Cloud-Based Online Concert Ticketing System

Zhong Xi Lu
University of Antwerp
Antwerp, Belgium
zhong-xi.lu@student.uantwerpen.be

***Abstract*—Tomorrowland, being a large music festival, usually has a pre-sale where customers can order a ticket online. However, depending on the amount of customers, the system that handles all the order requests will receive an enormous amount of load. To deal with this, the system could be deployed on the cloud where it could scale elastically while still managing this peak load. This means that customers will not have to wait hours to get a response from the system.**

I. Introduction

Over the last years, some of the customers of Tomorrowland [1], have been frustrated by the fact that they have to wait hours in queue to get a ticket ordered online [1]. To achieve a better customer satisfaction, the internal system that deals with order requests, needs to be adjusted to deal with this peak load. A solution to this is cloud computing where resources can allocated dynamically [2]. This cloud setup works perfectly in the context of handling a large amount of requests since depending on the current load, the system will scale itself automatically so the requests can be processed in a reasonable amount of time.

To confirm this, two steps need to be taken first, so we can evaluate whether this cloud solution will result in a better performance, i.e. a faster response time overall. In this context, the goal is to achieve a response time of ten minutes maximum for any request during any period of time. The initial step is to first model the system in ABS[2]. In this language, it is possible to explicitly model resources, such as the time cost of an operation. This is ideal since we want to observe how fast a request takes. After the system has been completely modelled, simulations can be created to observe the response times of all the requests. This will then tell us whether it is worth to invest more in this cloud solution, i.e. actually implement the system.

If the simulations have proven that the cloud setup is capable of dealing with this peak load, we can start by implementing a first version of the system or so-called proof-of-concept. With this proof-of-concept a more precise conclusion can be drawn. More precisely, by implementing the system, requests can be send and actually be timed to see how the system performs under heavy load. In other words, we can perform load testing on the system to see how the system deals with a big load of requests.

The paper is organized as follows: firstly, in Section II, the system will be described, i.e. its architecture and how the system can be deployed on the cloud. Next, Section III will go over the simulations and Section IV over the proof-of-concept. Future work is also needed to validate some of the assumptions we made and how we can move from the proof-of-concept to an actual implementation, this is elaborated in Section V. Finally, in Section VI, a conclusion is given.

[1]https://www.tomorrowland.com/en/festival/welcome
[2]https://abs-models.org/

II. Architecture

A. Microservices

In order to fully make use of the cloud, we must come up with a cloud-based architecture. More so, the system will have a microservice-oriented architecture [3]. Microservices have many advantages; one of these is the fact that it is possible to scale individual components, such as a service. This is useful in this context since we can more precisely scale the components that have more load than others.

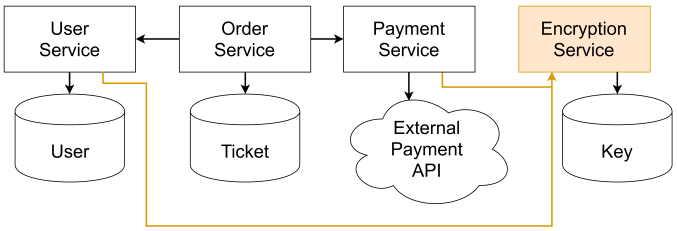

Fig. 1: General Overview of Microservices

In Figure 1, a general overview is given of all the microservices and all the different dependencies in the concert ticketing system. Each one of these services is developed as a standalone application and can be deployed independently of any other service. The way these services communicate with each other is through a precisely defined REST API, this makes it easy for services to send requests and responses to one another without worrying too much about their internal representation. A more detailed explanation of all these services is described below.

1) User Service: This component deals with all the user-related logic, such as user registration, updating user information, verifying user credentials as well as retrieving users. The users themselves are stored in the user database.

2) Order Service: The order service is responsible for all the incoming order requests and makes sure to call the other services to achieve its goal, namely creating a new ticket for a user or cancelling an order. These tickets are stored in the ticket database; on top of this, the amount of tickets that are available, is also stored in this database.

3) Payment Service: To order a ticket, a user also needs to register their credit card information beforehand so we can create a transaction on the right credit card account. To do so, the system relies on an external payment API. Credit card services usually have a dedicated API which developers can integrate in their own system. For instance, in our system, a user can register a MasterCard credit card, so in order to charge an amount on this card, the MasterCard API[3] needs to contacted. The payment service sort of acts as a middleman between our internal system and the credit card API.

However, in both the simulation and the proof-of-concept, these requests to the external payment API are mocked since it was not possible to use the actual API and charge actual amounts on credit cards and on top of this, the API sandbox does not allow for load testing, which is needed in our case.

