



Takhatta

Bahrain Airport Company is revolutionizing the Airport's Resource Allocation by using predictions calculated using an Event-Driven Architecture (Takhatta)

A thesis

by

Hasan Ali Marhoon
201802525

Bachelor of Science

in Information & Communication Technology

at

Bahrain Polytechnic

January 2023

Title

Bahrain Airport Company is revolutionizing the Airports Resource Allocation by using predictions calculated using an Event-Driven Architecture (Takhatta)

Copyright

Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

Signature	Hasan Marhoon
Name	Hasan Ali Marhoon
Date	7/1/2023

Abstract

Bahrain Airport Company is looking to effectively allocate human resources to the everchanging staff schedules due to inconsistencies in the timings and sizes of departing flights and passenger arrival time. Takhatta is a collection of systems, which at its core makes simulation-based predictions for the number of check-in employees needed each day for a month in advance. The system also records the passenger's queue time, gets feedback from passengers and notifies management in urgent cases such as high passenger traffic. The system implements an event-driven architecture, interconnecting all the subsystems together as they directly affect each other. The system consists of manager/check-in employee/passenger dashboards as well as a virtual queueing system. The system's infrastructure is scalable and runs at a low monthly cost. The system measures its own success, returning metrics and visual insights to the staff managers regarding the average wait time, average satisfaction responses, predictions etc. The designed architecture resulted in a system that solves an issue while simultaneously measuring its own success. End-to-end testing against historical data has directly reflected how accurate the system's predictions are and the cost-efficiency of implementing the new system and moving away from the system that is currently in place. The report aims to briefly describe the system in full, while focusing on the specifics of the virtual queueing system.

Table of contents

Contents

Title	2
Copyright.....	2
Declaration.....	2
Abstract	3
Table of contents.....	3
List of Figures.....	4
List of Abbreviations.....	5
1. Introduction	6
2. Background work.....	7
3. Methodology.....	11
4. Discussion.....	32
Conclusion	36
References	37

Appendices	40
------------------	----

List of Figures

Table 1 Technologies used	11
Table 2 requirement gathering methods.....	11
Table 3 functional user stories.....	13
Table 4 non-functional user stories.....	14
Figure 1 Takhatta system architecture	15
Figure 2 ticketing subsystem architecture	16
Figure 3 On-premises tablet: Home page.....	17
Figure 4 On-premises tablet: ticket page.....	18
Figure 5 User device: ticket display.....	19
Figure 6 Passenger virtual queueing system -Activity diagram	20
Figure 7 Virtual queueing system - sequence diagram	21
Figure 8 hosting a static website on S3	22
Figure 9 MainPage.html final implementation	23
Figure 10 JavaScript timer code	23
Figure 11 TicketDisplay.html - button displayed	24
Figure 12 TicketDisplay.html - button hidden	24
Figure 13 code snippet - change button display	24
Figure 14 satisfaction survey	25
Figure 15 passenger dashboard.....	25
Figure 16 IAM role - TicketCount function	25
Figure 17 DynamoDB streams - LatestTicketByZone table	26
Figure 18 NewTicketFunction - check ticket number.....	27
Figure 19 TicketCount function - check event type, put new record.....	27
Figure 20 TicketCount function - post to connection	28
Figure 21 TicketCount function - check if management has to be notified.....	28
Figure 22 sample of email notification	28
Table 5 functionality test – participants	30
Table 6 functionality test- matrix.....	31
Table 7 acceptance test – participants.....	31
Table 8 acceptance test – results.....	32
Table 9 LESPI.....	35
Figure 23 Project plan	40
Figure 24 user storyboard – problem	41
Figure 25 user storyboard – solution.....	42
Figure 26 BAC - Takhatta press release.....	43
Figure 27 BAC - Takhatta FAQ	44
Figure 28 Virtual queueing system - use-case	45

Figure 29 source code- NewTicketFunction 1.....	46
Figure 30 source code- NewTicketFunction 2.....	46
Figure 31 source code – PutToQueue	47
Figure 32 source code - onConnect.....	47
Figure 33 source code – onDisconnect	48
Figure 34 source code - TicketCount 1	48
Figure 35 source code - TicketCount 2	49
Figure 36 source code - TicketCount 3	49
Figure 37 source code - TicketCount 4	50
Figure 38 source code - TicketCount 5	50
Figure 39 table - LatestTicketByZone	50
Figure 40 table - TicketCountTable.....	51
Figure 41 table - Ticket_Count_Connections.....	51
Figure 42 table - CountersOpened	51
Figure 43 Takhatta - cost summary.....	52

List of Abbreviations

BAC	Bahrain Airport Company
SNS	Simple Notification Service
SNS	Simple Queue Service
S3	Simple Storage Service
IAM	Identity and Access Management
API	Application Programming Interface
RDS	Relational Database Service
ARN	Amazon Resource Name
FIFO	First-In-First-Out

1.Introduction

Bahrain airport moved to a new facility in 2021 and BAC (Bahrain Airport Company) handles the operations of the new airport. BAC's vision is to evolve the airport experience for both the passengers and the staff by introducing new technologies that abide by the principles of automation.

1.1.Project rationale

A prominent issue that BAC is currently facing is ineffective allocation of staff (check-in employees), which causes overstaffing in turn unnecessarily increasing the operational costs, or understaffing causing high passenger traffic in the check-in areas and limiting the time that passengers have to enjoy the various airport facilities. It is vital for BAC to eliminate the manual system of allocating staff to achieve their vision of revolutionizing the new airport and increase their end of year profit margin. BAC does not have a process to record feedback from all the passengers that go through the check-in areas, which would undoubtedly prove useful in creating performance reports. Lastly, recording the actual queue time of each passenger enables the staff management to judge their employees' performance and make last second managerial decisions in real-time.

1.2.Project objectives

The main objective of the system is to aid management in allocating staff to step away from the manual system, analyze its success and improve passenger experience. This is achieved by:

Staff allocation:

- Making daily simulation-based predictions for the upcoming months using flight capacity, flight departure time and intervals of passenger arrival times.
- Manager dashboard

Success analysis:

- Introducing a virtual queueing system which
 - Places passenger in queue
 - Enables management to track employee performance
 - Gives precise statistics of actual passenger queue time
 - Enables passengers to enjoy airport facilities instead of waiting in line
 - Notifies management on high passenger traffic
- Check-in employee dashboard
- Satisfaction survey

Passenger experience:

- Passenger dashboard

1.3.Proposed solution

The core issue that the organization is currently facing is inappropriate allocation of resources. Takhatta introduces a simulation-based event-driven system to take in the required parameters and automatically make predictions for the number of staff needed per flight. Takhatta goes above and beyond this issue to meet the other requirements gathered from the solution.

Takhatta composes of the following:

- Manager dashboard: displays all relevant data to the staff managers. Example: predictions, average queue time, average satisfaction feedback etc.

- Satisfaction survey: records passenger's feedback on their experience in using the virtual queue. The survey consists of one question, with preset answers (1-5).
- Check-in employee dashboard: gives the check-in employee the power to manage the virtual queue (next ticket, ticket announcement etc.)
- Passenger dashboard: enables passengers to lookup their flight and get details. Example: gate status, flight status etc.
- Virtual queueing system: assigns a ticket to the passenger and places them in a virtual queue and notifies management on high passenger traffic.

The combination of all these components resolve the client's problem while also displaying evidence of the success of the system. **This paper focuses on the virtual queueing system.**

1.4. Description of the report

The following parts will cover the background work done, methodologies, and a final discussion of the virtual queueing system and its place in the overall solution.

In this report, the methods of requirement gathering and description of the final – functional and non-functional - requirements are under 3.1. Requirements, diagrams on the system's functionality and the overall system architecture under 3.2. Solution design, a detailed description on the implementation of the virtual queueing system under 3.3. Implementation and the functionality of the system under 4.1. System functionality.

2. Background work

2.1. Related theory

2.1.1. Event-driven architecture

An event-driven architecture ("Event-Driven Architecture," 2023) is an architecture that triggers and communicates with the rest of the system upon any event, depending on the event type. Event-driven architectures are widely used in modern service-based applications and gives the developer more freedom and control over the application and how it handles events.

Event-driven architectures consist of:

- Event producers
- Event routers
- Event consumers

For example, in the virtual queueing system for the displaying the count of tickets ahead to the user (user device),

- the event producer is the DynamoDB table, a modify event is triggered upon changes to the ticket currently being served
- the event router is the DynamoDB stream behind the same DynamoDB table, routing the change in the table to the backend Lambda function which returns the count to the user's screen
- the event consumer is the backend function, consuming fields of the modified table and performing backend functionality.

Event-driven architectures are very beneficial to the developers as it scales and fails independently, not affecting other microservices in the application, and cuts costs as these triggers are push-based, and the organization is billed per event.

2.1.2. *AWS services*

AWS provides its users with over 200 microservices enabling them to achieve their goals with freedom and without having to worry about server configuration (Barney & Gillis, 2022).

Some of the services provided by AWS include:

- Databases
- Artificial intelligence
- Notifications

Other than the ease of configuration, the services provided by AWS are beneficial to its users. These benefits include but are not limited to:

- Availability – AWS has data centers in more than 85 availability zones, making the servers available for users almost globally at a low latency.
- Pay-as-you-go billing – with AWS, the user does not pay upon deployment but is billed at a specified date depending on the usage. This is excellent for startups or companies with a low income that cannot make big financial commitments.
- Monitoring – AWS enables users to monitor the usage of all of the services provided, from billing costs to more technical aspects such as requests.

2.1.3. *Serverless application*

Amazon describes serverless applications as “A serverless architecture is a way to build and run applications and services without having to manage infrastructure. Your application still runs on servers, but all the server management is done by AWS. (“Serverless Architectures,” 2023).

Serverless applications are automatically provisioned and the user does not worry about managing/monitoring the infrastructure.

There are benefits that are shared between all serverless applications (“Why Use Serverless Computing? | Pros and Cons of Serverless,” 2023):

- Serverless applications are inherently scalable
- Can be updated and rolled out with ease
- Decreased latency

2.1.4. *API*

APIs (application programming interface) are used to connect the frontend with the backend and work by sending requests and receiving responses. Most common requests are:

- GET
- DELETE
- POST
- UPDATE





These requests are sent to the backend, which runs code depending on the type of requests sent and returns a **response** (typically in JSON).


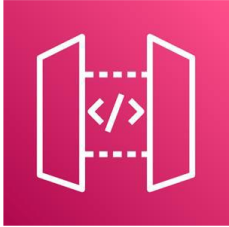


An example of an API is a twitter bot, that automatically sends a welcome message to any new followers. Different APIs have different protocols that separate them from others (“Difference between Rest API and Web Socket API - GeeksforGeeks,” 2020), for example:

- REST API – user has to send requests to receive responses
- WebSocket API – bi-directional communication, meaning that once the user is connected, they can keep receiving messages as long as they’re still connected.

2.2. Project technology

A table of the chosen AWS services in the deployment of the virtual queueing system and the motivation of choosing the service over alternatives:

Technology	Description	Motivation
 AWS Lambda	<p>Lambda functions are vital for developing scalable, serverless web applications. Lambda functions are the backbone of any application, they contain the code and give the user distinct options on how/when the function will run.</p> <p>Lambda features database access, scaling controls and even a feature called lambda layers. Lambda layers enable users to install libraries and attach them to the code (“What Is AWS Lambda? - AWS Lambda,” 2023).</p>	<p>For the purposes of this project, lambda functions have been used to carry out the backend functionality as they are combined with and triggered by other AWS services to return responses or make changes in the databases along with other use cases.</p>
 AWS DynamoDB	<p>DynamoDB is a NoSQL database fully managed and provisioned by AWS. NoSQL databases store data in a different format from relational databases, and attributes (items) can be added on the go.</p> <p>DynamoDB offers high durability and availability, by spreading data over multiple servers to decrease latency and data is stored in more than one availability zone, in case of failover (“What Is Amazon DynamoDB? - Amazon DynamoDB,” 2023).</p>	<p>There were two alternatives for data storage: Timestream and RDS (Relational Database Service).</p> <p>DynamoDB eliminates the restrictions of RDS and is more reliable and offers higher, more consistent speed for data processing than Timestream (which is typically used for time sensitive data storage).</p>
 AWS SNS	<p>SNS is a notification system which gives the user two notification options, either SMS notifications or email. SNS sends messages to subscribers and can be called in a lambda function using its ARN (Amazon Resource Name).</p>	<p>There were no AWS services that could have been used as an alternative. SNS is essential in the system to meet the requirements set by the client.</p>
 AWS SQS	<p>SQS is a service that creates and queues that can be integrated with other functions – like lambda functions.</p> <p>SQS has two types of queues:</p> <ul style="list-style-type: none">• FIFO (First-In-First-Out) queue• Standard queue	<p>FIFO queues have been used in the implementation for the following reasons (“Amazon SQS Queue Types - Amazon Simple Queue Service,” 2023):</p> <ul style="list-style-type: none">• Messages are processed exactly once before being deleted

		<ul style="list-style-type: none"> • The order of the messages is kept the same • Supports 3000 calls per second
 <p>AWS S3</p>	<p>S3 is a storage service which enables users to store all types of files from HTML files to MP4 files. S3 offers storage management, access management, storage monitoring among many, many other features (“What Is Amazon S3? - Amazon Simple Storage Service,” 2023).</p>	<p>S3 is used for the static hosting of all of the webpages of the system. All of the pages of the web application are stored in S3 buckets, and access to data can be managed directly in the AWS console.</p>
 <p>AWS API Gateway</p>	<p>AWS API Gateway is used to create and configure APIs at any scale. AWS offers three types of APIs: RESTful API, WebSocket API and HTTP API (“What Is Amazon API Gateway? - Amazon API Gateway,” 2023).</p> <p>API Gateway enables access to AWS services from the frontend, so a Lambda function in the backend can be invoked from the frontend.</p> <p>API Gateway also enables users to pass data from the frontend, depending on the type of the deployed API.</p>	<p>In the implementation of the system, REST API and WebSocket API are required to invoke lambda functions as necessary.</p>
 <p>AWS IAM</p>	<p>IAM is a service to control access permissions between services. A Lambda function that does not have the correct permissions to send notifications, cannot use SNS.</p> <p>IAM is a security layer which ensures that the application abides by the rules of least privilege. Access to other services can be controlled with modifiable configuration. A lambda function can have permissions to send notifications using SNS, but its functionality is limited to that if it does not have permissions to add subscribers to SNS topic.</p>	<p>IAM is vital for the services to be connected to each other and invoke each other as intended.</p>
 <p>DynamoDB streams</p>	<p>In short, DynamoDB streams is a trigger on tables. Users can set which Lambda has to be triggered on any changes in the table. These changes create an event:</p> <ul style="list-style-type: none"> • INSERT event: when a new item is inserted in the table • MODIFY event: when an item in the table gets updated 	<p>Kinesis streams were an alternative, but the service is more costly in comparison to DynamoDB streams, and streams only consumes Lambda functions.</p> <p>DynamoDB streams ensures that the system fulfills on its</p>

	<ul style="list-style-type: none"> REMOVE event: when an item in the table gets removed 	<p>promise – being an event-driven system.</p> <p>Most of the Lambda functions are triggered using DynamoDB streams, which will be made clearer later on in the report.</p>
--	--	---

Table 1 Technologies used

3. Methodology

This section discusses all of the processes in the development of Takhatta. First, the requirements are listed along with some user stories to make the requirements clear. Then the design of the system will be discussed, followed by the implementation process and finally the testing phase.

