduct comprehensive implementation and analysis of the adaptive configurations based on our proposed system. The final solution and configurations are validated by BASH[1].

II. Related work

There have been a lot of work done in terms of ticket booking services [10] [3] [8] [11], some work addressing time-critical tasks [6] [5]. The book [10] helps us get a deeper comprehension of the tools and resources that enable the development of scalable and time-critical applications. These studies look into how to assess the cost and quality of a ticket-booking service. [3] analyzes the effectiveness and prices of various cloud services offered by well-known cloud

[1]BASH is a tech startup building a platform where people can easily organize and attend events.

companies. The same methodology may now choose a cloud provider and the appropriate services based on the desired performance and cost balance. Sidra Aslam et al. [1] [4] provide further insight into the role of run-time load balancing in distributed, quality service-dependent systems. The current state of availability in cloud technology is investigated and summarized by Mina Nabi in their article [8]. The article explores the popular mechanisms used by cloud providers for error recovery and evaluates their performance in ensuring availability. To predict the workload of a cloud application and dynamically scale resources as needed, Nilabja Roy et al. [11] have developed a system. In terms of load balancing mechanisms, Kubernetes is supported by several cloud providers, including Google Cloud Platform [12]. Kubernetes is a container orchestration system that implements both load balancing and error recovery.

After reviewing the aforementioned works, it becomes clear that there is a significant focus on performance adaptability, including load balancing, scalability, and assessing the quality of service and booking costs. However, the study [3] that evaluates the cost-to-performance ratio of specific cloud services is outdated and doesn't consider newer technologies like cloud functions or even Kubernetes. This work aims to build up a performance-adaptive autonomous booking system and implement comprehensive implementation to demonstrate its performance.

III. Architecture

A. Requirements

1) Functional level: The booking service must offer the expected functionality. At the very least, it should have HTTP endpoints that enable users to: 1) purchase tickets, 2) view purchased tickets, 3) create and sell tickets, and 4) check the number of available tickets.

2) Performance level: From the end-user's perspective, the following properties are crucial: **Minimal latency**: High latency can negatively impact user experience and lead to inaccurate purchases. **Service availability**: The service must be accessible at all times, with minimal downtime. **Consistency**: Customer purchase behavior must align with ticket availability, which should be continuously updated. **Distributed**: To provide low-latency access to users worldwide, multiple instances of the service should run concurrently across different continents. **Scalable/elastic**: The instances should be capable of horizontal and vertical scaling to handle increased concurrent user traffic. **High fault tolerance**: The service should remain operational in the event of errors or failures, ensuring minimal downtime.

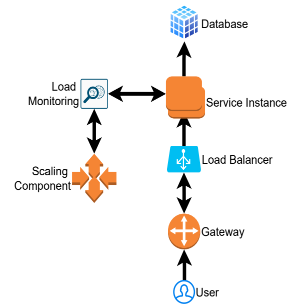

Fig. 2. An instance of the service architecture created by the customization component as an illustration.

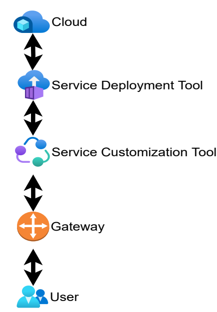

Fig. 1. Architecture of service customization component

IV. System design

The architecture for our service customization component is presented in Figure 1. By indicating the required number of concurrent users the service should handle, the user communicates with the component. With this data, the Service Customization Tool creates a deployment description that supports the desired user count while minimizing expenses. The service deployment tool receives the created description from the tool and distributes the service automatically in accordance with the description.

Figure 2 illustrates an architecture example of the service developed by the customisation component to meet the specifications listed earlier. The service starts at the load balancer, which the user may access by using a front-end to make HTTP requests to our service. The load balancer then splits up the requests amongst the service instances, and the load monitoring component keeps track of the workload and relays that information to the scaling component. In accordance with pre-established configuration limitations and taking into account the workload of the instances, the scaling component modifies the number of service instances. The service will be able to scale to fulfill the demands for availability and low latency. Last but not least, the service instances establish a connection to our database, which stores and retrieves data on tickets, users, and orders. Figure 3 illustrates the information flow within the system, using the example of a user buying a ticket.