4) Encryption Service: As mentioned in the previous section, the credit card information of a user is also stored in the database, but in the case that the database gets hacked and the information becomes accessible to potential hackers, we need to make sure that this information is encrypted. In our case, AES is used to encrypt all the necessary credit card information [4]. The key and initialization vector (IV) is stored in a key database or the "secure vault". To retrieve the key and IV for a specific user, a request needs to be send to the encryption service. This service is also responsible for generating a personal key and IV for a user.

B. Deployment Diagram

The deployment diagram of the system is shown in Figure 10 of Appendix A. Since our architecture is microservice-oriented, all of the services and databases are deployed on their own server. These can then interact with each other through their REST API which is done over HTTPS.

C. Service API's and Database Schemas

The precise API of all the services and the database schemas are defined in Figure 11 of Appendix A. Note that these are only partial API's of the services; only the requests that are needed to do the load testing are considered for the remaining part of the paper. Most of the API's and schemas are already implicitly described in Section II-A about the microservices. The yellow part in the user database schema represents the credit card information and as mentioned earlier, this is encrypted using AES. For the ticket schema, multiple database instances are represent since database sharding is applied (see Section III-C1). Hence, the i indicates the i-th shard. All the other parts of the API's and database schemas speak for themselves and are not further discussed in this section.

D. Sequence Diagram

The load testing that will be done in Section III and IV revolves around ordering tickets. Hence, in this section, a thorough explanation is given about the flow of ordering a ticket. The exact sequence diagram is shown in Figure 12 of Appendix B.

The first request in the sequence is sending a request to the order service, this request also contains information of the user, including the user id and token. The order service will first verify the user, i.e. verifying whether the token is legitimate and valid for that specific user. To do so, it sends a request to the user service which will do all sorts of checks and finally send a response message back. The following step of ordering a ticket is to find a database (shard) that has at least one ticket left. However, since multiple databases are present, it is possible that we need to iterate over all these shards to find one that has tickets left. In worst case, this means that if there are no tickets available in the system, all the shards need to be iterated through, only to send a response back that the tickets are sold out.

If a shard with tickets left is found however, the actual payment can be issued. This is done by the payment service. This service will receive encrypted credit card information. For this, an intermediate request is sent to the encryption service to retrieve the key and IV for a user. With this, the decrypted credit card information can be forwarded to an external payment API. As mentioned earlier, the actual call to the external payment API is mocked in our case, so no calls are actually made. Finally, if everything went successful until this point, a new ticket is created and sent back to the user.

III. Simulation

A. Modelling

Before implementing the proof-of-concept, we need to make sure that it is worthwhile to invest more in this system and architecture. This is done by modelling this system in ABS and running simulations. The components that are modelled are all the services and databases as well as the network itself. By doing so, all the different interactions and operations can be explicitly modelled in time. Next to this, the scaling algorithm also needs to be modelled, this is based on the horizontal pod autoscaler of Kubernetes[4] (HPA). The HPA can create and delete new instances, called replicas, of a specific service depending on the load of that service.

The modelling is done at the lowest possible abstraction level. For instance, calculations needed to encrypt and decrypt, database queries, transit times of requests, ..., i.e. all the steps in the sequence diagram in Figure 12 are taken into account when modelling the system. This means that we can get a more accurate representation of the flow of the requirements. The communication between these services and databases is modelled using the broker pattern where the broker represents the network. The user can then send a request to this network which will then forward it to the corresponding service.

[3]https://developer.mastercard.com/apis

[4]https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#algorithm-details

B. Calibrating Model

After completely modelling everything, we need to configure all the different time costs of each operation and calibrate our model to get a representative model of the system. However, most of these costs are not explicitly documented somewhere, which means that it becomes hard to find concrete timings. Hence, some of the final costs are measured by hand or by a small script that times a certain operation. This way, it is still possible to get at least a meaningful cost.

Another issue is the processing time of the external payment API. It was expected that these services have a service-level agreement in which they state how fast a request is processed, but none of them actually gave information about the performance. For instance, MasterCard does not make any commitment to the developer about its performance[5]. Though, it is possible to find some timing costs of handling payments through other sources[6]; as an average processing time, 1.4 seconds is taken. This is a big assumption and needs to be taken into account when evaluating the final results.

Another noteworthy assumption is that during the simulations it is assumed that the requests to the services, except the order service, all have a constant response time. This assumption, however, can be backed up by the fact that these requests are not made up of complex calculations, most of them are simple retrievals. Therefore, by scaling them up, they can achieve a constant response time. Finally, it is expected that there are 350.000 tickets[7] available at the launch of the system. The remaining part of the paper will revolve around this amount.

C. Simulation Results

1) Database Sharding: Given the model of the system and the cost model, it is now possible to run simulations. A first simulation when ordering 100 tickets at the same time already resulted in more than two minutes for the longest response time. Clearly, there was a big bottleneck somewhere and after further inspection, it was shown that the database was this bottleneck. In order to assure that the system does not sell more tickets than there are available, access to the ticket count in the database is done sequentially and then locked during the remaining part of the transaction. This leads to that requests are actually processed sequentially.