3.1. Requirements

3.1.1. Requirements elicitation

Method	Description	Date
Client brief	The client brief is a short one-page document provided by BAC to describe the problem that they are facing, the extent of the problem and other issues that branch out from the main problem. The brief was provided to the team in the first week, and the team held three meetings to utterly understand the document and create a document listing the requirements as understood from the brief.	10/10/2022
Client interview	The requirements document was presented to the BAC team, and we made sure that they approved all of the requirements before moving on to the implementation.	12/10/2022
AWS workshops	AWS held multiple workshops in two weeks night to discuss the requirements after their approval from the client, discuss how these requirements can be met and start planning for the design of the system.	16/10/2022 – 27/10/2022

Table 2 requirement gathering methods

This approach to gathering requirement is more effective than surveys or even interviews for a variety of reasons. The main reason is that our team had to truly understand the problem and how to fully mitigate it from the brief provided by the client, giving us a better understanding of the issue and more freedom in envisioning what the solution would be.

3.1.2. Functional requirements

Management Dashboard:

- The dashboard should display predictions of the number of counters and staff needed per hour for the upcoming month.
- The system should allow users to filter graphs on the dashboard according to zones.
- The dashboard should display the historical average wait time and number of staff over time.
- The dashboard should display the number of expected flights and passengers per day for the current month.
- The dashboard should provide the user with the ability to filter the data based on date, time, and flights.
- The system should notify the manager when any changes in the needed staff prediction occur.
- The system should display the count of tickets ahead for each passenger
- The dashboard should display the number of currently open counters.
- The dashboard should display the passenger's feedback log.
- The system should authenticate users by email and password.

Customer Satisfaction Subsystem:

- The system should display a survey on a screen in the airport for passengers to leave their feedback on.
- The system should measure the average satisfaction rate of passengers to be displayed on the management dashboard.

Virtual Queueing Subsystem:

- The system should enable passengers to click a button on a display to add themselves into the virtual queue
- The system should display the passenger ticket numbers that are in line for the check-in staff members.
- The system should allow check-in staff members to click a button to get the next passenger in line.
- The system should be able to announce the next passenger number to be checked in.
- The system should measure the actual wait-time of each passenger
- The system should calculate the average wait time and display it on the management dashboard.

Passenger Dashboard:

- The system should allow passengers to input their flight ID into a dashboard to get details about their flights (Check-in zone ID, gate number, gate state, boarding state).
- The system should allow passengers to view current estimated check-in wait time based on current staffed counters and queue lengths so travelers can come to the airport earlier.
- The system should notify passengers on any changes in gate number, check-in zone number or flight details.

3.1.3. Non-functional requirements

Actions:

- The system should automatically switch to a reliable backup system when a failure occurs.
- The system's infrastructure should be able to scale to support the connection load.

Management Dashboard (main dashboard):

- The dashboard's interface should be easy to use and user-friendly.

- The system shall provide a responsive dashboard that can be accessed by desktops and mobile devices.

Users:

- The system should apply a strong password policy and encrypt the user's passwords.

3.1.4. User stories

Functional User stories:

As a [Role]	I want to [goal]	So that [benefit]
Staff manager	Know the number of staff needed for the upcoming week	I know how many counters to open to minimize passenger queue times and eliminate manual calculations for staff allocation to shifts.
Staff manager	Have user authentication before having access to the dashboard	We can ensure security and privacy of airport data
Staff manager	Have myself notified upon any prediction changes	We can manage and update the schedule for the employees' shifts.
Staff manager	Filter the predictions based on date, time, and flights.	I can easily find the details of a specific time.
Passenger	Share my experience with the customer experience team	They can improve their services
Passenger	View my queue time, flight details, gate status, and get notified when changes occur.	I can manage my time and be aware of any last-minute changes
Staff manager	Filter the predictions based on date, time, and zone.	I can easily find the details of a specific time and zone.

Table 3 functional user stories

Non-functional user stories:

As a [Role]	I want to [goal]	So that [benefit]
Staff manager	Have accurate prediction with a minimum of 80%	To enhance customer satisfaction by allocating staff accurately
Staff manager	Have a failover system	The system is available 24/7 even in case of downtime
Staff manager	Have access to the system at any time	I can open the system whenever I need
Staff manager	I have access to the system from my desktop, tablet, and mobile.	I am always updated on the number of needed staff/counters without any restrictions
Staff manager	The system to be user-friendly, simple, and easy to use.	I can easily navigate through different services
Staff manager	Apply strong password policy and encrypt the passwords of the users	It meets the industry security requirements

<i>Staff manager</i>	The system to be scalable	It can handle many simultaneous connections
----------------------	---------------------------	---

Table 4 non-functional user stories

3.1.5. Constraints

The solution is being developed as a prototype for BAC, and not a production level solution. There are no constraints in the scope provided by the client.

3.2. *Solution design*

After the requirements have been gathered, approved by the client and finalized by the AWS team, we moved to designing the system before starting in the implementation phase.

3.2.1. Data structures

As mentioned earlier in the report, DynamoDB is a NoSQL database. DynamoDB is a key-value database, where instead of having a primary key and a foreign key, the database uses individual keys or combinations of keys to retrieve values (key-value pairs). This creates a messy schema, which is one of the disadvantages of NoSQL databases, and so creating an entity relationship diagram is not an option in this case (“Key-Value Database: How It Works, Key Features, Advantages,” 2022).

S3 also follows the same data structure for storing the data.

NoSQL is common in AWS services due to its scalability and high speeds in reading and writing data, increasing the performance which is what AWS promises their users.

3.2.2. Design diagrams

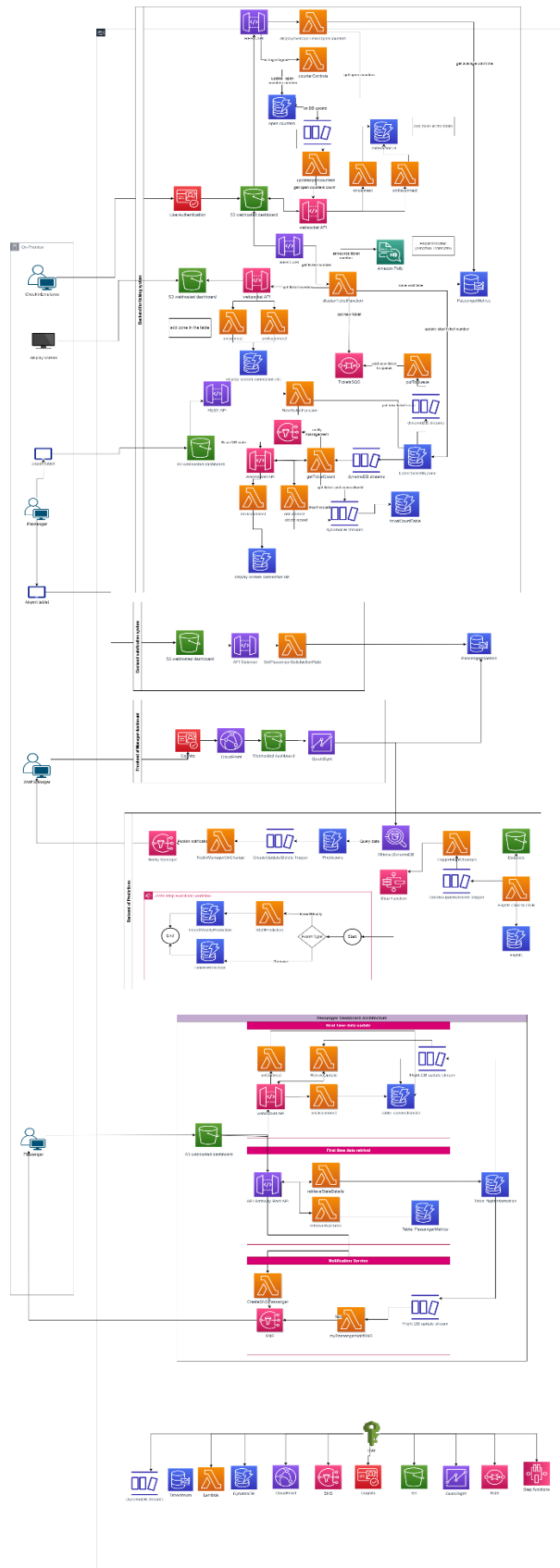
Structural diagram:

Structural diagrams represent the static relations of elements of a system (“UML - Behavioral Diagram vs Structural Diagram,” 2022). Since the Takhatta system comprises of multiple subsystems and each subsystem has a unique combination of services, a system architecture has been designed to outline these services and their relations, as well as designing the relations between the systems, i.e., tying the check-in employee dashboard to the virtual queueing system, representing which databases they interact with and how this relation fits in with the larger picture of the system.

Creating an AWS system architecture diagram is one of the first steps in the creation and deployment of new systems on the cloud. The main purposes of designing a system architecture diagram (“How to Draw AWS Architecture Diagrams,” 2019):

- Designing for cost optimization
- Recognition of single points of error
- Proof of concepts to clients and future developers
- Faster troubleshooting

Overall architecture of the Takhatta system:



The architecture represents all of the services for the final implementation of the system (covered in the project technology section). The purpose of the architecture is to represent how the subsystems work together in achieving the success of the system. This is important in designing cloud-based applications to finalize the services that will be used, estimate costs as well as ensuring that the solution is being developed efficiently. The architecture shows all of the subsystems and the services used in each subsystem.

Figure 1 Takhatta system architecture

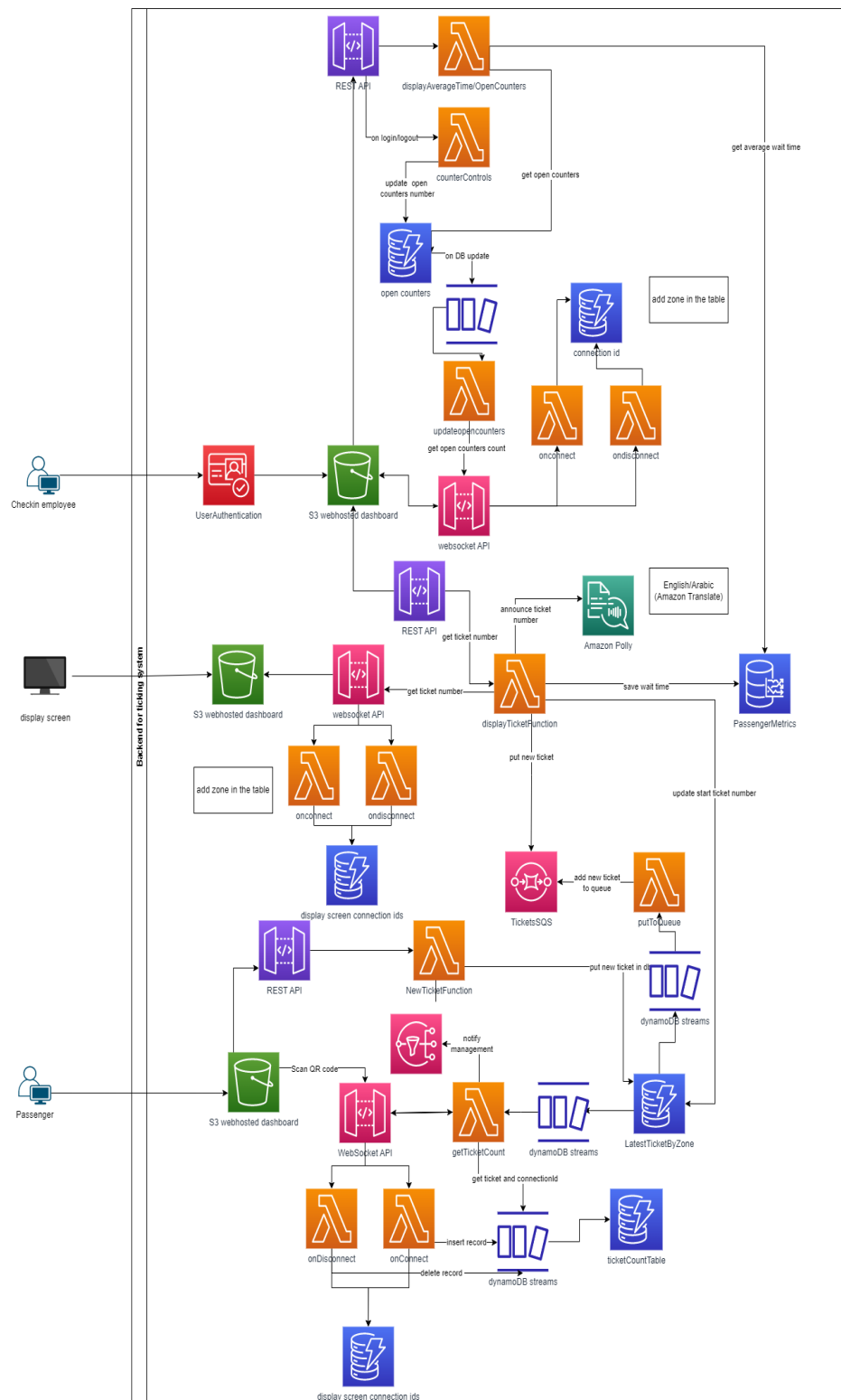


Figure 2 ticketing subsystem architecture

More detailed view of the virtual queueing system. The virtual queueing system directly ties in with the check-in employee dashboard, and both subsystems get from and store data to the same databases.

