LockSpace : A Cloud-Based Room Reservation System

Jasmeet Khalsa
Department of Computer Science
University of Bristol
Candidate No.:97313
email: lu18275@bristol.ac.uk

Tushar Bhatnagar
Department of Computer Science
University of Bristol
Candidate No.:97607
email:wq18309@bristol.ac.uk

Samia Mohinta
Department of Computer Science
University of Bristol
Candidate No.:97293
email: dc18393@bristol.ac.uk

***Abstract*—This paper presents an implementation of a reservation system to book study rooms across the University of Bristol campus. In particular, LockSpace is a cloud-based, Slack integrated live chat service prototype hosted on Amazon Web Services, which can be used to book study rooms inside different University buildings by students, faculty and staff. The application can be accessed via Slack and the details are at:**

https://github.com/jaskhalsa/lex-book-studyroom

Kindly follow the instructions given the readme page of the GitHub repository to interact with the chat bot through Slack.

Keywords—Cloud Computing, Room Reservation System, Amazon Lex Chat Bot, Amazon Web Services, Scalability

I. INTRODUCTION

Almost every area of human endeavor that involves the use of shared resources relies on a reservation system to manage the booking of those resources [1]. Hotels, restaurants, railways and even universities rely on reservation systems to ensure the optimum and harmonious use of the assets over time so as to strike a balance between supply and demand. However, the introduction of on-demand, pay-as-you-go cloud computing technologies has transformed these legacy computer reservation platforms to a next generation system, now based on the IT industry's latest technology. Hosting in-house reservation systems is no longer a viable option owing to the inflexibility and cost-ineffectiveness of maintaining these systems. Consequently, building a reservation system that utilizes cloud services is perhaps the most economic and convenient option for developers, as it eliminates the need for any upfront capital expenditure and hands-on attention to the underlying architecture of an application environment.

II. MOTIVATION

Nowadays, customers live in a multi-channel world, and expect to book a table in a restaurant or a hotel room via a brand's website, by using an app on a mobile device, on the phone or in person [2]. As a result, diverse sectors are adapting their systems to offer all these channels, synchronizing and updating the same in real-time by making use of leading-edge IT innovations and technology, such as the cloud. An online, highly available, reliable cloud-based booking solution is not only easy to implement but also is extremely cost efficient and ensures effective customer service. We identified that we could extend these advantages by building a cloud-based platform to book University study rooms to benefit the students and staff alike, as there is no such current platform in the University of Bristol. Moreover, the present cumbersome process of visiting a University building's library to check for study room availability and also reserve one if needed, was also major motivator towards taking up this project.

Additionally, if the amount of communication done through textual interactions is considered, it can be noted to almost infinitely exceed voice and social communication [3]. Developing a text-based booking system chat bot can therefore be a fun and interactive way to allow students and staff to reserve study rooms as per their needs, while still using a common messaging platform.

III. CLOUD SERVICES USED FROM AWS

A. Amazon Lex

Amazon Lex is a service that enables building of conversational interfaces that can be integrated with any text or voice chat-bot on mobile, web or in other chat services [4]. As a fully managed Amazon service, Amazon Lex also scales automatically as the bot's usage increases or decreases, thereby allowing application developers to only pay for the number of text or voice queries processed by Amazon Lex.

Amazon Lex incorporates advanced deep learning functionalities of automatic speech recognition (ASR) for converting speech to text, and natural language understanding (NLU) to recognize the intent of the text [5], which supports our motivation of building an easy to use conversational bot to enable seamless booking of study rooms.

B. Amazon Lambda

AWS Lambda is a compute service that has been designed to work in response to events. In other words, AWS Lambda can invoke application code while reacting to events like changes to data in an AWS database table or receiving HTTP requests using an Amazon API Gateway. It enables developers to build serverless applications composed of functions that are triggered by events [6].

Additionally, AWS lambda facilitates continuous scaling of the application depending on the size of the workload. The code runs in parallel for each trigger individually. The added advantage of sub-second metering also prevents developers from hourly rates for using the service, unlike using AWS EC2 [7], thereby saving money when the application is not in use at all.

C. Amazon DynamoDB

DynamoDB is a non-relational database offered by Amazon Web Services. It is a NoSQL database service that is fast, highly reliable and cost-effective for internet scale applications [8]. Amazon DynamoDB allows developers scaling cloud-based applications to begin with the capacity required and then increase the request capacity of a table as the demand improves [9]. Moreover, since it is a service and not a database software, AWS does all the heavy lifting including automatic data replication across multiple IT stacks or availability zones to ensure fast, available and secure customer databases.