A solution to this is database sharding [5]; this one database of tickets is split up into multiple smaller databases, called shards. The reason to do this is so that multiple databases can be accessed concurrently and this also means that multiple locks are now represent. After modelling the sharding as well, a significant sped-up could be noticed: rerunning the same simulation from before (ordering 100 tickets) with 350 shards roughly sped up the longest response time by ten times as can be seen in Table I. To determine the number of shards was not possible through simulations due to the unforeseen nature of ABS. In the remaining part of the simulation, 350 shards were chosen out of 350.000 tickets.

	w/o Sharding	w/ Sharding
Shortest Response Time	1613.5ms	1613.5ms
Average Response Time	73685ms	7542ms
Longest Response Time	145757.5ms	14789.5ms

TABLE I: Results for database sharding with 350 shards when ordering 100 tickets at the same time

2) Bottlenecks: Continuing with this model, potential other bottlenecks could be identified. The first one in mind is the encryption or rather decryption when creating a new payment. This, however, did not produce significant results (see Table II), given a constant retrieval time of the key.

	w/ Encryption	w/o Encryption
Shortest Response Time	1613.5ms	1605ms
Average Response Time	7542ms	7500ms
Longest Response Time	14789.5ms	14704ms

TABLE II: Results for encryption when ordering 100 tickets at the same time

Another potential bottleneck is the dependency on the external payment API. To assess this, the cost model is configured with different processing times for this API. The final result can be found in Figure 2. As can be seen, there is a noticeable bottleneck on the credit card API; more specifically, the longest response time of an order grows linearly to the response time of this API. For instance, doubling the response time of the external API also means doubling the longest response time of ordering a ticket.

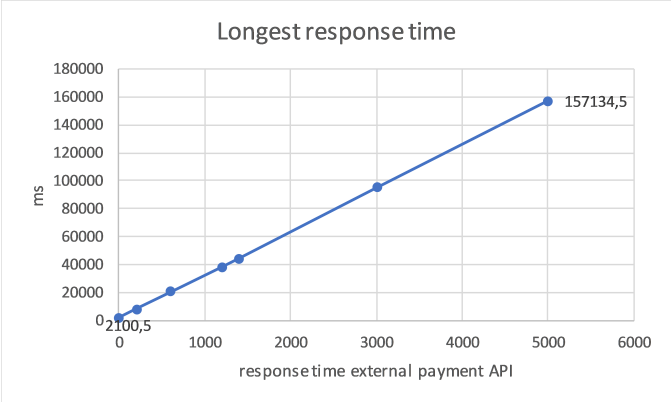

Fig. 2: Response time for different external API processing times

3) Servers Needed: When deploying the system, one would also like to know how many servers are needed to setup the whole architecture. This way, an estimate of the cost is known beforehand. In this case, the amount of servers is related to the amount of replicas that are created by the HPA. From the simulation, it was predicted that 35.000 replicas were needed when 350.000 tickets were being ordered at the same time

[5]https://developer.mastercard.com/page/developers-evaluation-agreement#5-availability-and-support

[6]https://data.spreedly.com/

[7]https://thegroovecartel.com/news/tomorrowland-2019-sells-out-how-get-tickets/

(see Figure 3). Though, in reality, it is not needed to have this amount available. Since in this setup, 350 shards are available, only 350 replicas are needed at the minimum as well, but we assume that it is better to make more available, because the shard that is contacted to check if there are any tickets left, is randomly selected. Hence, it is very likely that some shards are not being occupied. Important to note is that one replica does not necessarily mean one server. More replicas can perfectly be ran on a single server. In any case, it is hard to give a precise number, but it is expected that around 1000 servers will suffice to run all these replicas. Given that 35.000 replicas are needed for 350.000 tickets, each server can run 7 replicas (see Section IV-B3) and a single replica can process 5 requests at the same time[8], we get 1000 servers as an end result (see Equation 1). Though, these are all assumptions and more may be needed in the case that the peak load is more than 350.000 requests at the same time, but it still gives us a rough estimate.

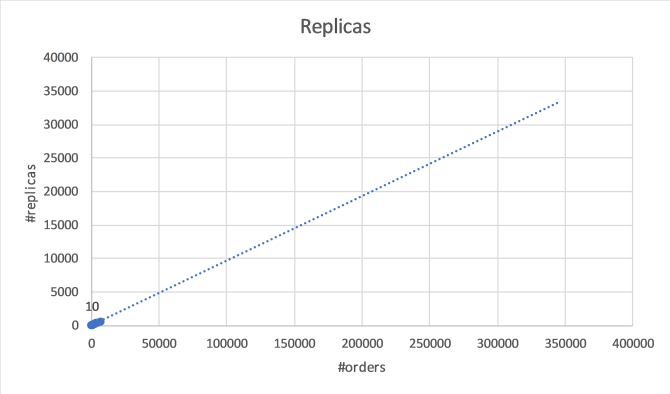

Fig. 3: Prediction for the amount of replicas

$$\frac{35.000\ replicas}{7\ replicas/server \cdot 5\ requests/replica} = 1000\ servers \tag{1}$$