Prototype of ticketing subsystem:
On-premises tablet: Home page

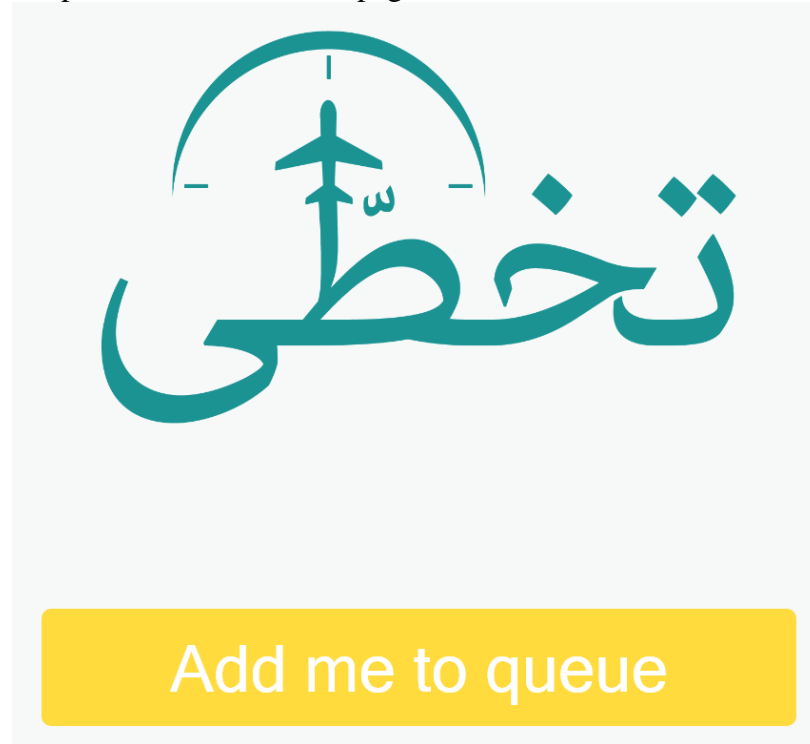


Figure 3 On-premises tablet: Home page

The virtual queueing system for the airport will be accessed by people with different technological backgrounds and of different ages. For that reason, the user interface was made to be very straight forward and simple, with minimal human intervention required. The only interaction that passengers make with the system is when they click the button to be added to queue and scanning the QR code in the next page.

On-premises tablet: ticket page



435

Ticket Number



Scan to Save

Done

Figure 4 On-premises tablet: ticket page

The ticket page displays the ticket number to the passenger, as well as a unique, automatically generated QR code for each ticket number. The page has a 30 second timer (in the final implementation) to redirect the tablet back to the human page. This is done to avoid conflict in ticket numbers between passengers. The QR code uses qrcode.js library to generate the QR code.

User device: ticket display

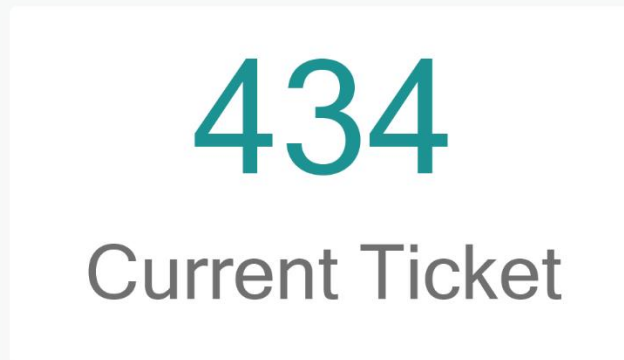


Figure 5 User device: ticket display

The ticket display page, which is what the user gets when they scan their QR code, adds the user to the virtual queue and saves their ticket number in their device. This page calls a REST API that invokes a lambda function. The lambda function gets the ticket from the database, increments it, and sends it to the frontend. In the final implementation, and as a part of the project's scope, a field displaying the count of the tickets ahead of the user is displayed as well. This field calls a WebSocket API to return data in real time, and each ticket number gets a count relevant to their ticket number only.

Behavioral diagrams:

The main purpose for designing behavioral diagrams is to represent how the system behaves and interacts within itself ("UML - Behavioral Diagram vs Structural Diagram," 2022), and with other external entities (end users). Behavioral diagrams show the flow of data, decisions that are being made within the system, where data is being retrieved from and what triggers the retrieval of said data.

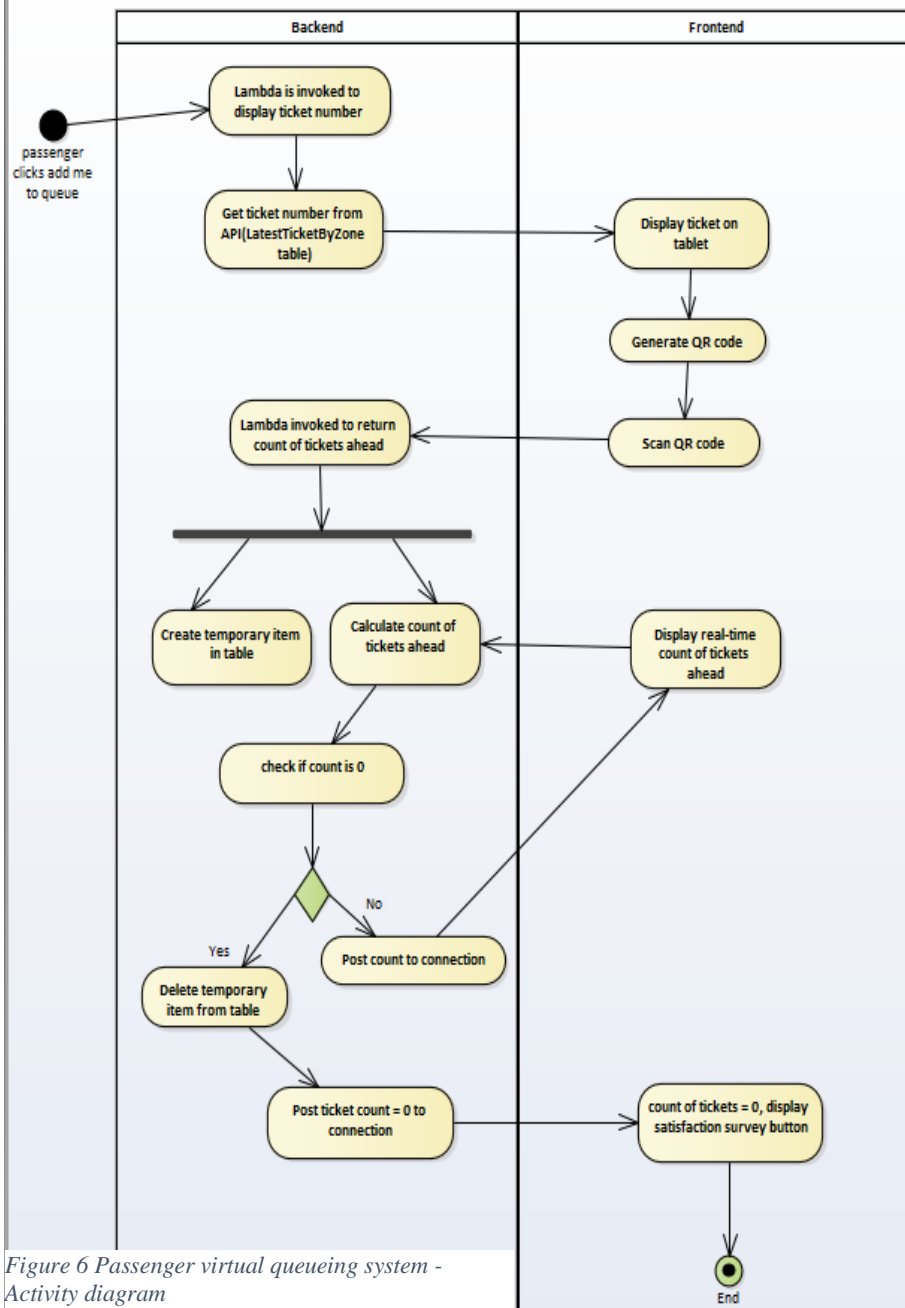
Some of the types of behavioral diagrams:

- Activity diagram: activity diagrams could be described as more advanced flowcharts ("What Is Activity Diagram?," 2022). Activity diagrams show the process that the system follows to achieve its goal, the decisions that led to the goal as well as the logic followed in the process.
- Sequence diagram: sequence diagrams fall under interaction diagrams, which is a subset of UML behavioral diagrams. Sequence diagrams detail how operations are carried out by capturing interaction between the user and the system, as well as one component of the

system with another component within the system (“What Is Sequence Diagram?,” 2022). Sequence diagrams clearly show the flow of tasks as their order is shown clearly and can be understood with ease.

- Use-case diagram: a use-case diagram is a visual representation of what the system is designed to do, the different actors (users) of the system, what they will be interacting with and what those interactions can include or extend. A key concept of creating a use case diagram is designing the system from the end-user's perspective ("What Is Use Case Diagram?," 2022).

Activity diagram of the Takhatta passenger virtual queueing system:



The activity diagram shows that the event starts when the passenger clicks the add me to queue button, where the lambda function is immediately triggered to get a new ticket from the backend database by being invoked by the REST API. The retrieved ticket is then displayed in the frontend (tablet) and a QR code is generated that the user is meant to scan. Once the QR code is scanned, another lambda is invoked by the WebSocket API. This activity action goes to a fork – create temporary item for user in database and calculate count of tickets ahead for the same user. The lambda proceeds to check if the ticket count is 0. If the answer is No, the lambda posts the count of tickets ahead in the frontend (user's device) and goes back to calculate the count of tickets ahead again. Once the count of tickets ahead hits 0, the lambda posts 0 to the user and they are presented with a button which redirects to the satisfaction survey.

Figure 6 Passenger virtual queueing system - Activity diagram

Use-case:

The use case diagram of the Takhatta virtual queueing system can be found under appendix 2: detailed design document.