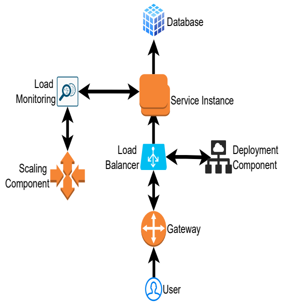

Fig. 3. Sequence diagram of the service

V. System implementation

A. System Structure

It is the purpose of the aforementioned architecture to be provider- and technology-neutral. Nevertheless, in this study, we have utilized two different services provided by Google Cloud Platform, namely Cloud Functions and Kubernetes, to create the suggested architecture shown in Figure ref-fig:architecture diagram. We may concentrate on developing our service while taking use of the scalability provided by these platforms by employing the existent load balancing, monitoring, scaling, and instance management features of these services.

In terms of the database element, we have decided to utilize a Postgres instance hosted by Google Cloud SQL. In order to prevent the incidence of incorrect ticket purchases, this decision is based on the database's ACID compliance. Also, the chosen database has the Multiversion Concurrency Control (MVCC) mechanism set up, allowing read operations to proceed without obstructing write activities. Given that many people will be accessing the accessible ticket information while others are simultaneously amending it, this capability is important for our use case.

B. Service

We have created two independent versions of our service due to the unique programming needs of cloud functionalities. The first was designed with containerization in mind, and it contains a core function that accepts requests for all conceivable functions and starts new threads for each one that comes in. The second version, however, lacks a fundamental feature because all request processing responsibilities must be handled by the cloud service provider.

1) Containerization: The containerized version of our service has been designed for usage with common services like Kubernetes and includes a core function that performs a certain routine for each incoming request. These processes communicate with the database and produce an appropriate answer. We have implemented a connection pooling method that allows for up to 10 concurrent connections per instance in order to reduce the performance limits that usually arise from the database's connection management in such services.

Our containerized approach uses Gin and GORM, two libraries, to deliver the required functionality. While GORM is our preferred Object-Relational Mapping (ORM) package, Gin is used to manage HTTP requests and URL matching.

2) Cloud functions: In contrast to the containerized version, the cloud function implementation of the service is not self-contained. Unlike the Kubernetes-based method, services like Google Cloud Functions simply need customers to deliver the code for a single function that complies with the cloud functions service's HTTP framework criteria. In our particular situation, Google Cloud Functions anticipates a Go function with the arguments "http.ResponseWriter, http.Request" since it makes use of Go's built-in HTTP framework.base.

While function instances can be reused, a different init function is used each time a new instance is generated to establish the database connection. It is not feasible to include a connection pooling mechanism within the functions themselves due to the total isolation of cloud function instances. A other service, such PgBouncer, must be hosted and linked to the database in order to solve this problem. Then, rather than directly connecting to the database, cloud functions would link to the PgBouncer instance.

C. Functionality

Both implementations must provide the same functionality and make use of the same logic in order for tests conducted between them to be reliable. The following HTTP endpoints are exposed by both implementations:

- **/tickets (GET, POST, DELETE)**: Find, create or delete tickets. You can also retrieve a single ticket using the 'id' query parameter.
- **/tickets-buy (POST)**: Buy a ticket and decrease the available number of that ticket. This returns an *order* object which can later be retrieved again by the user in a different call.
- **/users (GET, POST)**: Find or create a user/users.
- **/users-tickets (GET)**: Find the tickets bought by a specific user, using the 'id' query parameter to select a user.

The vast majority of these functions use simple SQL queries to communicate with the database and produce desired results. Contrarily, the **tickets-buy** endpoint is considerably more complicated and may be used to introduce erroneous purchases or racial circumstances. By assuring the availability of sufficient tickets and changing the number of available tickets inside the same Postgres transaction, we try to allay both of these worries. This method ensures that the ticket count remains constant while we validate it and alter it appropriately because Postgres transactions are atomic. We then create a new order and provide it to the ticket buyer if there are still enough tickets available and the count has been effectively dropped.

VI. Experiments

We carry out a number of trials with various setups of our booking service in an effort to strike the ideal balance between performance and cost. We can ascertain the precise effect of each component on the overall performance and cost of the service by methodically adjusting the configuration of individual components. We then integrate the outcomes of

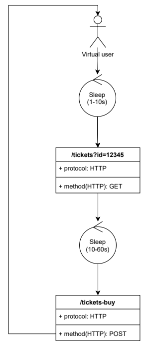

Fig. 4. Lifecycle of a virtual user

these tests to produce a variety of configurations that satisfy our specifications. These configurations range in price and performance, with some providing better performance at a higher price and others doing so at a lesser price. This gives the platform utilizing the service freedom by letting them select a configuration that suits their needs, preventing excessive spending on inefficient resources.