4) Longest Response Time: To see whether the system can respond in less than 10 minutes during peak load, a prediction model was needed as can be seen in Figure 4. It was predicted that the longest response time is linear to the number of orders at the same time. If all the tickets (350.000) were ordered at the same time, the longest response time would be roughly 31 minutes. This was not our goal; if we check how many tickets we can order while still maintaining our goal, we can order 106.300 tickets. Note that these order requests all contain valid information (valid user and token, enough tickets left and sufficient balance on credit card), meaning that all the orders are successfully being handled and finished. This way, the longest path in the sequence diagram in Figure 12 could be traversed resulting in the worst case, i.e. the longest possible time a request could be in our system. As a conclusion, if the peak load is no more than the previously mentioned amount and if our cost model was correct, then the system should be able to respond in a reasonable amount of time, i.e. within 10 minutes.

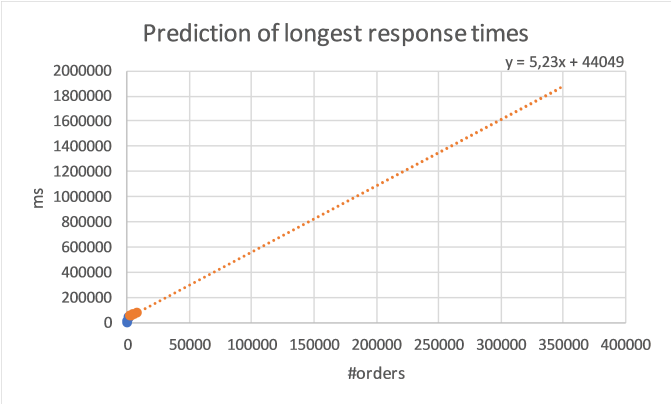

Fig. 4: Prediction for the longest response time for different amount of orders

IV. Proof of Concept

The simulations have shown that it was worthwhile to invest more in this architecture. Not only that, it also gave us insights to our initial design. For instance, database sharding is adapted in our proof-of-concept since it has been proven to be a faster solution. The simulations have also shown that when a new pod is started, it has a slight startup delay, so it is best to scale immediately to a considerable high amount instead of the minimum at the launch of the system. The reason for this is that we know beforehand that customers are likely to order right after the order service goes live, so to anticipate this, we can try to have the maximum amount of servers ready to serve them.

Since we are dealing with a proof-of-concept, most of the functionalities are actually implemented, such as the encryption. Though, some parts are left out, because they would not contribute much to the load testing on this proof-of-concept; for example, generating a QR code is left out in this implementation. At the minimum, however, everything from the sequence diagram in Figure 12 is implemented to incorporate the functional and non-functional requirements.

A. Technology Stack

The technologies used to implement the proof-of-concept are listed in Figure 5. The important part is Docker and Kubernetes; all of the microservices and databases are put into their own container and then eventually managed by Kubernetes. Due to limited resources, a local setup was chosen using Minikube[9]. For the scaling mechanism, the Kubernetes horizontal pod autoscaler was used.

B. Load Testing Results

1) Database Sharding: From our simulations, it was not possible to determine the optimal amount of shards. Unfortunately, with our setup it was hard to get a representative

[8]https://flask.palletsprojects.com/en/1.1.x/api/#flask.Request.is_multithread

[9]https://kubernetes.io/docs/setup/learning-environment/minikube/

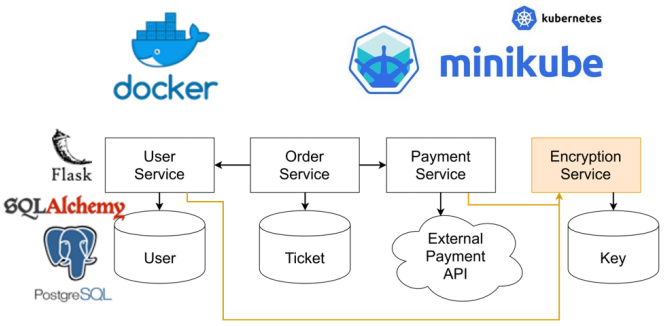

Fig. 5: Technology Stack

amount again since the whole setup was based on one machine. Anyhow, it is shown that having just one database will significantly slow down the response times. Whereas, having multiple shards improves the overall response time as can be seen in Figure 6. Note that in the limit, when there is only one ticket per shard, the complexity actually increases and will have a negative impact on the response time. When only a few tickets are left, the algorithm to select a shard, will need to iterate over all shards until it finds one that has tickets left, so this iteration might take some time as well. There must be a trade-off to select the right amount of shards; in this context, 350 shards are chosen as was in the simulations.