Sequence diagram of the Takhatta virtual queueing system:

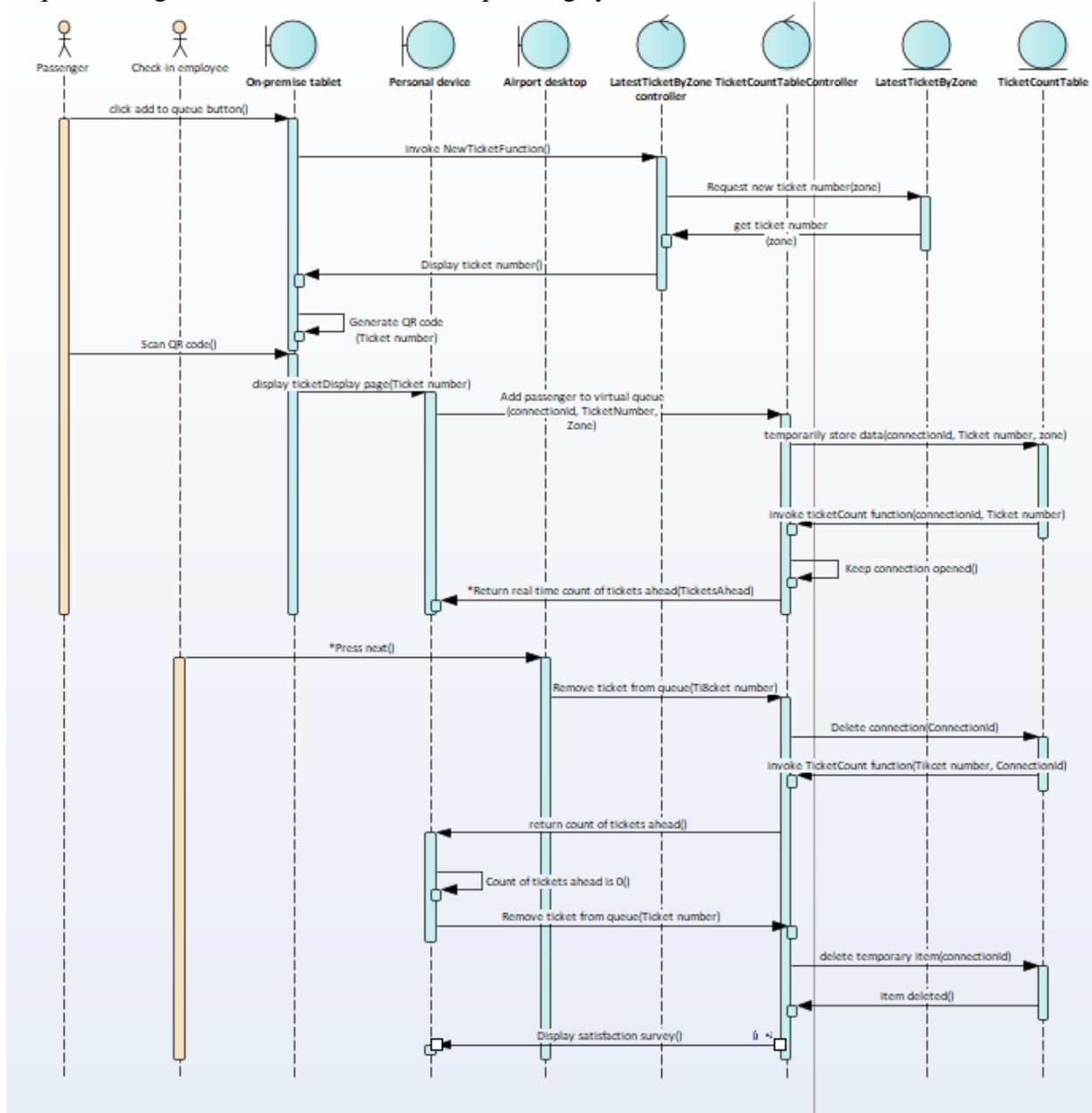


Figure 7 Virtual queueing system - sequence diagram

3.3.Implementation

The implementation of the passenger virtual queueing system consists of many different services as shown in the system architecture in the design section of the report. Below is a step-by-step guide on the implementation phase of the subsystem.

3.3.1. Description by components

First step: create S3 bucket and host webpages

A S3 bucket was created to store the HTML files, CSS stylesheet as well as the images needed for the system. Static website hosting was enabled after the bucket was created to generate a URL for the bucket for customers to be able to access content at the website endpoint, and a file named index.html was configured as the index document.

Static website hosting

Use this bucket to host a website or redirect requests. [Learn more](#)

Static website hosting

- ☐ Disable
☒ Enable

Hosting type

- ☒ Host a static website
Use the bucket endpoint as the web address. [Learn more](#)
☐ Redirect requests for an object
Redirect requests to another bucket or domain. [Learn more](#)

i For your customers to access content at the website endpoint, you must make all your content publicly readable. To do so, you can edit the S3 Block Public Access settings for the bucket. For more information, see [Using Amazon S3 Block Public Access](#)

Index document

Specify the home or default page of the website.

index.html

Figure 8 hosting a static website on S3

Next in this step, the frontend of the system was created. The virtual queueing system consists of three webpages,

- Index.html: a simple page, displaying the logo of Takhatta and a button to add user to queue. There have been no changes to the index.html from the prototype.
- MainPage.html: the main page displays the ticket number to the user and a QR code unique to that ticket number.
- TicketDisplay.html: the page that the user is redirected to when they scan the QR code on their personal device. It displays the same ticket number, and a count of the tickets ahead of the user.

MainPage.html:

The ticket number is posted to the user using a REST API which invokes the newTicketFunction lambda function, which will be discussed later on.

Changes in MainPage.html from prototype:

- 30 second timer that automatically redirects to index.html when the timer is zero.
- Dynamically generated QR code for each ticket number, giving each ticket a unique URL. Both of these functionalities have been done in the frontend in JavaScript. The QR code is generated using qrcode.js library, passing the URL of the static webpage and using the ticket number as a URL parameter to display on the user's device.

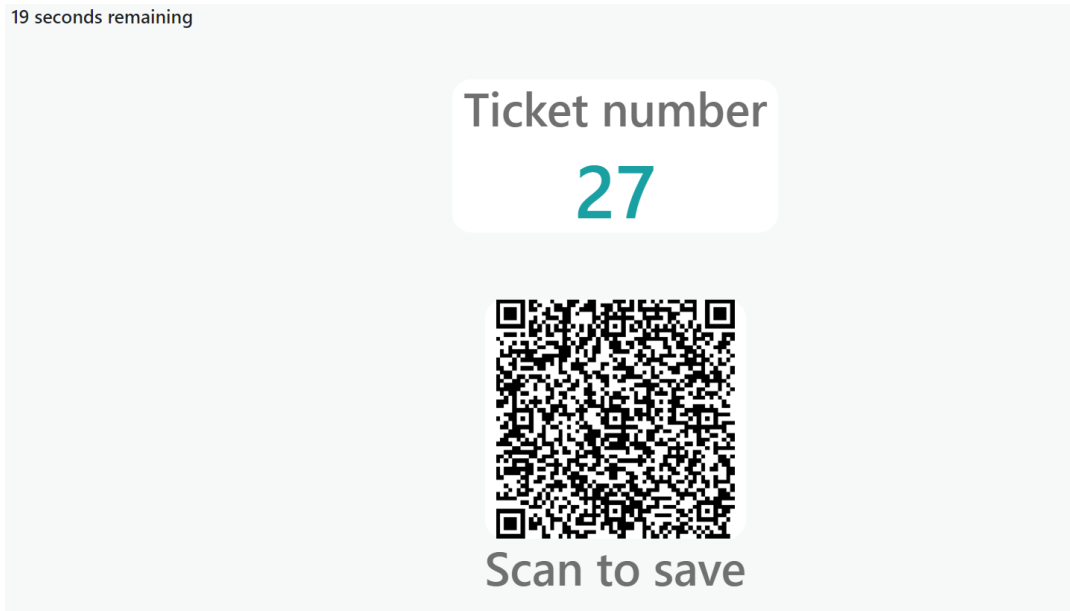


Figure 9 MainPage.html final implementation

```
var timeLeft = 30;
var elem = document.getElementById('countdTimer');

var timerId = setInterval(countdown, 1000);

function countdown() {
  if (timeLeft == -1) {
    clearTimeout(timerId);
    window.location.replace("http://takhatta-ticketingdashboard.s3-website-us-east-1.amazonaws.com")
  } else {
    elem.innerHTML = timeLeft + ' seconds remaining';
    timeLeft--;
  }
}
```

Figure 10 JavaScript timer code

Changes made in the TicketDisplay.html page from the prototype:

- Displays a real time count of tickets ahead, unique to each ticket number
 - Displays a button to redirect to the satisfaction survey when the count of tickets ahead is zero.
- The count of tickets ahead is returned using a WebSocket API which invokes the ticketCount Lambda function, with a 10-minute idle timeout duration. The button is displayed using JavaScript in the frontend.

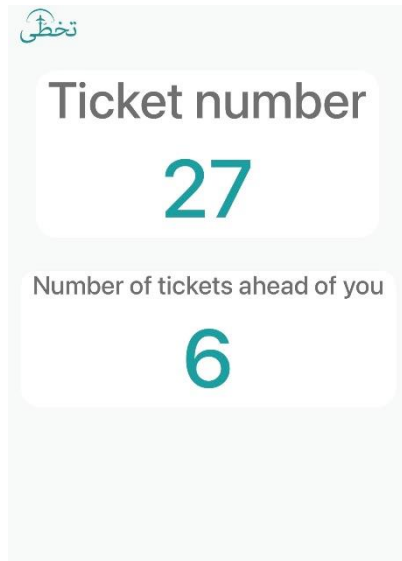


Figure 12 TicketDisplay.html - button hidden

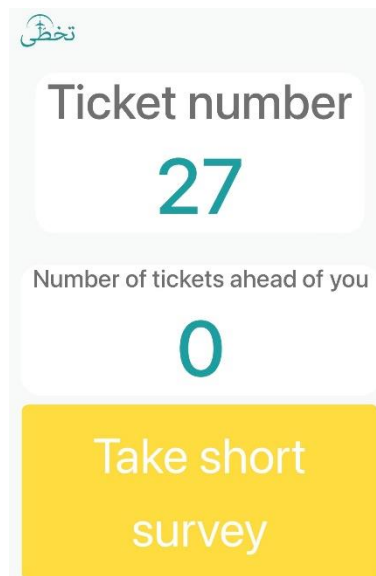


Figure 11 TicketDisplay.html - button displayed

The button is hidden by default and displayed when the count of tickets ahead is 0 by modifying the CSS styling in the script.

```
if(data == 0){
    document.getElementById("Btnredirect").style.display = "block";
}
else{
    document.getElementById("Btnredirect").style.display = "none";
}
```

Figure 13 code snippet - change button display

When the user clicks the “take short survey” button, they are redirected to the satisfaction survey which consists of one question and after they take the survey, they are redirected to the passenger dashboard. This is unique to the virtual queueing system, as the on-premises tablet that displays the satisfaction survey does not redirect to the passenger dashboard. This was achieved by modifying the code of one of the other team members and passing a URL parameter which will check if the satisfaction survey was opened from the virtual queue or on the tablet.



Figure 14 satisfaction survey

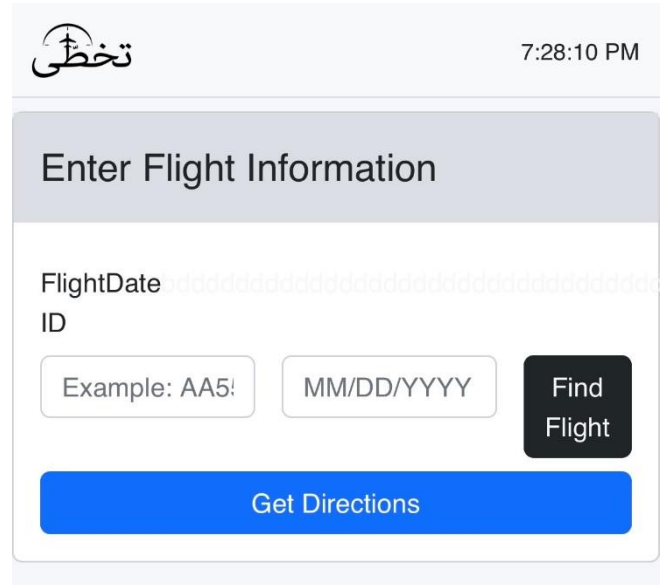


Figure 15 passenger dashboard

Second step: Create IAM roles

Create IAM roles for Lambda functions. IAM roles are roles that have a collection of policies that control the permissions of the Lambda functions, what they can or cannot access from the other services. This adds a security layer to the system, as if there was any breach the hackers will have limited access to resources because the IAM roles have been configured in a way that gives the Lambda functions least privilege.

By default, upon creation, Lambda functions have the AWSLambdaBasicExecution role, which does not enable the lambda function to communicate with other AWS services.

Below is a screenshot of the IAM role for the TicketCount Lambda, which enables it to communicate with API Gateway, SNS and DynamoDB.

AWS cloud developers have the freedom to attach and remove policies as they please, as well as creating custom inline policies.

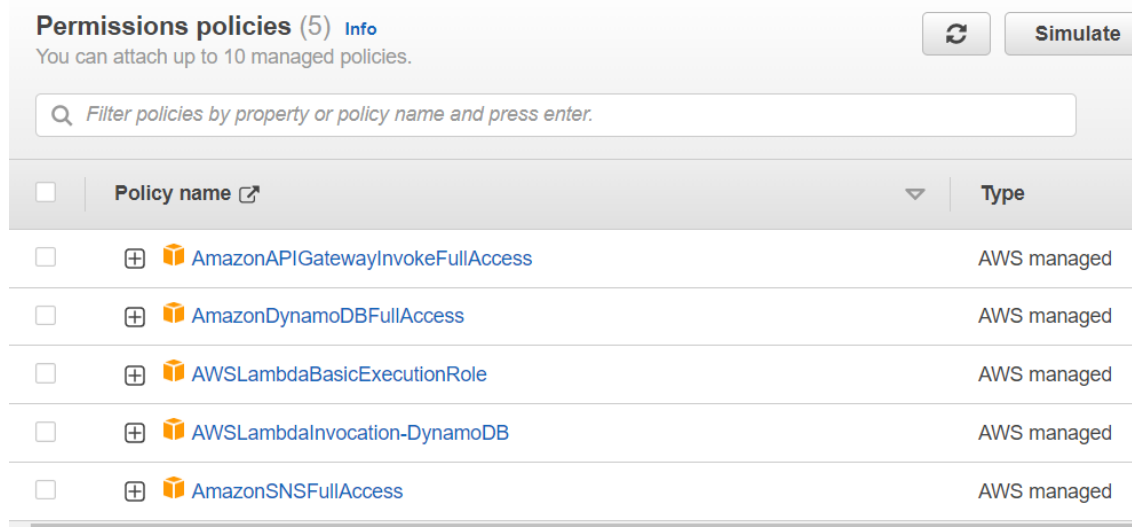


Figure 16 IAM role - TicketCount function

Third step: Create DynamoDB tables

The passenger virtual queueing system gets data, and sends data to two separate tables,

- LatestTicketByZone: the partition key of this table is Zone (String). Read and write capacity for this table has been configured to provisioned with auto scaling. Each item in this table consists of three attributes:
 - Island: the island in the airport which the ticket is in, values can be A, B, C etc.
 - LastTicket: the ticket number that will be displayed to the next passenger that hits the add me to queue button.
 - Ticket_NowServing: the ticket number that is currently being served at the counter, in the zone. This attribute is necessary for the TicketCount function to return the count of tickets ahead.
- ticketCountTable: this table has been created to handle simultaneous connections to the WebSocket API. Records are entered in the TicketCount function and removed in the onDisconnect function (which has been modified to do so). userConnection attribute is the partition key of this table.