VII. Methodology

We run a workload simulation that simulates concurrent customers submitting requests in an identical way to actual users in order to assess the performance of the service for each configuration. These tests entail exponentially raising the number of virtual users in order to determine the service's failure threshold. When 99% of requests' maximum request time reaches 2 seconds, the test is considered to be complete. In order to emulate users, we employ the k6 framework, as illustrated in Figure 4. Each user initially waits for a random period between 1 and 10 seconds, and then queries the remaining number of tickets for a specific ticket using an HTTP GET request to **/tickets?id=ticket-id**. By waiting for a random duration, we ensure that users do not send requests simultaneously. After an additional waiting period of 10 to 60 seconds, the user purchases a ticket by sending an HTTP POST request to **/tickets-buy**. Once a ticket is purchased, the user restarts and acts as a new user, repeating the aforementioned sequence of actions until the completion of the test.

A. *Metrics*

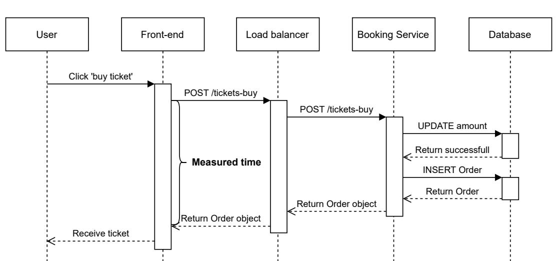

Fig. 5. Location of time measurements

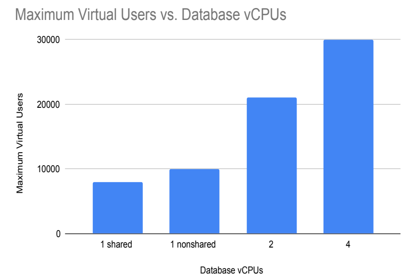

Fig. 6. Database performance vs assigned vCPUs

We keep track of each HTTP request's duration in order to assess how well our service is functioning during these testing. We stop the test when the 99th percentile of these times hits 2 seconds, and we keep track of the most virtual users who were reached up to that point.

We gather these metrics at the front-end level, where we time how long it takes from the time a request is sent to our service to the time it is returned, as shown in Figure 5.

B. *Testing environment*

We carried out the testing on a Google Compute Instance to make sure that our trials could be repeated and that they were consistent. We specifically used the 'e2-standard-8' machine type, which has 32 GB of RAM and 8 virtual central processing units (vCPUs).

VIII. Experiment 1: Database performance

A. *Methodology*

We progressively increase the number of simulated users until the 99th percentile of HTTP requests sent by these users reach a threshold of 2 seconds in order to find the best configuration for our system. Iteratively increasing the number of vCPUs assigned to the database with each iteration of this operation. The impact of introducing a read replica to the system is examined in a second experiment once the ideal number of vCPUs has been established. The primary instance of the database is offloaded from some read operations by a read replica, which runs as a separate instance of the database and duplicates each update from it.

B. *Results*

The visualization depicted in figure 6 reveals that enabling access to more CPU cores leads to a substantial increase in the maximum count of simultaneous supported users. Furthermore, introducing a read replica yielded a performance improvement within expected bounds, as demonstrated in figure 7. Notably, during the experiments, approximately 75% of the requests issued by the simulated users were merely of a read nature.

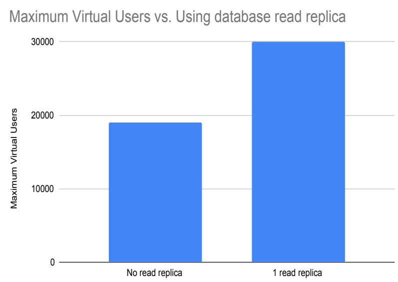

Fig. 7. Database performance with or without a read replica

IX. Experiment 2: Service performance when using Kubernetes

A. Methodology

Until the 99th percentile of the HTTP request length hits a specified threshold of 2 seconds, we gradually increase the number of virtual users in our workload to establish the service's maximum capacity. We do several tests on a handful of chosen machine types to assess the effect of various machine types on the service capacity. We conduct the experiment many times with progressively more nodes running our service on each type of computer.