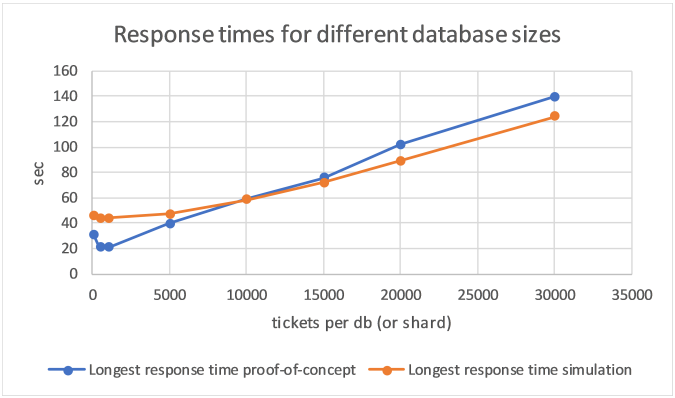

Fig. 6: Longest response time for different shard sizes when ordering 1000 tickets at the same time

2) Bottlenecks: To assure that the bottlenecks that were identified in the simulations, are actually bottlenecks, we can try to run the same configurations in the proof-of-concept and check if there are any noticeable changes.

The encryption functionality does not seem to be a troublesome bottleneck (see Table III). The longest response time takes roughly 6 seconds more with encryption and decryption. This small increase is probably affordable, especially when credit card information is stored in the database.

The biggest bottleneck that was identified was the dependency on the external payment API. Again, this result can be confirmed by the graph in Figure 7. Though, in the proof-of-concept, the slope is a little bit less, in other words, the dependency on this API is not as high as was predicted in the simulations, but it still remains a bottleneck in the system.

	w/ Encryption	w/o Encryption
Shortest Response Time	4.5 sec	4.29 sec
Average Response Time	90.57 sec	87.711 sec
Longest Response Time	178.023 sec	172.408 sec

TABLE III: Results for encryption when ordering 10.000 tickets at the same time

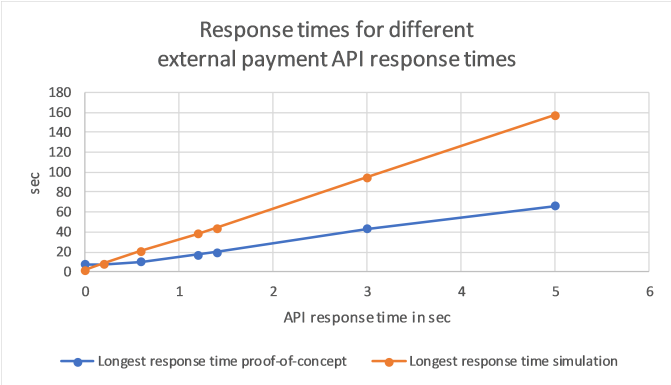

Fig. 7: Longest response time for different processing times of the external payment API

3) Servers Needed: Since this load testing is based on a system that is ran on just a single machine (or server), the results are not as reliable as in the real-world. For instance, when calculating the right amount of servers that are needed to keep the system running, it only holds for this specific setup, i.e. on one machine. Nevertheless, the optimal amount of replicas can be found in Figure 8, which is 7 in this case. It is assumed that in the real-world, when more resources are available, more replicas can be allocated across multiple servers.

It is also confirmed that having more replicas dramatically lowers the overall response time, because more requests can be handled simultaneously. One replica on its own can also process multiple requests concurrently[10], which will further decrease the amount of servers needed. Altogether, an exact number can still not be estimated, but the same conclusion from the simulations applies here; 1000 servers will probably suffice to keep the system running while still processing incoming requests at a decent rate.

4) Longest Response Time: Finally, through load testing, we can observe whether the system responds fast enough during the maximum peak load. Before that, a small comparison is made with the simulations, which can be seen in Figure 9. The reason why the proof-of-concept is initially faster than the simulation is because in the proof-of-concept the system is already scaled up to a high amount of replicas at the start, which was not the case during the simulations where the new

[10]https://flask.palletsprojects.com/en/1.1.x/api/#flask.Request.is_multithread

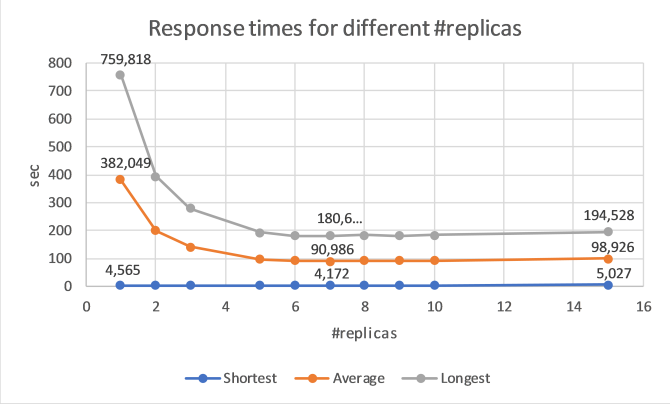

Fig. 8: Longest response time for different amount of replicas

pods need setup time. As mentioned before, the system under test is ran on a single machine, that is why the scaling does not work that well. In the real-world, it is expected that we can allocate more replicas and servers, which will result in a similar curve as the simulation one (blue curve).