DynamoDB streams has been enabled on both tables to deliver on the promise of create a system with an event-driven architecture.

DynamoDB stream details
Capture item-level changes in your table, and push the changes to a DynamoDB stream. You then can access the change information through the DynamoDB Streams API.

Disable


Stream status

✔ Enabled

View type

Old image

Latest stream ARN


 arn:aws:dynamodb:us-east-1:505643665857:table/LatestTicketByZone/stream/2022-12-02T16:04:18.166

▼ Triggers (2)

Use triggers to invoke an AWS Lambda function every time an item is changed, and then your DynamoDB stream is updated.

Trigger (2)

↻

Configure 


Delete

Create trigger

<

1

>




	Function 	State	Last processing result
<input type="radio"/>	TicketCount	Enabled	OK
<input type="radio"/>	putToQueue	Enabled	OK

Figure 17 DynamoDB streams - LatestTicketByZone table

Fourth step: Create Lambda functions

The passenger virtual queueing system consists of three lambda functions (not including onConnect and onDisconnect functions of WebSocket API). The runtime of the functions is python (boto3 Software Development Kit).

- NewTicketFunction:
 - Triggered on REST API call when passenger clicks “add me to queue” button
 - Gets value of LastTicket from LatestTicketByZone table
 - Checks if the ticket number is less than 500, if it is equal to 501 returns 1 to table, otherwise increments ticket number by 1 and returns old value to API

- Updates LastTicket with the new value

```

if (TicketCount>499):
    #recentTicket[0]['LastTicket'] = 0
    updateTable = table.update_item(
        Key={
            'Zone': 'A'
        },
        UpdateExpression="set LastTicket = :val",
        ExpressionAttributeValues={
            ':val': decimal.Decimal(1)
        },
        ReturnValues="UPDATED_NEW"
    )
else :
    updateTable = table.update_item(
        Key={
            'Zone': 'A'
        },
        UpdateExpression="set LastTicket = LastTicket + :val",
        ExpressionAttributeValues={
            ':val': decimal.Decimal(1)
        },
        ReturnValues="UPDATED_NEW"
    )

```

Figure 18 NewTicketFunction - check ticket number

- PutToQueue:
 - Triggered by MODIFY event on LatestTicketByZone
 - Adds the ticket number to SQS queue, depending on the zone of the ticket.
- TicketCount:
 - Gets value of LastTicket, Ticket_NowServing
 - Puts new item in ticketCountTable if event type is INSERT

```

# check if event is insert (new ticket)
event_type = event['Records'][0]['eventName']
if (event_type=='INSERT'):
    user = event['Records'][0]['dynamodb']['NewImage']
    ticketnum = user['userTicket']['S']
    zone = user['userZone']['S']
    conId = user['connectionId']['S']
    record = {
        "userTicket": {"S" : str(ticketnum)},
        "userZone": {"S" : str(zone)},
        "connectionId": {"S" : str(conId)}
    }
    response = ticketCountTable.put_item(
        Item={
            'userConnection': conId,
            'userTicketNum': ticketnum,
            'userIsland': zone
        }
    )

```

Figure 19 TicketCount function - check event type, put new record

- Returns real-time count of tickets ahead to user. Since the queue auto-resets at ticket number 500, the TicketCount function checks if the value of Ticket_NowServing is greater than the value of LastTicket, to determine which way the queue is going.

- If the user is next in line, deletes temporary item in TicketCountTable

```
#do calculation and send response to specific id AND ticket
if(item["userConnection"]["S"] == userConn and int(item["userTicketNum"]["S"]) == userTicket):
    if(nowServingTicket>LastTicket):
        ticketsAhead = 500-nowServingTicket+userTicket-1
        result = API.post_to_connection(ConnectionId=item["userConnection"]["S"],Data=json.dumps(int(ticketsAhead)))
    elif(nowServingTicket<LastTicket):
        ticketsAhead = userTicket-nowServingTicket
        result = API.post_to_connection(ConnectionId=item["userConnection"]["S"],Data=json.dumps(int(ticketsAhead)))
    #if there are no tickets ahead, delete table item
    else:
        result = API.post_to_connection(ConnectionId=item["userConnection"]["S"],Data=json.dumps(int(0)))
        response = ticketCountTable.delete_item(
            Key={
                'userConnection': userConn
            }
        )
    )
```

Figure 20 TicketCount function - post to connection

- Lastly, the TicketCount function checks if there is high passenger traffic depending on the count of tickets ahead returned to the user. The function calculates the total wait minute for EACH counter, and if the counters will take more than 25 minutes to remove all of the passengers currently in queue, an email is sent to management using SNS. If management is notified, the function changes the value of “Notified” (separate database) to Yes, to avoid spamming management.

```
#calculation of wait time for all waiting tickets
# 3 is check-in time
waitMinutes = (ticketsAhead*3)/counters
#notify management
if(waitMinutes>25 and managementNotified=="No"):
    response = sns.publish(
        TopicArn='arn:aws:sns:us-east-1:505643665857:OpenCounterNotification',
        Message='Traffic is too high in island A. Consider opening more counters.',
        Subject='Urgent: Action required in island A.',
        MessageStructure='string',
        MessageAttributes={
            'Message': {
                'DataType': 'String',
                'StringValue': 'Urgent'
            }
        }
    )
```

Figure 21 TicketCount function - check if management has to be notified

Urgent: Action required in island A. Inbox x



AWS Notifications <no-reply@sns.amazonaws.com>

to me ▼

Traffic is too high in island A. Consider opening more counters.

Figure 22 sample of email notification

- The onConnect and onDisconnect functions have been modified as follows:
 - onConnect: store user ticket number and zone along with connectionID
 - onDisconnect: delete temporary item from TicketCountTable

Fourth step: SQS and SNS

SQS and SNS are straightforward to create and configure.

For SQS, FIFO queues have been used, queues have been created for each zone to handle them and the ARN is used in the Lambda to specify which queue a ticket is placed in.

For SNS, a topic has been created with the email addresses of management to send them the automated notifications, and the notifications are triggered using the topic ARN.

The message is written inside the lambda function.

Testing

During the implementation of the Takhatta system passenger virtual queue, we had two meetings each week to show our progress and get it approved by the AWS consultants (more details in appendix 1).

This led to some changes in the design of the system to work around some limitations and restrictions.

This section discusses the two test types that have been carried out when the final system was developed:

- functionality test: functionality testing is testing the final result against the requirements that have been gathered and ensuring that the system achieves its objectives. It is preferred that the participants of a functionality test have an experience in development so they can run specific test cases (Bose, 2021).
- acceptance test: acceptance testing is testing if the system achieves the business objectives, and since the virtual queueing system will be used by passengers of all ages and technological backgrounds, participants have been picked carefully to simulate how a normal person would interact with the system.

3.3.2. Test plan

Strategy:

For both functionality test and acceptance testing, the participants have not been informed on what the objectives or requirements of the system are. They were presented with the system and had the freedom to navigate it and test it for themselves.

Later on, since the participants of the functionality test are more technical people, some of the core concepts of the system have been explained to them for more specific test cases that they come up with. This proved very beneficial to the developers as some changes had to be made in the system, to avoid some of the bugs that were discovered during testing.

Test schedule:

After finding the participants and aligning with each of their schedules, the following test schedule has been created, with one hour of one day has been dedicated to one test participant:

Test type	Dates
Functionality testing	20/12/2022 – 22/12/2022
Acceptance testing	25/12/2022 – 27/12/2022

3.3.3. Functionality test cases and results

Since the functionality testing that has been carried out deals exclusively with the passenger queueing system, the following steps have been followed (“What Is Functional Testing? Types & Examples,” 2020):

- Determining which functionality has to be tested. This includes:
 - Providing sequential, unique ticket numbers
 - Generating appropriate QR codes for each ticket

- WebSocket API handling simultaneous connections
- Idle connection testing
- Testing freely, to find any unanticipated bugs
- Modifying ticket number in the database to fit each test case
- Determining the output of each test case prior to carrying out test
- Comparing actual test results with predicted results

Participants of functionality test case:

Participant	Motivation
Mahmood Abbas (P1)	Mahmood is a senior software engineer at Citi Bahrain. He took various courses to expand his knowledge in the technology field and tries to keep up with the modern technologies that are rolled out regularly
Ali Jaffer (P2)	Ali is a graduate Artificial Intelligence student, who spent the last three years abroad pursuing his career. Ali has an immense technological background but has never used the AWS console.
Ali Waleed (P3)	Ali Waleed is currently working in the technology department of NBB. He has almost three years of experience in the field making him a great candidate for testing the system.

Table 5 functionality test – participants

The table below highlights the different test cases carried with all three participants, the actual outcome and their feedback.

Test case	Expected outcome	Actual outcome	Feedback	Pass/fail
The system provides sequential, unique ticket numbers to each passenger	The system increments each ticket number for the next ticket, resetting to 1 after ticket 500	Tickets are displayed sequentially, without any issues	P1 noted that the display of the ticket number and the QR code generation takes a couple of seconds, but the delay isn't long enough to cause any issues when the system is deployed.	Pass
The system generates a QR code unique to each ticket	The URL parameters of the QR code are consistent with the ticket displayed on the screen	URL parameters were consistent with ticket number displayed for each participant.	P3 had an issue with the ticket number being passed in the URL parameters, stating "What if the passenger changes the URL parameter? How would that affect the system?." He was instructed to do that in the last test case.	Pass
WebSocket API should be able to handle simultaneous connections	The count of tickets ahead should be displayed for each ticket, depending on their ticket	The count of tickets ahead is displayed as intended.	P3 wanted to test two different ticket numbers, ticket number 5 and ticket number 496, since he was told that the ticket counter	Pass

	number and the ticket being served currently.		resets at 500 and wanted to test the count of tickets ahead. This has already been considered, and the backend function checks to see which way the queue is going.	
Free testing	The participants should all be able to test the system with use cases they come up with on the spot, with no issues or bugs.	Participant defined test-case specific.	P3 tested changing the ticket number in the URL parameter, which caused the ticket number displayed on the screen to update, as well as the ticket count of tickets ahead. This does not cause any issues in the system since the objective of the system is to limit queue time to three minutes, and passengers wouldn't have time to fiddle with the URL. Added to future work.	Pass

Table 6 functionality test- matrix

As it is apparent from the table above, the system passed all of the functionality testing that has been carried out. However, a minor bug regarding the URL parameters has been uncovered by P3. This bug is a part of the future work, where the URL parameters can be encrypted and a REST API can be used to pass the secret key from the backend.

3.3.4. Acceptance test process and results

Since the system has been designed with the concept of least human intervention, each participant was asked to simulate a real-life process of getting in queue, where the system's index page was opened on a tablet and the participants used it as the developers intended. Participants were presented with the tablet, with no prior knowledge of what the system looks like or what is supposed to happen. They then each proceeded to use the virtual queueing system, and the results are as described below.

Acceptance testing participants:

Participant	Motivation
Marwa Jasim	Marwa is a Digital marketing student at Bahrain Polytechnic. She travels frequently and is familiar with the current queueing and check-in process at the airport.
Mariam Ali	Mariam works in the pharmacy at Al Salmaniya hospital and is familiar with their queueing system which is paper based.

Table 7 acceptance test – participants

Both of these participants have little to no technological background, making them perfect candidates to simulate airport customers (passengers).

Participant	Expected outcome	Actual outcome	Pass/Fail
-------------	------------------	----------------	-----------

Marwa Jasim	The participant is able to pick up the system, add themselves to queue and take the satisfaction survey with no issues.	The participant was able to use the system as intended.	Pass
Mariam Ali	The participant is able to pick up the system, add themselves to queue and take the satisfaction survey with no issues.	Mariam did not have her phone on her at the time of testing, so she was unable to scan the QR code initially due to the 30-second timer in the ticket display page. Once she realized this, she tried again by getting a second ticket and was able to use the system with no issues faced.	Pass

Table 8 acceptance test – results

The functionality test results and the acceptance test results show that the Takhatta passenger virtual queueing system passes all of the client requirements and achieves the success matrix set by BAC.

These tests have been carried out at the end of the implementation, and don't take into consideration the weekly meetings with AWS consultants.

Specifics regarding the weekly meetings are defined clearly in appendix 1.

4. Discussion

4.1. System functionality

The core functionality of the system is to make simulation-based predictions on the allocation of human resources by BAC based on historical data provided by the client. The passenger virtual queueing system's functionality ties in directly with the predictions system, where the average queue time is presented to management on their dashboard directly as well as on the check-in employee's dashboard. The virtual queueing subsystem records the actual wait time of each passenger in order to store actual data which updates in real time to the management. This enables the management to not only measure the success of the system but monitor their check-in employees' performance for performance reports and generate more revenue from airport facilities because passengers are not stuck in line to complete the check-in process. The tests carried out proves that the system works as intended both functionally and for the end-users of the virtual queueing system. Implementing Takhatta at the airport would not only increase the revenue but cut costs as well as the airport will not place too many employees on counters.