B. Results

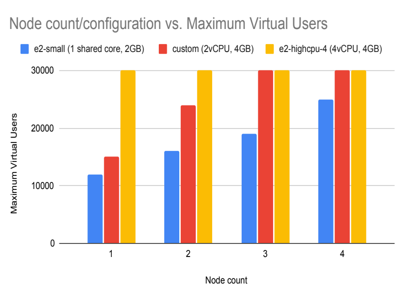

Fig. 8. Maximum supported users for different Kubernetes configurations

Several settings can be used to attain certain maximum supported concurrent user counts, as illustrated in figure 8. Each node type's performance improves as the number of nodes rises, as predicted. The fact that the 'e2-highcpu-4' node type could never support more than 30000 virtual users on this graph further demonstrates the networking limitations of our testing gear. Both the "custom" and "e2-highcpu-4" node types could probably accommodate more than 30,000 virtual users, although better hardware would be needed to confirm this assertion.

X. Experiment 3: Service performance when using Cloud Functions

A. Methodology

The allotted RAM and the maximum number of instances for each function are changed for our tests using the Cloud Functions implementation.

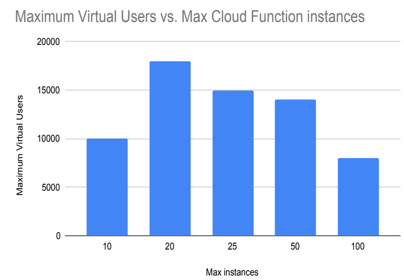

Fig. 9. Maximum supported users vs. limiting the maximum number of instances per Cloud Function

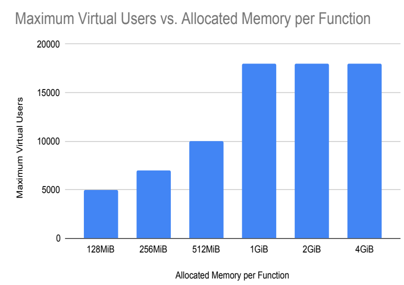

Fig. 10. Maximum supported users vs. allocated memory per Cloud Function

B. Results

Figure 9 illustrates how restricting the number of concurrent instances of each cloud function increases the number of concurrent users that our service can accommodate. The greatest results are obtained when the maximum number of instances per function is 20. If we raise the maximum number of open connections, performance suffers similarly, albeit not to the same extent as with the Cloud Functions implementation, as demonstrated in figure 11. The findings in figure 10 demonstrate when memory allocation for individual cloud functions begins to impede the system's overall throughput. The bottleneck is removed by allocating 1 GiB or more of RAM for each function. Any additional gigabytes appear to have little advantage.

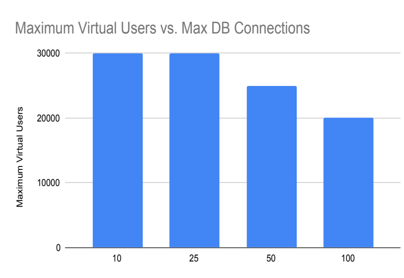

Fig. 11. Maximum supported users vs. maximum open database connections using Kubernetes

XI. Experiment 4: Service performance characteristics of different Configurations

A. Methodology

Lastly, we suggest a variety of setups that enable growing numbers of concurrent users utilizing the findings from the tests previously presented. We try to minimize the anticipated expenses for each of these setups. We calculate these expenses using Googles pricing calculator

B. Results

Configuration	Database vCPUs	Read replica	Cloud service	Cloud Functions Memory	Kubernetes Node Type	Kubernetes Node Count
A	1	No	Cloud Functions	256MiB	N/A	N/A
B	2	No	Cloud Functions	1GiB	N/A	N/A
C	2	No	Cloud Functions	1GiB	N/A	N/A
D	2	Yes	Kubernetes	N/A	2vCPU, 4GB	1-2
E	2	Yes	Kubernetes	N/A	4vCPU, 4GB	1
F	4	Yes	Kubernetes	N/A	4vCPU, 4GB	1

TABLE I
PROPOSED CONFIGURATIONS OF THE BOOKING SERVICE

Configuration	Maximum concurrent users	Cost/Month
A	5000	€28.33 - €309.87
B	10000	€72.55 - €721.71
C	15000	€72.55 - €1049.90
D	20000	€187.16 - €225.94
E	20000	€213.53
F	30000	€312.43

TABLE II
SUPPORTED CONCURRENT USERS AND ESTIMATED MONTHLY COST OF PROPOSED CONFIGURATIONS

The six configurations we chose are displayed in Table I. These configurations are in accordance with table II, which lists the maximum number of concurrent users for each setup along with the associated monthly cost. For the amount of people it serves, each configuration attempts to be as economical as feasible. All four of the configurations—A, B, C, and D—utilize Cloud Functions. Cloud Functions have a zero scaling capability. Two configurations, D and E, are available to accommodate a maximum of 20000 concurrent users. The 20000 users are supported by Configuration D, which employs a less efficient node type that scales up to two instances. Configuration E is a preferable choice since it never scales down if a platform employing this service needs continual availability for up to 20,000 users. Option D works better if the platform hardly ever uses all of its capacity. Configuration F offers the best value for 30000 concurrent users.