Crucially, in our application the only capacity we need is to be able to look up an entry, based on a value, which DynamoDB is perfect for, as it supports key-value and document data structures. Additionally, by virtue of being a provisioned auto-scaling service, we no longer need to worry about the underlying operational and scaling concerns, which the service automatically handles with growing datasets. As a result, it takes away the issues of having to set up our own database on a server for our application, which would be both inefficient and scale poorly.

IV. IMPLEMENTATION

A. Building the Conversational Interface using Amazon Lex

1) Adding Intent

We manually created and configured our custom bot for booking study rooms in the University and for that we first had to add an intent to the Amazon Lex console. An intent is basically an action that a user wants to perform by providing minimum information. Since, our application intends to book study rooms, we added an intent called 'BookStudyRoom' to the Lex Console. This intent will allow the users to fulfill their intention of booking study rooms by gathering information from a series of text conversations between the bot and the user. However, for the bot to function, we need to add specific slots, the values of which helps the bot to identify the intent of the user.

2) Slot Types and Slots

Slot types are parameter values that an intent takes in as input. With our application, a user will be able to booking a room in a specific building of the University and for a particular session. Thus, location of the room (RoomLocation) and session (BookingSession) serve as our Slot types.

In an ideal University study room booking scenario, staff and students can book rooms any day for a certain number of hours starting from a specified time. That is, a room can be booked on 28-01-2019 for two hours starting 14:00 at the Queen's building. However, to simplify the implementation of our prototypical solution, we have enabled bookings based on sessions and not on calendar dates and time. In our application, a user is allowed to book a room for specified number of hours for only two sessions across the university. For example, a student can book a room any day for a morning or evening session for two hours at the Queen's building.

Slot type 'RoomLocations' was created to include five major buildings (Queens, Merchant Ventures, Social Sciences, Hawthorns, Senate House), which are generally the most used by the students and house group study room facility inside them. The remaining university spaces can also be incorporated into the application, but it lies in the scope of our future work of enhancing this prototype to a fully functional app version.

3) Configuring the Intent – Adding Sample Utterances

The next logical step was configuring the intent to fulfill a user's room booking request. We added some sample utterances, which are strings with slot names passed as parameters, for the bot to identify an action the user wants to perform. These set of strings when typed into the chat bot, will trigger a conversation between the bot and the user as well as entitle the bot to probe the user for other necessary information to fulfill the intent.

4) Error Handling

Amazon Lex bot needs to be configured to probe the user for important information when it does not understand a user's intent. The bot responds to a user with a message asking for clarification (ClarificationPrompt) and waits for an understandable input. This clarification prompt can be repeated for a fixed number of times to elicit required information, before the bot goes into aborting the conversation with an abort message. Clarification and abort prompts are set under error handling in the Lex console. We have allowed our bot to try to get a clarification for a maximum of five times before it ends a conversation with the user by displaying a goodbye message.

B. Creating DynamoDB tables

For our application, we constructed two database tables, namely 'locations' table to store location details of every room across the university buildings and 'rooms' table containing items such as roomID and session values. The 'locations' table is treated more like a metadata table, containing an overall information of which rooms are housed under which building of the university. On the other hand, the 'rooms' table is a transactional database table, wherein reads and updates specifying bookings are fired. That is, when a room booking request is placed through the bot, this table is first checked to ensure availability and if a room is available for the requested session, an update statement is fired to reserve that session. In our tables, we have maintained session values for 28 rooms spread across the five university buildings and the availability values of rooms are refreshed twice every day, once before the start of the day and once in the evening with the help of a CRON (time-based job scheduler software utility) job that calls a Lambda function used to reset all session values to available for every room.

Since, our application is mostly based on static data, only required to verify and book the same set of rooms across locations, we have not implemented database scaling features of DynamoDB. In case, we had required to store user information upon signup, we could have incorporated the auto-scaling measures in the database. However, this is currently not required for the application to work.

C. *Using AWS Lambda*

AWS Lambda is a Function as a Service (FaaS) platform offered by Amazon and used for building a microservices application (an application consisting of loosely coupled software modules) based on a serverless architecture. Using AWS Lambda prevents wastage of resources because the code inside a Lambda function is only executed in response to an event. More simply, the Lambda function just sits as a code file, while waiting for a trigger event, which in our case is the intent to book a study room. When our Lex bot captures such a user intent and after it has gathered all the necessary information required for fulfilling the action, it triggers the Lambda function and our code executes to validate the user inputs with the database table values and thereby commences with a booking.