The success metrics that have been provided by the client but cannot be tested unless it's in a real-world deployment:

- Three-minute passenger queue time – too many variables in the real world, so this cannot be tested unless the system is implemented in the airport
- Increase customer satisfaction experience to 85% - Takhatta gives passengers the ability to send feedback to the customer experience team, but this cannot be tested as each passenger's experience will have other factors like check-in employee performance.

4.2. Summary of achieved objectives

This section summarizes the passenger virtual queueing objectives, briefing how each objective has been met:

- Places passenger in queue
 - This objective has been successfully achieved using the event-driven architecture that retrieves a ticket number for each passenger, returns it to them within seconds and gives passengers the ability to save their own tickets on their personal devices. The passenger is placed in the SQS queue upon scanning the ticket number, placing them in queue and their ticket number is displayed on the on-premises screen along with the ticket being served currently.
- Enables management to track employee performance
 - Management is able to track their employees' collective performance depending on how many notifications they get per month, the average queue time of passengers for a day/week. This can also motivate employees increasing their performance, since the check-in employees have access to the average wait time and have a goal to achieve (three minutes).
- Gives precise statistics of actual passenger queue time
 - Upon entering/exiting queue, timestamps are recorded and stored in a database which displays the precise queue time for each passenger. The main reason of implementing this for the client is to measure the success of the predictions system.
- Enables passengers to enjoy airport facilities instead of waiting in line
 - This objective is achieved by displaying the real-time count of tickets ahead to each passenger, giving them the ability to roam around freely in the airport while they wait for their turn, without stressing about missing it.
- Notifies management on high passenger traffic
 - In urgent cases of high passenger traffic, the system sends an email to notify management in real time in order for them to take immediate action.

4.3. Project issues

There were several challenges that were faced in the implementation of the system, which is expected in the process of the development of a large-scale system with vast amounts of processing.

Challenges faced in the development of the passenger virtual queueing system:

- Generate QR code

Initially, QR code generation was supposed to be done in the backend in the lambda function. Lambda Layers was created to import the QRCode library, uuid was used to generate random scripts that would be used in the URL and encryption was being done in the backend. Lambda layers is a service provided by AWS enabling developers to download custom libraries, zip them using CloudWatch CLI and attach them to a lambda which would then be able to import and use the library. This approach caused a lot of delay in the development of the system. This approach works as follows:

 - Generate a random string using uuid
 - Generate a QR code for the ticket number using the generated uuid in the URL
 - Store the generated QR code image in a S3 bucket
 - Retrieve the QR code image in the frontend
 - Delete the QR code from the S3 bucket

There were three key issues with this approach:

- Performance: the process of generating the QR code in the backend, storing it in a bucket and retrieving it in the frontend immensely decreased the system's performance due to unnecessary processing on every request.
- Cost: this approach not only has a higher cost due to the processing that needs to be done upon each request, but storing each image in an S3 bucket, even if it was temporary, would not be feasible and would significantly increase the running cost of the system in an airport that is open all day, every day of the week and gets thousands of customers every day.
- Retrieval: the QR code's URL is generated using a randomly generated string, making the retrieval of each QR code to the corresponding ticket number an impossible task.

As soon as this issue has been identified, the QR code is being generated in the JavaScript code in the frontend. This causes the ticket number to be passed in the URL parameters, but due to the limited time that the passenger would have with the system before they're next in line, this is not a prominent issue in the system.

- Simultaneous WebSocket connections/ sudden disconnection

When the WebSocket API was created and on initial testing, a single connection worked as intended but as soon as more than one user is connected to the system, two issues arise:

- Both connections get the same count of tickets ahead. This means that if one device has ticket number 5 and another device has ticket number 6 and the ticket currently being served is number 2, both devices would get the same count of tickets ahead (4), depending on the ticket number of the latest connection.
- If a connected device suddenly disconnects by leaving the page, closing the browser or losing internet connectivity, the entire system crashes and all the other devices will not get real-time updates of the tickets ahead of them in queue. This occurs because once a device disconnects, the system stores the value of tickets ahead as null in the database, which cannot be processed in the backend due to unexpected data type.

These issues with WebSocket API would ruin the passenger's experience in using the system and would make the system useless to them. The process carried out to mitigate these issues are as follows:

- Create a new table, TicketCount table, to temporarily store the connected user's connection id, ticket number and zone.
- Automatically add each new connection to the table by modifying the default onConnect WebSocket API function.
- Post the changes to each connection using a for loop in the backend.
- Delete the connected user's item in two cases – if the ticket count returned to them is 0 or if they disconnect before their count is 0.

Using this workaround mitigates the issues highlighted in this section.

4.4. LESPI

Below is a table of the potential Legal, Ethical, Social and professional issues of the Takhatta system and how they affect the system:

Issue classification	Issue description	Effect on Takhatta
Legal	Some of the historical data provided by the client contains sensitive data that is not available to the public. The AWS consultant sat the team down and discussed this manner with them, and it was agreed that none of the data would be leaked or shared with anyone.	This does not negatively affect Takhatta, it actually shows how much trust BAC are putting in the team which motivated and inspired the team even more to deliver the system.
Ethical	Managerial decisions have to be fair between staff. Managers have to fairly schedule employee's working hours by using the predictions but ensuring that they do not overwork a selected group of overperforming employees to stick to the predictions that have been made by the system.	Underperforming employees may feel like they don't have a fair chance to prove themselves to the management due to the new system.
Social	Since the predictions system makes daily predictions for a month in advance, this will inevitably lead to management creating employee schedule very early on – leading to social issues like when the employee plans on taking a leave or urgent, unpredictable situations that may occur after the employee schedule has been created.	Management has to use Takhatta to its full potential to benefit the airport, but the decisions have to be flexible and not too strict, otherwise employees will feel like Takhatta detracts them rather than benefitting them.
Professional	Although the system enables management to track check-in employee performance by monitoring the passengers' queue times, management should not completely rely on the system to assess their employees' performance for the following reasons: <ul style="list-style-type: none"> Unpredictable factors (number of passengers per ticket, number of departing flights) The wait times are displayed for an entire zone – not a specific counter 	This may cause employees to despise the system if management relies on it solely for their assessment. Managers have to agree with employees that the system would affect their performance report but wouldn't be the only method of assessing them.

Table 9 LESPI

4.5. Future work

Takhatta is a large-scale system that involves itself in many operations in the airport. This makes it so the overall system has high potential to be involved in even more operations, such as:

- Automatically creating employee schedule: the system could be worked on further to automatically create monthly schedules for employees using algorithms. This is not far-fetched from the current system as it already makes predictions, takes in many parameters that could be included in the algorithm and displays a real-time count of the counters opened. This could be achieved by entering each individual employee's details, BAC shifts according to their policy and flight details.

The virtual queueing system could introduce new features such as:

- Integrating a luggage self-check in system: the virtual queueing system could integrate functionality to enable passengers in queue to check in their luggage using their ticket

number, which would then be presented to the check-in employee who would be able to verify it and process it.

- Reserving a ticket without being physically present at the airport: the virtual queueing system could be upgraded to be its own standalone system, accessible from anywhere at any time for employees to schedule their arrival time, and the system makes prediction on the ticket number at the time of arrival of the passenger and reserve the ticket number for them.

Virtual queueing system improvements:

- Encrypted URL parameters: encryption of the URL parameters isn't a necessity for the current state of the virtual queueing system, but it would be vital if the new functions mentioned earlier were to be implemented.
- Increase idle timeout duration: currently, the idle timeout duration for AWS WebSocket APIs is 10 minutes by default and cannot *be changed* ("Extend WebSocket Idle Connection Timeout | AWS Re:Post," 2023).

4.6. Synopsis of experience

Developing Takhatta with AWS has undeniably been one of the most challenging and stressful experiences I have had as a student, but that does not mean that it was not beneficial to me or that I did not enjoy it. I am an individual that can only learn when under pressure or stress, and so being faced with so many challenges helped me in acquiring knowledge that I wouldn't have been able to acquire otherwise.

The most challenging aspects of working on the project were adapting to the AWS console, which I had no prior experience in as well as using python for the first time, and the boto3 SDK. The experience was a very steep learning curve between familiarizing myself with the services, learning python and effectively managing my time to submit the tasks on time and to the quality that is expected of me.

The AWS consultants and my university supervisor have been very supportive throughout the process and I was never left in the shadows when I needed help.

I can now confidently say that I have a moderate to high level knowledge about AWS services and cloud computing in general.

I had to manage my time efficiently, manage myself effectively and ensure that I am always communicating with the other team members which are all soft skills that are sought after in the industry, and I have improved them.

Working with AWS is an experience that I will carry with me for the rest of my professional career, as it came so early in my career and it has been very beneficial for me, it exposed me to the cloud, the benefits of using the cloud and building a system comprising of decoupled microservices.

Conclusion

The thesis briefly outlines the Takhatta system as a whole while giving a more detailed explanation of the virtual queueing system, its purpose and how it ties in within the whole system.

As it became apparent, the virtual queueing system is vital not only for BAC to track the success of the Takhatta system, but it also has a direct effect on the employees and the experience of the customer which ties back to the objectives of the system.

In the development process of the system, I have acquired a lot of knowledge regarding cloud computing, microservices and modern technologies that I had no prior experience in. I have never

worked with NoSQL databases prior to this project, and WebSocket APIs is a new concept to me which I plan on building on further in the near future as it will definitely come in handy to me in the future when I start working in the field.

The knowledge that has been gained can be furthered by taking AWS courses and getting cloud certifications, which every team member has expressed their willingness to do due to how pleasant and challenging working with the AWS console has been.

References

Amazon SQS queue types - Amazon Simple Queue Service. (2023). Retrieved January 8, 2023, from Amazon.com website:

<https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-queue-types.html>

Barney, N., & Gillis, A. S. (2022). Amazon Web Services (AWS). Retrieved January 8, 2023, from SearchAWS website: [https://www.techtarget.com/searchaws/definition/Amazon-](https://www.techtarget.com/searchaws/definition/Amazon-Web-Services)

[Web-Services](https://www.techtarget.com/searchaws/definition/Amazon-Web-Services)

Bose, S. (2021, May 11). Functional Testing: Definition, Types & Examples | BrowserStack.

Retrieved January 8, 2023, from BrowserStack website:

<https://www.browserstack.com/guide/functional-testing>

Difference between Rest API and Web Socket API - GeeksforGeeks. (2020, October 10).

Retrieved January 8, 2023, from GeeksforGeeks website:

<https://www.geeksforgeeks.org/difference-between-rest-api-and-web-socket-api/>

Event-Driven Architecture. (2023). Retrieved January 8, 2023, from Amazon Web Services, Inc.

website: <https://aws.amazon.com/event-driven-architecture/>

Extend websocket idle connection timeout | AWS re:Post. (2023). Retrieved January 8, 2023,

from Amazon Web Services, Inc. website:

<https://repost.aws/questions/QUFbNpkJJvTySYuHA7uHQAEQ/extend-websocket-idle-connection-timeout>

How to Draw AWS Architecture Diagrams. (2019, March 8). Retrieved January 8, 2023, from Lucidchart website: <https://www.lucidchart.com/blog/how-to-build-aws-architecture-diagrams#:~:text=Benefits%20of%20using%20AWS%20architecture%20diagrams&text=AWS%20architecture%20diagrams%20make%20it,be%20carefully%20and%20thoughtfully%20designed>

Key-Value Database: How It Works, Key Features, Advantages. (2022, November 28).

Retrieved January 8, 2023, from InfluxData website: [https://www.influxdata.com/key-value-database/#:~:text=A%20key%2Dvalue%20database%20\(also,known%20as%20key%2Dvalue%20pairs](https://www.influxdata.com/key-value-database/#:~:text=A%20key%2Dvalue%20database%20(also,known%20as%20key%2Dvalue%20pairs)

Serverless Architectures. (2023). Retrieved January 8, 2023, from Amazon Web Services, Inc.

website: <https://aws.amazon.com/lambda/serverless-architectures-learn-more/#:~:text=A%20serverless%20architecture%20is%20a,management%20is%20done%20by%20AWS>

UML - Behavioral Diagram vs Structural Diagram. (2022). Retrieved January 8, 2023, from

Visual-paradigm.com website: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/behavior-vs-structural-diagram/#:~:text=Behavioral%20Diagrams,-UML's%20five%20behavioral&text=It%20shows%20how%20the%20system,system%20to%20change%20internal%20states>

UML - Behavioral Diagram vs Structural Diagram. (2022). Retrieved January 8, 2023, from

Visual-paradigm.com website: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/behavior-vs-structural-diagram/#:~:text=linear%20time%20axis,-.Structural%20Diagrams,nodes%2C%20components%2C%20and%20interfaces>

What is Activity Diagram? (2022). Retrieved January 8, 2023, from Visual-paradigm.com website: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-activity-diagram/>

What is Amazon API Gateway? - Amazon API Gateway. (2023). Retrieved January 8, 2023, from Amazon.com website: <https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html>

What is Amazon DynamoDB? - Amazon DynamoDB. (2023). Retrieved January 8, 2023, from Amazon.com website: <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html>

What is Amazon S3? - Amazon Simple Storage Service. (2023). Retrieved January 8, 2023, from Amazon.com website: <https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html>

What is AWS Lambda? - AWS Lambda. (2023). Retrieved January 8, 2023, from Amazon.com website: <https://docs.aws.amazon.com/lambda/latest/dg/welcome.html>

What is Functional Testing? Types & Examples. (2020, April 3). Retrieved January 8, 2023, from Guru99 website: <https://www.guru99.com/functional-testing.html>

What is Sequence Diagram? (2022). Retrieved January 8, 2023, from Visual-paradigm.com website: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-sequence-diagram/>

What is Use Case Diagram? (2022). Retrieved January 8, 2023, from Visual-paradigm.com website: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/>

Why use serverless computing? | Pros and cons of serverless. (2023). Retrieved January 8, 2023, from Cloudflare website: <https://www.cloudflare.com/learning/serverless/why-use-serverless/>

Appendices

Appendix 1: Survey analysis & interviews summary

Planning:

Once the requirements have been finalized, the team members created a directory with all of the subtasks to be carried by each member, which were then reviewed and approved by the AWS consultants.