XII. Conclusion

In this work, we suggest and put into practice a cost-effective autonomous, adaptive booking service. A minimum of 30000 concurrent users have been tested to operate flawlessly under the installed service. It offers the fundamental features one would anticipate from a booking service, enabling users to purchase, resell, and get event tickets. We have given a variety of adaptive options for this service that reduce the cost for a specific number of concurrent supported customers. These configurations can be used by bigger current platforms as well as tiny, budget-constrained platforms. This service was created with cloud independence in mind. Each of the three major cloud service providers—Google Cloud, Amazon Web Services, and Microsoft Azure—offers the same set of services, including the database, cloud functions, and Kubernetes engine. As a result, other cloud providers can also use the options we've offered.

Acknowledgment

This research is funded by China Scholarship Council and the European Union's Horizon 2020 research and innovation program under grant agreements 825134 (ARTICONF project), 862409(BlueCloud project) and 824068 (ENVRI-FAIR project). The research is also supported by EU Life-Watch ERIC.

References

[1] Aslam, S., Shah, M.A.: Load balancing algorithms in cloud computing: A survey of modern techniques. In: 2015 National software engineering conference (NSEC). pp. 30–35. IEEE (2015)
[2] Bachis, E., Piga, C.A.: Low-cost airlines and online price dispersion. International Journal of Industrial Organization **29**(6), 655–667 (2011)
[3] Brebner, P., Liu, A.: Performance and cost assessment of cloud services. In: Service-Oriented Computing, Lecture Notes in Computer Science, vol. 6568, pp. 39–50. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
[4] Liu, H., Chen, P., Ouyang, X., Gao, H., Yan, B., Grosso, P., Zhao, Z.: Robustness challenges in reinforcement learning based time-critical cloud resource scheduling: A meta-learning based solution. Future Generation Computer Systems (2023)
[5] Liu, H., Chen, P., Zhao, Z.: Towards a robust meta-reinforcement learning-based scheduling framework for time critical tasks in cloud environments. In: 2021 IEEE 14th International Conference on Cloud Computing (CLOUD). pp. 637–647. IEEE (2021)
[6] Liu, H., Xin, R., Chen, P., Zhao, Z.: Multi-objective robust workflow offloading in edge-to-cloud continuum. In: 2022 IEEE 15th International Conference on Cloud Computing (CLOUD). pp. 469–478. IEEE (2022)
[7] Molchanova, V.S.: The use of online service booking in tourist activity. European Journal of Social and Human Sciences (2), 75–80 (2014)
[8] Nabi, M., Toeroe, M., Khendek, F.: Availability in the cloud: State of the art. Journal of network and computer applications **60**, 54–67 (2016)
[9] ÖZBEK, A.P.V., GÜNALAN, L.M., KOÇ, A.P.F., Şahin, N., Eda, K.: The effects of perceived risk and cost on technology acceptance: A study on tourists' use of online booking. Manisa Celal Bayar Üniversitesi Sosyal Bilimler Dergisi **13**(2), 227–244 (2015)
[10] Reese, G.: Cloud Application Architectures: Building Applications and Infrastructure in the Cloud. Theory in Practice (O'Reilly), O'Reilly Media (2009), https://books.google.nl/books?id=j8YO7gVqMqAC
[11] Roy, N., Dubey, A., Gokhale, A.: Efficient autoscaling in the cloud using predictive models for workload forecasting. In: 2011 IEEE 4th International Conference on Cloud Computing. pp. 500–507. IEEE (2011)
[12] Sayfan, G.: Mastering Kubernetes. Packt Publishing (2017), https://books.google.nl/books?id=dnc5DwAAQBAJ
[13] Zhang, M., Zhang, C., Sun, Q., Cai, Q., Yang, H., Zhang, Y.: Questionnaire survey about use of an online appointment booking system in one large tertiary public hospital outpatient service center in china. BMC medical informatics and decision making **14**(1), 1–11 (2014)