Nevertheless, when counting the amount of tickets that could be processed by the system in 10 minutes, it resulted in 31.186 tickets. This is a lot less than we predicted in the simulation, but again, this is due to the fact that this was based on one machine.

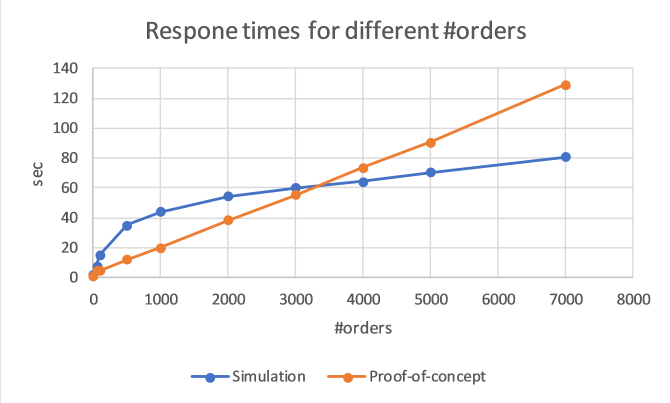

Fig. 9: Longest response time for different amount of tickets ordered at the same time

V. Future Work

Both the simulation and proof-of-concept were crucial steps in the whole development process of designing a good system. For instance, during the simulation, we could easily identify bottlenecks and come up with quick solutions that would normally cost a lot later in the development. By doing so, the database was found to be a bottleneck in our initial version, for which database sharding was incorporated. If this was not done, the cost of adjusting the system to work with multiple shards would be significantly higher during the implementation phase.

The proof-of-concept mostly confirmed the conclusions of the simulation, e.g. the external payment API is a bottleneck. It also showed that the results can be achieved in practise and that the system could actually be deployed on a cloud or rather using Kubernetes as a container orchestration system. Though, as mentioned several times, since the proof-of-concept was deployed on a single machine, the results should be re-evaluated in a real cloud environment when the necessary resources are available, i.e. around 1000 servers. By doing so, we can even get a more precise number on how many shards are needed or how many replicas are needed (at the start), which leads to the most optimal configuration of the system and the fastest response times. And most importantly, we can see how much load the system can handle while still responding in less than ten minutes. In this setup, it was demonstrated that one machine could handle 31.186 tickets while still respecting the requirements, but when using the actual cloud, this number would increase drastically.

During the simulation and in the proof-of-concept, the biggest bottleneck that was identified was this external payment API, but there was not a solution given to this. In our system, it is hard, if not impossible, to deal with this. An external solution might be to contact the corresponding credit card services and make some sort of contract in which they themselves scale their system during the time that our system goes live. This then leads to significantly faster response times as was shown that the response times heavily rely on this payment API.

VI. Conclusion

The main question of this feasibility study was to evaluate if an online ticket ordering system would benefit from a cloud setup, i.e. faster response times. More so, if scaling the system (elastically) would help dealing with the peak load. In our proof-of-concept especially, it could be seen that having several replicas would rapidly speed up the response times. This could even scale to a large amount (more than 10.000) with the result that the system responds faster. As a conclusion, deploying the system on the cloud is definitely a feasible solution to this peak load.

However, whether it respects the non-functional requirement of responding in less than ten minutes, still remains partially unsolved due to insufficient resources to perform the load testing. Though, from the simulations, it was predicted that if the load is no more than 106.300 tickets being ordered at the same time, the system can handle the peak load. As mentioned in Section V, it is best to re-evaluate the load testing and experiments in a real cloud environment to get a more precise result. One other thing to consider, which was not done in the simulation nor in the proof-of-concept, was calculating the optimal amount of shards, this could also significantly decrease the overall response time.

Anyhow, in this feasibility study, it was proven that the cloud setup would be a good solution to deal with peak load. Hence, we should proceed with the project and investigate the system more in a real environment to get the optimal configuration and more accurate results.

REFERENCES

[1] Tendelle, "6 ways to get tomorrowland tickets 2019 (even after missing general sale!)." https://travelalatendelle.com/6-ways-how-to-get-tomorrowland-tickets/. Accessed: 13/12/2019.

[2] P. Mell, T. Grance, *et al.*, "The nist definition of cloud computing," 2011.

[3] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," in *Present and ulterior software engineering*, pp. 195–216, Springer, 2017.