Below is a screenshot showing most of the subtasks that I was responsible for, all with due dates and comments on each task.

In the weekly meetings with the AWS consultants, they reviewed our work and gave us feedback on it. It was not an uncommon occurrence to have our work reviewed by the AWS team and getting negative feedback which led to us changing our approach to a task.

All of the team members stuck to their schedules and met each respective task's deadline.

<input type="radio"/>	modify lambda functions to make the system scalable	3/5	Hasan Ali Hasan Ali H...	12/2022	In-Progress
<input type="radio"/>	add functionality to redirect to passenger dashboard after taking survey fr...		Hasan Ali Hasan Ali H...	12/2022	Review
<input type="radio"/>	add 30 seconds timer to display qr page		Hasan Ali Hasan Ali H...	12/2022	Review
<input type="radio"/>	add button to take survey when tickets ahead is 0		Hasan Ali Hasan Ali H...	12/2022	Review
<input type="radio"/>	modify ticketCount function to handle multiple simultaneous connections		Hasan Ali Hasan Ali H...	12/2022	Review
<input type="radio"/>	update ticketing subsystem architecture		Hasan Ali Hasan Ali H...	12/2022	Done
<input type="radio"/>	Stream triggered lambda function to add ticket in queue(SQS)		Hasan Ali Hasan Ali H...	12/2022	Done
<input type="radio"/>	Add SNS functionality to TicketCount function		Hasan Ali Hasan Ali H...	12/2022	Done
<input type="radio"/>	Create putToQueue lambda function to send message (new ticket number)...		Hasan Ali Hasan Ali H...	11/2022	Done
<input type="radio"/>	create TicketCount lambda function		Hasan Ali Hasan Ali H...	12/2022	Done
<input type="radio"/>	Create SQS for two zones		Hasan Ali Hasan Ali H...	12/2022	Done
<input type="radio"/>	create and configure WebSocket API		Hasan Ali Hasan Ali H...	12/2022	Done
<input type="radio"/>	display saved ticket number on passenger device		Hasan Ali Hasan Ali H...	11/2022	Done
<input type="radio"/>	create PassengerMetrics database		Hasan Ali Hasan Ali H...	11/2022	Done
<input type="radio"/>	create dynamodb stream		Hasan Ali Hasan Ali H...	12/2022	Done

Figure 23 Project plan

User storyboards:

Storyboards have been created prior to starting the implementation to represent the problem that the client is facing in a non-technical manner, as well as how Takhatta resolves this issue in a real-world environment.

The below storyboard shows the issue, where in the first pane the passenger is shocked at how the check-in queue is so long and only one counter is opened. The passenger inquires with the employee about this issue and the check-in employee tells him that they are short staffed at the moment, which causes a delay for the passenger who cannot properly enjoy the airport facilities.

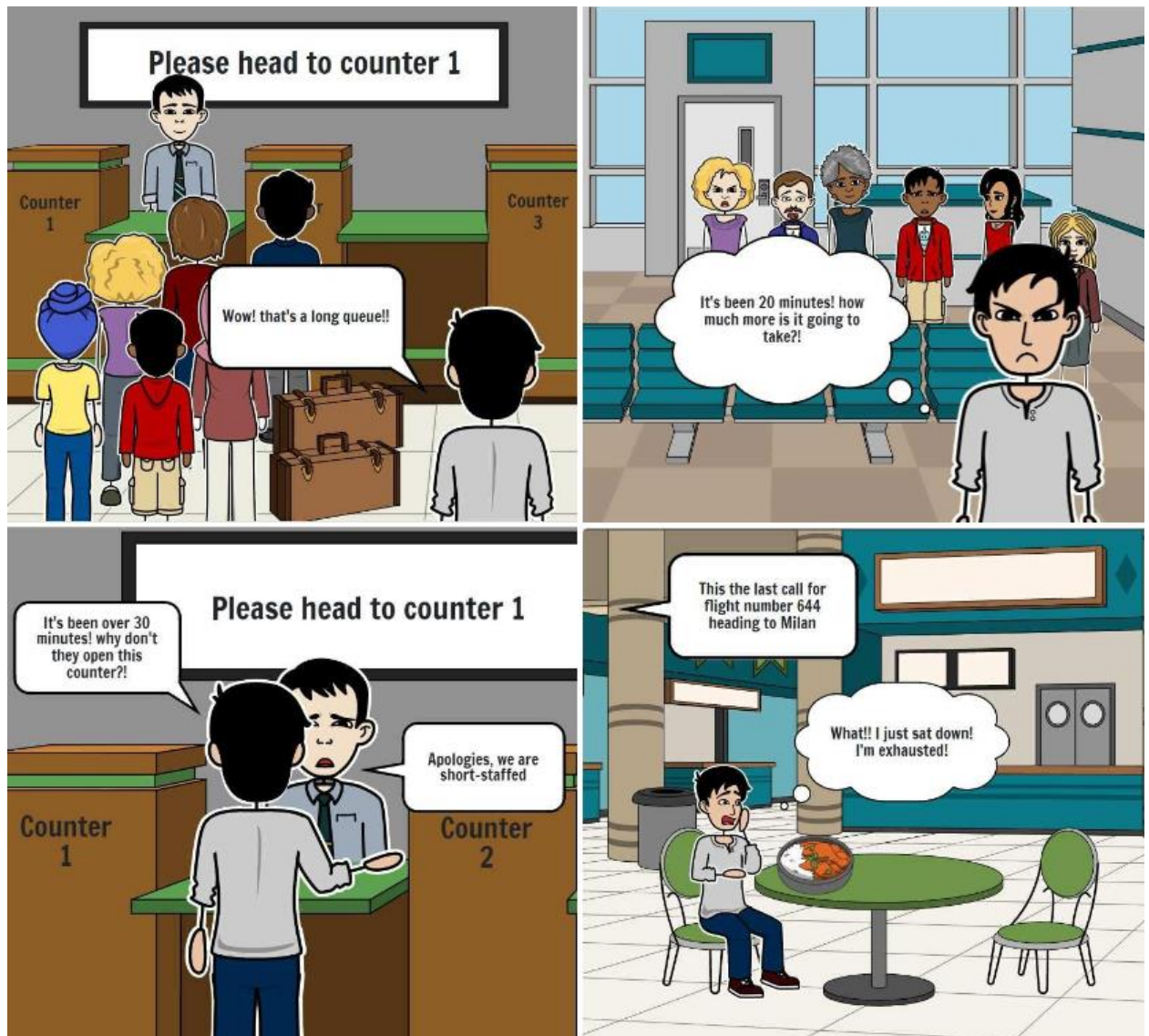


Figure 24 user storyboard – problem

The solution storyboards represent how different the passenger's experience is post the deployment of Takhatta. The staff manager gets accurate predictions of how many check-in employees need to be assigned for the expected passenger traffic, and he proceeds to assign 7 counters for that shift. The storyboard shows how the virtual queueing system works, where the passenger can sit down or enjoy the precheck-in facilities in the airport while waiting for his turn. Due to the controlled wait time, the passenger has enough time to check-in and enjoy the airport facilities without stressing about the flight boarding announcement.



Figure 25 user storyboard – solution

Customer press release:

The press release document has been provided by the client, where it highlights what the final prototype of Takhatta has to be capable of and was one of the methods used for gathering requirements.

The document has been reviewed by the AWS consultants and approved by the client.

Muharraq - BUSINESS WIRE - 1st Jan 2023 - BAC adopted the **Takhatta** system to meet their requirements of effectively allocating check-in desk staff which has increased customer satisfaction and significantly reduced queue times and cut back on costs. At **Takhatta**'s core, it predicts the number of staff needed based on incoming flights and virtually handles passenger queuing. It also includes a passenger dashboard that enables passengers to keep track of all the details related to their flight, a check-in dashboard to handle virtual queuing which replaced the need of physical queueing; as well as a management dashboard that allows BAC management team to forecast and filter staff allocation and number of incoming passengers for the upcoming month, as well as view the overall customer satisfaction.

BAC was facing various challenges due to the previously manual and inefficient staff allocation process; passengers were met with long queues which in turn increased the load on the check-in staff. In addition, passengers did not have enough time to enjoy the airport facilities as they spent most of their time at the airport check-in counters, which impacted BAC's revenue. On the other hand, BAC lacked a way to measure passengers' satisfaction for the check-in process and therefore were not able to take measurable actions to enhance the customer experience.

The newly adopted system "**Takhatta**" built on the Amazon Web Services Cloud is an innovative solution that uses modern technologies within an event-driven architecture to capture and analyze data. The system uses flights data to predict the ideal number of staff needed in the check-in area for each time of day to reduce the average queue time ~~by 60% to three minutes~~, which results in a faster and smoother check-in process. A management dashboard is created to visually display and filter the results. Alongside that, the system utilizes a virtual queuing system that passengers can access from their mobile devices, which enables BAC to capture the actual wait time for each passenger. The virtual queues are controlled by the check-in employees through the check-in dashboard. This eliminates the need for physical queueing, leaving passengers free to enjoy airport facilities even while waiting for their turn in the queue, which has revolutionized the passenger's check-in experience. A passenger survey to gauge feedback is also integrated within the queuing system to help the BAC team to continue to improve the experience further. After the passengers check in, they can access a dashboard to view live details about their flight with the ability to get notified of any updates to their flight or gate. As a result, adopting **Takhatta** has led to a 15% increase in passengers' satisfaction.

"The Bahrain Airport Company is passionate and committed to enhance their customer experience. **Takhatta** is here to give BAC a vehicle to accelerate their journey to Digital Transformation," said Mike Bainbridge, EMEA Cloud Innovation Centers leader at AWS. "We want BAC to tap into the latest AWS technologies to solve their real-world problems and increase customer satisfaction, optimize their staff allocation and ease the pressure on the airport facilities with efficiency and automation".

Passengers are now able to check in in less than 3 minutes through the virtual queue and enjoy airport facilities and new Duty-Free shopping experience with ease. While comfortably having access to the flight dashboard to monitor their flight details.

"It was an eye opener working closely with AWS and BP to develop **Takhatta** " said VP ICT at BAC – Mahmood AlSeddigi. "**Takhatta** helps us to predict the staff required to prevent future work overload and provides several additional services to improve our travelers experience"

Takhatta is now live at Bahrain International Airport, continuously predicting the airport staff needed in the check-in area and enhancing the experience of your next trip abroad.

Figure 26 BAC - Takhatta press release

Finally, the FAQ regarding the final implementation of the Takhatta prototype prior to deployment – with non-technical answers to make it understandable for the BAC operations team.

Frequent Asked Questions:

BAC Team FAQ

- How can I use the services?
Management Dashboard: Login with staff email/username and password
Passenger Dashboard: Find flight details using flight ID and date
Virtual Queueing Subsystem (Passengers Dashboard): Enter queue by scanning a QR Code
Virtual Queueing Subsystem (Employee Dashboard): Login with staff email/username and password
- Is there a user manual?
Yes
- Is it free or paid?
Free
- Is there any equipment needed?
 Tablets, TV, and speakers.
- How frequently is data updated?
 Management Dashboard: flights data change and real time
 Passenger Dashboard: real time
 Virtual Queue (passengers): real time
 Virtual Queue (employees): real time
- How does the prediction work?
 It simulates the check-in process and experiments with different numbers of employees until it meets the required average wait time.
- How will the virtual queuing system benefit BAC as well as the passengers?
 For the passengers, it will eliminate the physical queues, allowing them to enjoy the airport's pre-check-in facilities as they will have a live count of the tickets ahead of them.
 For the BAC, it allows them to precisely measure the actual wait time of passengers in queue.
- How does the passenger dashboard differ from the current system?
 The passenger's dashboard offers upgraded visuals and adds visual accessibility settings. It includes more information such as gate status, check-in island, check-in status, countdown, and more. In addition, passengers can opt-in to be notified when changes occur.

Stakeholders FAQ

- Is it adopted by BAC?
Yes
- Is Tamkeen funding the pilot?
Yes
- How many students are working on this challenge?
5
- Which university is leading
Bahrain polytechnic
- Is it going to be live or prototype
Live

Figure 27 BAC - Takhatta FAQ

Appendix 2: Design Diagrams

Use-case:

The use-case diagram of the virtual queueing system shows the actions that the actors carry out with the system and how each of these actions affect each other.