AWS also manages scaling of the function to accommodate growing needs. It dynamically scales the function execution in response to increased traffic and the function's concurrency bursts to an initial level between 500 and 3000 concurrent executions that vary per region [10], under the effect of a sustained load. The function's capacity increases by 500 concurrent executions thereafter, till the load is handled or the concurrency limit is hit. We have made use of this auto-scaling feature supported by AWS Lambda to test load handling in our application and the results are described further down the report under section V.

A complete process flow diagram of the interacting AWS services is given in Figure 1.

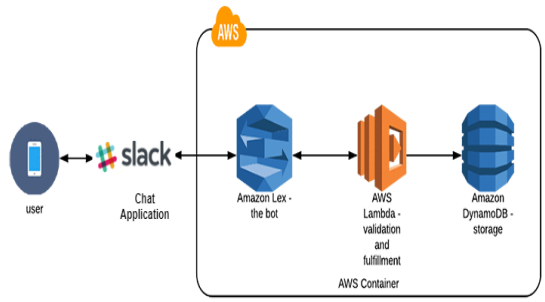

Figure 1 **:** Process flow diagram of the interacting AWS services

D. *Social Media Integration - Slack*

Slack is an acronym for 'Searchable log of All Conversation and Knowledge'. It is a cloud-based messaging application that is often used at workplaces to communicate and collaborate inside the organization. Slack supports integration with a large number of third party and community built services. Slack was chosen as an integration platform for the bot, because it is free to use and hosts almost all other functionalities found in most renowned messaging platforms, such as Facebook Messenger and Twilio. Thus, having a Slack account will not only enable users to communicate easily, but also will prevent the 'app fatigue' associated with using multiple communication platforms.

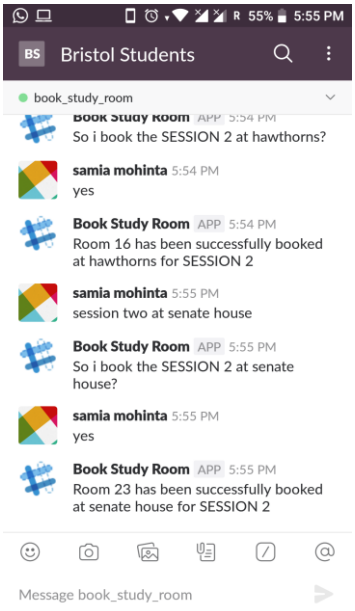

Figure 2: Chat service User Interface in Slack

Integration of our Amazon Lex Bot with Slack first required creation of a Slack messaging application using the Slack API Console. The application was then configured to add bot users and interactive messaging features. The integration with Slack was completed by setting a few credentials across the Amazon Lex and Slack API consoles.

On the whole, the Slack messaging platform can now be used to chat with our study room booking bot, thus serving as the user interface (Figure 2).

V. TESTING APP SCALABILITY

The application's scalability has been tested in terms of increase in the number of concurrent executions of the AWS Lambda function with growing traffic. We have used Locust, a python-based, distributed user load testing tool [11] to generate swarms of parallel users that the AWS Lambda via AWS API Gateway. Figure 3 displays a step by step sequence of the testing routine.

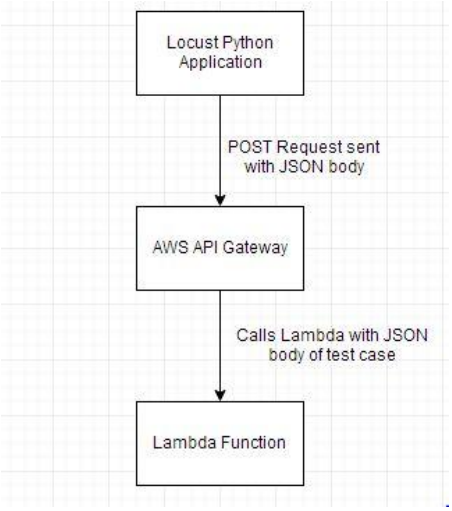

Figure 3 : Flowchart showing how the Lambda function was called from the Locust Web UI via AWS API Gateway.

We tested the scaling performance of our application with 2000 concurrent users and then used Amazon CloudWatch to collect and track application's scaling with respect to increase in concurrent Lambda function executions and the response times from the application as the traffic increased. Figure 4 displays the scaling graph of the concurrent executions. It can be seen that the number of instances of lambda functions gradually grow as the number of incoming concurrent POST requests increase through the API Gateway. However, it is also noted that the scaling abruptly stops at 1000 user requests, and is believed to be because 1000 is the default number of concurrent executions allowed for AWS Lambda until changed manually in the console.