[4] P. Hamalainen, T. Alho, M. Hannikainen, and T. D. Hamalainen, "Design and implementation of low-area and low-power aes encryption hardware core," in *9th EUROMICRO conference on digital system design (DSD'06)*, pp. 577–583, IEEE, 2006.

[5] S. Bagui and L. T. Nguyen, "Database sharding: to provide fault tolerance and scalability of big data on the cloud," *International Journal of Cloud Applications and Computing (IJCAC)*, vol. 5, no. 2, pp. 36–52, 2015.

APPENDIX A
CLASS AND DEPLOYMENT DIAGRAM

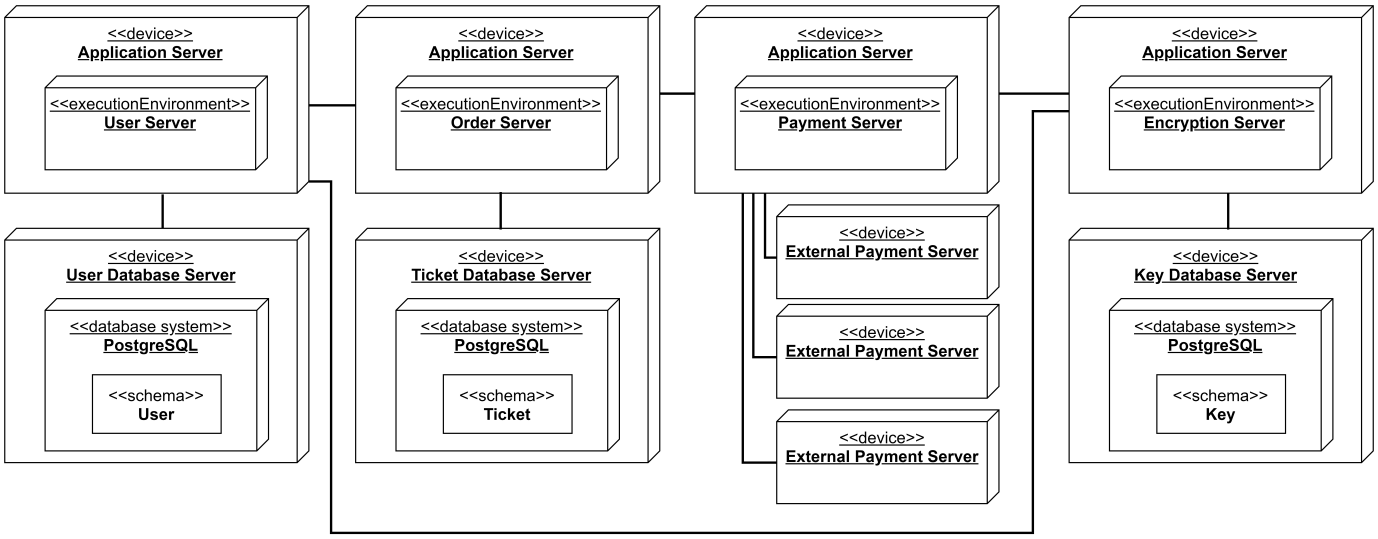

Fig. 10: Deployment Diagram of the System

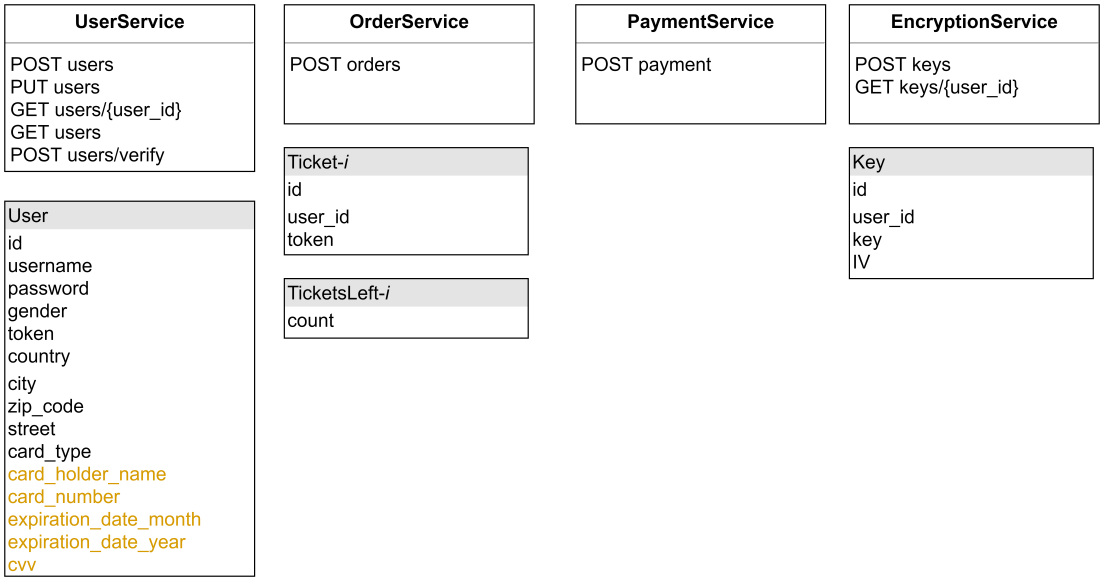

Fig. 11: Service API's and Database Schemas

APPENDIX B
SEQUENCE DIAGRAM

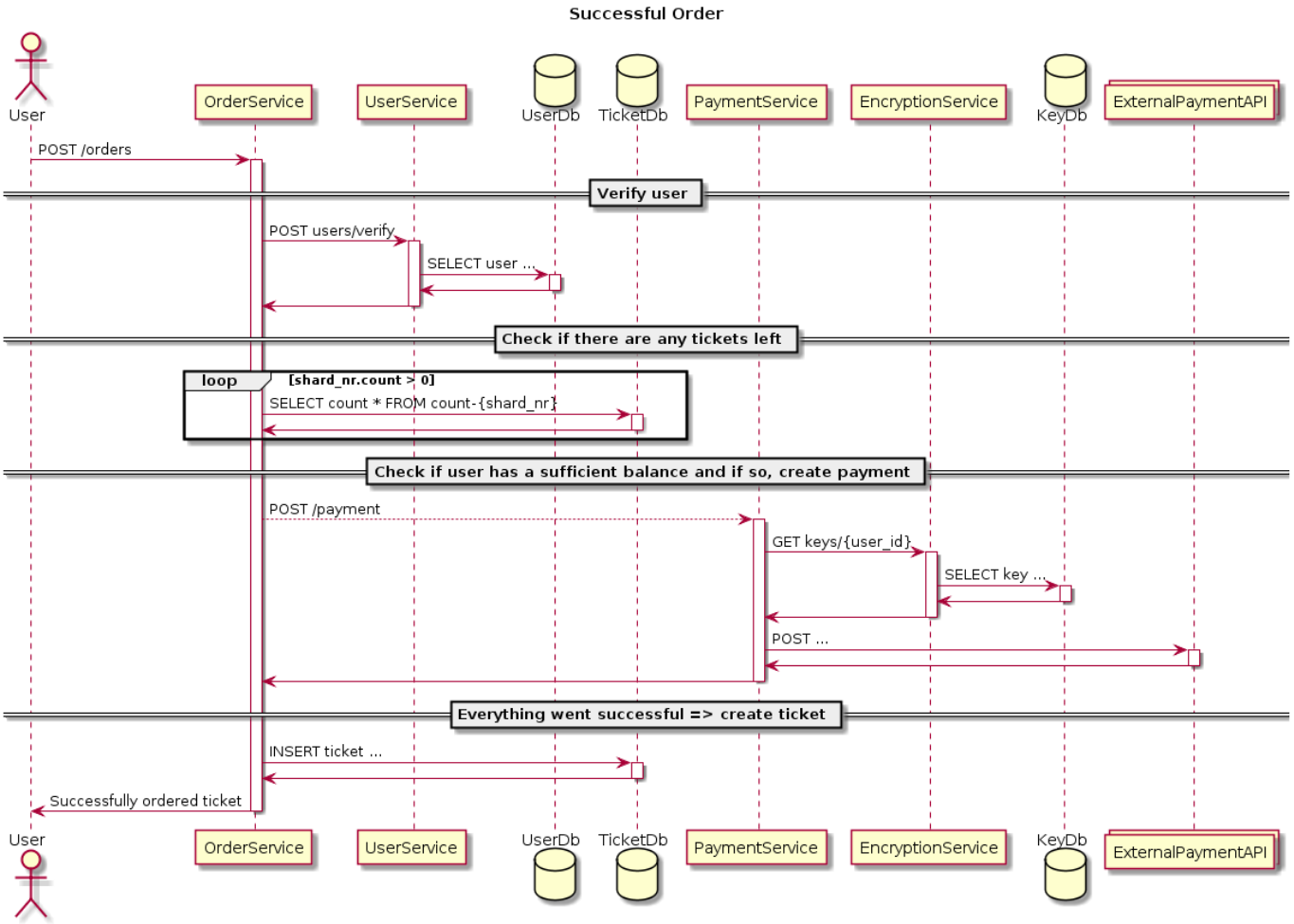

Fig. 12: Sequence Diagram of Ordering a Ticket

APPENDIX C
REPOSITORY REFERENCE

All the code and resources used in this paper are publicly available in following GitHub repository: https://github.com/ZhongXiLu/CapitaSelectaSE. To be precise, the code for the ABS simulation can be found in the *abs* directory or downloaded via following url: https://github.com/ZhongXiLu/CapitaSelectaSE/releases/download/v1.0/abs.zip and the implementation of all the microservices in the *services* directory. There is also a small *README* attached for instructions on how to setup the system (in minikube) and perform the load testing.