Figure 5 shows the minimum, maximum and average response time of the Lambda function as it records heavy traffic. From the graph, it is evident that the maximum response time does increase with increasing number of users. However, the minimum response time almost remains the same for the entire duration even when application is hit by more users. The average time to respond to requests also increases with greater number of users. In other words, with a large number of parallel invocations of the function, it is possible that the function takes a longer time to respond to requests of some users.

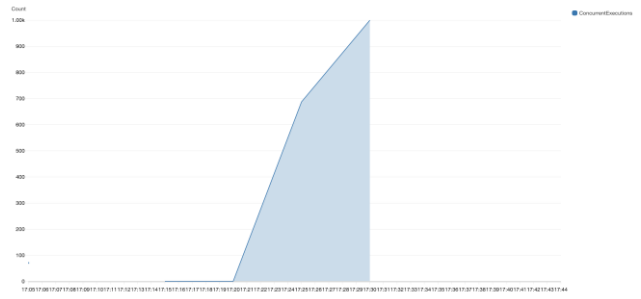

Figure 4 : Concurrent Executions Scaling graph on Amazon CloudWatch. Scaled to mimic 1000 parallel users using the application.

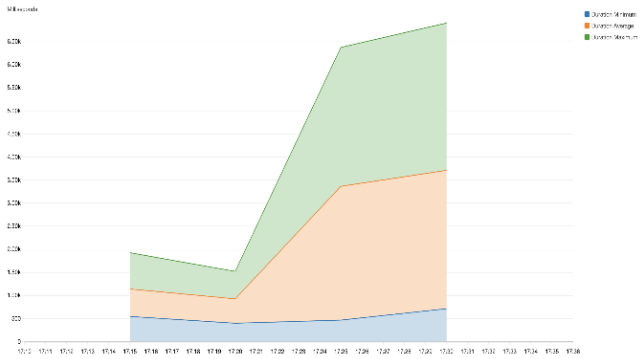

Figure 5: Graph Shows Response Time of Lambda Function with increased number of users. Legends: (Blue) Duration Minimum, (Orange) Duration Average, (Green) Duration Maximum.

Our load testing results assure that our application will definitely be able to handle large numbers of user requests in a real-world scenario.

VI. PITFALLS

A. *AWS Lambda Cold Starts*

A cold start occurs when an AWS Lambda function is invoked after not being used for an extended period of time resulting in increased invocation latency [12]. Since, our application was not always in use, we faced an issue of a cold start on infrequent invocations. This is believed to be because the container created in response to one invocation is not reused for a triggers occurring at a later point in time [13]. So on some requests, AWS needs to re-provision a container with the code before it can process the request. By warming up the container (keeping it active), we save the overhead of provisioning a new container, which improves the response time for initial requests.

As we anticipate a traffic load at 9am every day, we keep the function 'warm' using a CloudWatch event, which runs as a CRON job and triggers the Lambda function at 8.57 am, 8.58am and 8.59am every day. Figure 6 shows a graph for the cold start proof of concept. The request purposely sent payload for a study room with an incorrect location. The first invocation from the CRON job was at 8.57 am, and we can see that the function execution duration decreases for subsequent invocations. The last one is probably faster than the second due to caching, where the request invokes the same response each time.

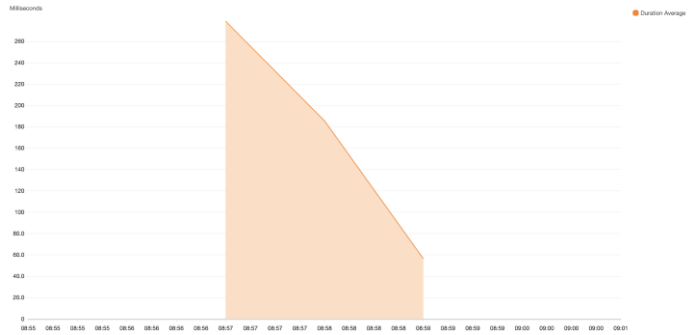

Figure 6 : Average Response Time decreasing with the Lambda function being warmed up.

VII. FUTURE ENHANCEMENTS

A. *Incorporating bookings based on Date and Time*

Our prototype application supports bookings based on sessions. That is, a room can be booked for either session one or session two. Thus, as a future enhancement, we plan to implement bookings based on date and time of a calendar day. This will improve the application's suitability in a practical scenario.

B. *Room use tracking*

Currently, our application provides an easy to use interface to University students and staff to book study rooms through a messaging service. However, there is no way we can check if the booked room is being used. In future, we plan to add this feature to our application to track

the use of rooms booked so as to promote optimum use of University assets.

All booked rooms when in use need to be flagged up by the user by sending a message to the same chat bot. The bot will register this as a new intent and call an AWS Lambda function, which in turn will change the values in the database table to mark the room in use. On the other hand, if a room is not flagged up within 15 minutes of its scheduled booking time, it will released to the pool of available rooms, allowing queued users to book it.

C. Tackling cold starts with Thundra

Operational maintenance to decrease the response time of the lambda function when invoked after certain gaps in time involves a lot of effort into setting up and randomizing warm-up calls. Thundra, an observability and monitoring platform for serverless architectures [14], provides a warm-up module that be easily configured to prevent cold starts. We wish to use the features in Thundra to transform this prototype into a robust and deployable application.

D. Wider Applicability

The application currently focuses on just booking study rooms in the University, but its functionality can be extending to booking guest rooms, conference halls and even rooms inside student halls of residence. Also, there is a scope to develop this application into an open-source platform that is not restricted to booking University resources, instead can be used to book hotels, tables in restaurants and other reservable resources across cities. The application can also be integrated with other chat and voice based communication platforms to reach a wider customer base.

VIII. CONCLUSION

In this paper, we have presented a prototypical cloud-based chat service integrated on a messaging platform called Slack, which can be used to book study rooms by students and staff inside five important buildings of the University of Bristol. It was found that no such service currently exists and therefore, this novel application can be highly effective in making the lives of the University students and staff more convenient. It is an easy to use chat service that involves sending text messages to a virtual chat bot, requesting it to book a study room in a preferred University building and for a specified session.

The application has been tested to perform against a heavy user load and the results have been discussed under section V. The application is still in a prototypical stage and requires incorporation of many further enhancements to turn it into a complete product. A few future enhancements have been discussed in section VII.

We believe, the novelty of this application will be highly appreciated by its users, once it is introduced as a finished product in the University of Bristol community.

REFERENCES

[1] Andrew Hillier, “Capacity Reservation System for Virtual and Cloud Environments”. Retrieved from http://docs.media.bitpipe.com/io_10x/io_106304/item_600750/CiRBA-wp-Capacity-Reservation-System%2011.1.12.pdf

[2] Elly Earls, “The power of restaurant reservation systems”. Retrieved from https://www.thecaterer.com/articles/494589/the-power-of-restaurant-reservation-systems

[3] Vivian Michaels, “Is Chatbot a synonym for great Customer Experience?”. Retrieved from https://chatbotslife.com/is-chatbot-a-synonym-for-great-customer-experience-c40515e713c

[4] Sarah Perez, “ Amazon Lex, the technology behind Alexa, opens up to developers”.Retrieved from https://techcrunch.com/2017/04/20/amazon-lex-the-technology-behind-alexa-opens-up-to-developers/

[5] Amazon, “ Amazon Lex ” . Retrieved from https://aws.amazon.com/lex/

[6] Amazon, “AWS Lambda”. Retrieved from https://aws.amazon.com/lambda/

[7] Amazon, “How are EC2 instance hours billed?”. Retrieved from https://aws.amazon.com/premiumsupport/knowledge-center/ec2-instance-hour-billing/

[8] Werner Vogel, “All Things Distributed”. Retrieved from https://www.allthingsdistributed.com/2012/01/amazon-dynamodb.html

[9] Jack Clark, “Amazon switches on DynamoDB cloud database service”. Retrieved from https://www.zdnet.com/article/amazon-switches-on-dynamodb-cloud-database-service/

[10] AWS Lambda - Automatic Scaling . Retrieved from https://docs.aws.amazon.com/lambda/latest/dg/scaling.html

[11] Locust. Retrieved from https://locust.io/

[12] Yan Cui, “I am afraid about AWS Lambda cold starts all wrong”. Retrieved from https://hackernoon.com/im-afraid-you-re-thinking-about-aws-lambda-cold-starts-all-wrong-7d907f278a4f

[13] Sam Corcos, “ How To Keep Your Lambda Functions Warm”. Retrieved from https://read.acloud.guru/how-to-keep-your-lambda-functions-warm-9d7e1aa6e2f0

[14] Thundra. Retrieved from https://www.thundra.io/