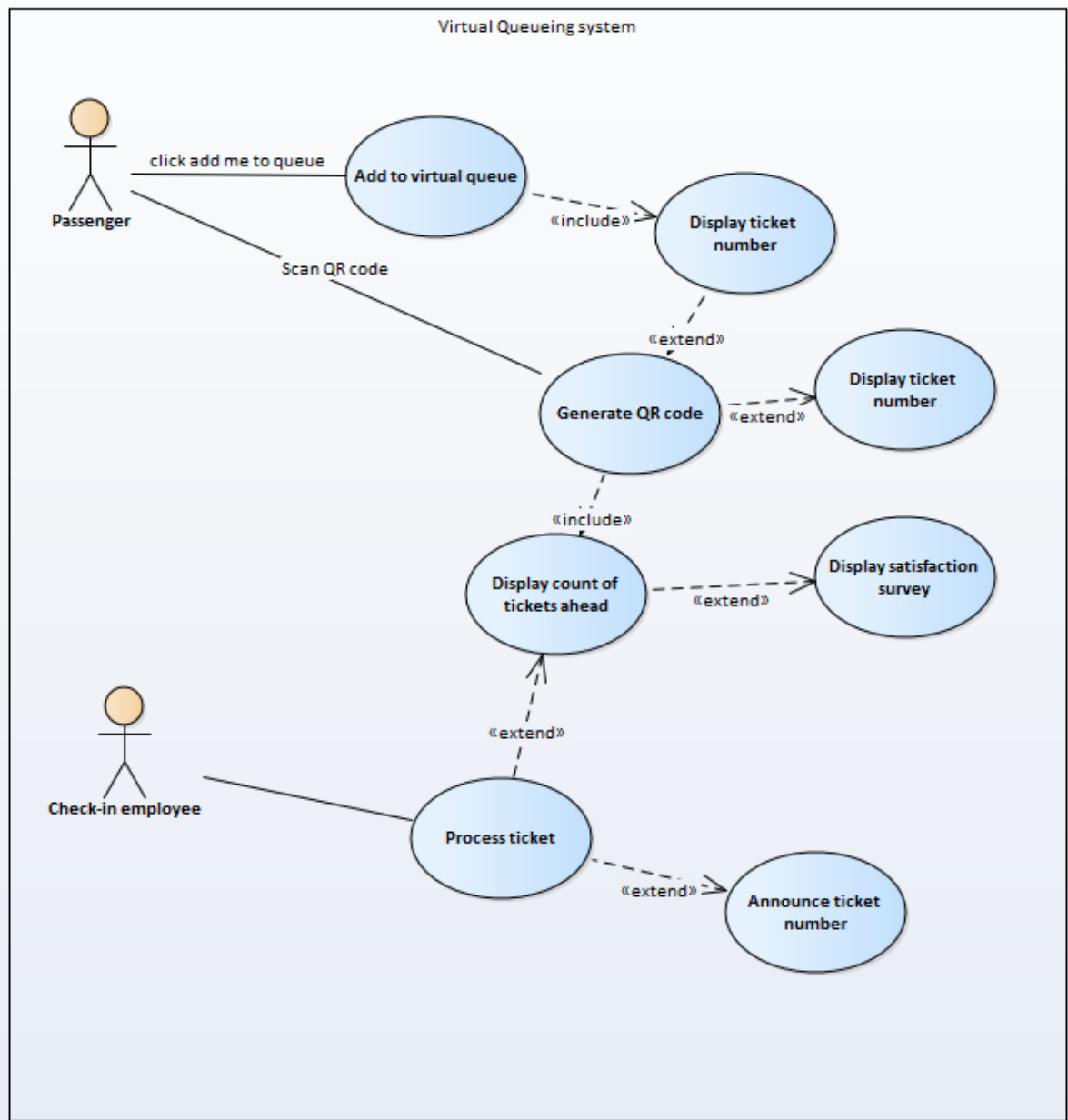


Figure 28 Virtual queueing system - use-case

Appendix 3: Source Code

Below is the full source code of all the Lambda functions used in the passenger virtual queueing system for Takhatta.

NewTicketFunction:

```
8
9 TABLE_NAME = "LatestTicketByZone"
10 def lambda_handler(event, context):
11     client = boto3.resource("dynamodb")
12     table = client.Table(TABLE_NAME)
13     response = table.query(
14         KeyConditionExpression=Key('Zone').eq('A'),
15         ProjectionExpression='LastTicket'
16     )
17     recentTicket = response["Items"]
18     TicketCount = response["Items"][0]['LastTicket']
19     if (TicketCount > 499):
20         #recentTicket[0]['LastTicket'] = 0
21         updateTable = table.update_item(
22             Key={
23                 'Zone': 'A'
24             },
25             UpdateExpression="set LastTicket = :val",
26             ExpressionAttributeValues={
27                 ':val': decimal.Decimal(1)
28             },
29             ReturnValues="UPDATED_NEW"
30         )
31     else :
32         updateTable = table.update_item(
33             Key={
34                 'Zone': 'A'
35             },
36             UpdateExpression="set LastTicket = LastTicket + :val",
37             ExpressionAttributeValues={
38                 ':val': decimal.Decimal(1)
39             },
40             ReturnValues="UPDATED_NEW"
41         )
42
```

Figure 29 source code- NewTicketFunction 1

```
44
45 class DecimalEncoder(json.JSONEncoder):
46     def default(self, obj):
47         if isinstance(obj, Decimal):
48             return float(obj)
49         return json.JSONEncoder.default(self, obj)
50
51
52 #return recentTicket
53 # return recentTicket
54 return {
55     'statusCode': 200,
56     #you will need to enable CORS and redeploy your API For the below headers to work.
57     'headers': {
58         'Access-Control-Allow-Headers': 'Content-Type' ,
59         'Access-Control-Allow-Origin': '*',
60         'Access-Control-Allow-Methods': 'OPTIONS,POST,GET'
61     },
62     #return the array as a response to the API request
63     'body': json.dumps(response["Items"][0]['LastTicket'],cls=DecimalEncoder)
64 }
```

Figure 30 source code- NewTicketFunction 2

PutToQueue:

```
import json
import boto3
from boto3.dynamodb.conditions import Key

TABLE_NAME = "LatestTicketByZone"
def lambda_handler(event, context):
    client = boto3.resource("dynamodb")
    table = client.Table(TABLE_NAME)
    response = table.query(
        KeyConditionExpression=Key('Zone').eq('A'),
        ProjectionExpression='LastTicket'
    )
    TicketToAdd = response["Items"][0]['LastTicket']

    sqs = boto3.resource('sqs')

    # Get the queue
    queue = sqs.get_queue_by_name(QueueName='ZoneA.fifo')

    # Create a new message
    response = queue.send_message(MessageBody=str(TicketToAdd),
        MessageGroupId='ZoneA')
```

Figure 31 source code – PutToQueue

onConnect:

The onConnect function was generated by AWS, but modifications have been made to store the passenger's ticket number along with their zone for purposes that have already been discussed earlier in the report. The function's runtime is Node.js.

```
const AWS = require('aws-sdk');

const ddb = new AWS.DynamoDB.DocumentClient({ apiVersion: '2012-08-10', region: process.env.AWS_REGION });

exports.handler = async event => {
    const putParams = {
        TableName: process.env.TABLE_NAME,
        Item: {
            connectionId: event.requestContext.connectionId,
            userTicket: event['queryStringParameters']['ticket'],
            userZone: event['queryStringParameters']['zone']
        }
    };

    try {
        await ddb.put(putParams).promise();
    } catch (err) {
        return { statusCode: 500, body: 'Failed to connect: ' + JSON.stringify(err) };
    }

    return { statusCode: 200, body: 'Connected.' };
};
```

Figure 32 source code - onConnect

onDisconnect:

The onDisconnect function was automatically generated by AWS as well, but modifications have been made here too. The modifications are to remove the connectionId of a disconnected passenger from the TicketCountTable, which is a custom table that I created for purposes that have been discussed earlier in the report.

```
const AWS = require('aws-sdk');

const ddb = new AWS.DynamoDB.DocumentClient({ apiVersion: '2012-08-10', region: process.env.AWS_REGION });

exports.handler = async event => {
  const putParams = {
    TableName: process.env.TABLE_NAME,
    Item: {
      connectionId: event.requestContext.connectionId,
      userTicket: event['queryStringParameters']['ticket'],
      userZone: event['queryStringParameters']['zone']
    }
  };

  try {
    await ddb.put(putParams).promise();
  } catch (err) {
    return { statusCode: 500, body: 'Failed to connect: ' + JSON.stringify(err) };
  }

  return { statusCode: 200, body: 'Connected.' };
};
```

Figure 33 source code – onDisconnect

TicketCount:

```
# TODO implement
TABLE_NAME = "LatestTicketByZone"
def lambda_handler(event, context):
    #set up environment
    URL = "https://20be7h92.execute-api.us-east-1.amazonaws.com/Prod"
    API = boto3.client("apigatewaymanagementapi", endpoint_url = URL)
    client = boto3.resource("dynamodb")
    sns = boto3.client('sns')
    table = client.Table(TABLE_NAME)
    countersTable = client.Table("countersOpened")
    dynamoDB = boto3.client("dynamodb")
    ticketCountTable = client.Table("ticketCountTable")

    #get the now serving ticket from dynamodb
    response = table.query(
        KeyConditionExpression=Key('Zone').eq('A'),
        ProjectionExpression= 'Ticket_NowServing, LastTicket'
    )

    #store values
    nowServingTicket = response["Items"][0]['Ticket_NowServing']
    LastTicket = int(response["Items"][0]['LastTicket'])
```

Figure 34 source code - TicketCount 1


```

# check if event is insert (new ticket)
event_type = event['Records'][0]['eventName']
if (event_type=='INSERT'):
    user = event['Records'][0]['dynamodb']['NewImage']
    ticketnum = user['userTicket']['S']
    zone = user['userZone']['S']
    conId = user['connectionId']['S']
    record = {
        "userTicket": {"S" : str(ticketnum)},
        "userZone": {"S" : str(zone)},
        "connectionId": {"S" : str(conId)}
    }
    response = ticketCountTable.put_item(
        Item={
            'userConnection': conId,
            'userTicketNum': ticketnum,
            'userIsland': zone
        }
    )
    scan = dynamoDB.scan(TableName = "ticketCountTable")

```

Figure 35 source code - TicketCount 2

```

#set tickets ahead to 1 to avoid dividing by 0 later
ticketsAhead=1
for item in scan["Items"]:
    userTicket = int(item["userTicketNum"]["S"])
    userConn = item["userConnection"]["S"]

    #do calculation and send response to specific id AND ticket
    if(item["userConnection"]["S"] == userConn and int(item["userTicketNum"]["S"]) == userTicket):
        if(nowServingTicket>LastTicket):
            ticketsAhead = 500-nowServingTicket+userTicket-1
            result = API.post_to_connection(ConnectionId=item["userConnection"]["S"],Data=json.dumps(int(ticketsAhead)))
        elif(nowServingTicket<LastTicket):
            ticketsAhead = userTicket-nowServingTicket
            result = API.post_to_connection(ConnectionId=item["userConnection"]["S"],Data=json.dumps(int(ticketsAhead)))
        #if there are no tickets ahead, delete table item
        else:
            result = API.post_to_connection(ConnectionId=item["userConnection"]["S"],Data=json.dumps(int(0)))
            response = ticketCountTable.delete_item(
                Key={
                    'userConnection': userConn
                }
            )

```

Figure 36 source code - TicketCount 3

```

#check if management was already notified, and get number of counters
response = countersTable.query(
    KeyConditionExpression=Key('Zone').eq('A'),
    ProjectionExpression= 'CountersOpened, Notified'
)
counters = response["Items"][0]['CountersOpened']
#set counters to 1 to avoid dividing by 0
if(counters == 0):
    counters = 1
managementNotified = str(response["Items"][0]['Notified'])

#calculation of wait time for all waiting tickets
# 3 is check-in time
waitMinutes = (ticketsAhead*3)/counters
#notify management
if(waitMinutes>25 and managementNotified=="No"):
    response = sns.publish(
        TopicArn='arn:aws:sns:us-east-1:505643665857:OpenCounterNotification',
        Message='Traffic is too high in island A. Consider opening more counters.',
        Subject='Urgent: Action required in island A.',
        MessageStructure='string',
        MessageAttributes={
            'Message': {
                'DataType': 'String',
                'StringValue': 'Urgent'
            }
        }
    )

```

Figure 37 source code - TicketCount 4

```

#if management gets notified, change in table
)
updateTable = countersTable.update_item(
    Key={
        'Zone': 'A'
    },
    UpdateExpression="set Notified = :r",
    ExpressionAttributeValues={
        ':r': "Yes"
    },
    ReturnValues="UPDATED_NEW"
)
return {
    'statusCode': 200
}

```

Figure 38 source code - TicketCount 5

Below are the tables that are accessed by the virtual queueing system.

LatestTicketByZone:

<input type="checkbox"/>	Zone	▼	Island	▼	LastTicket	▼	Ticket_NowServing
<input type="checkbox"/>	A		A		28		8
<input type="checkbox"/>	B		B		230		217

Figure 39 table - LatestTicketByZone

TicketCountTable:

Custom table that dynamically displays connected ticket numbers and their connectionId, created for purposes discussed earlier in the thesis

<input type="checkbox"/>	userConnection ▾	userIsland ▾	userTicketNum
<input type="checkbox"/>	eVUVadqPoAMCI6A=	A	28

Figure 40 table - TicketCountTable

Ticket Count Connections:

Automatically created by AWS upon configuration of WebSocket API.

<input type="checkbox"/>	connectionId ▾	userTicket ▾	userZone
<input type="checkbox"/>	eVUVadqPoAMCI6A=	28	A

Figure 41 table - Ticket_Count_Connections

CountersOpened:

Created for notification purposes.

<input type="checkbox"/>	Zone ▾	CountersOpened ▾	Notified
<input type="checkbox"/>	Total	16	
<input type="checkbox"/>	A	7	No
<input type="checkbox"/>	B	9	No

Figure 42 table - CountersOpened

Appendix IV: Testing Results

Testing costs:

The system has been thoroughly tested by the team members as well as the AWS consultants in various test cases to simulate real life testing.

The AWS console has a cost management directory that displays the costs. These figures are not precise for the deployment of a system but represent a close estimate of the monthly cost of deploying the system – and which services cost the most to run.

Around \$300 per month is the forecasted cost, which is a small price to pay considering the scale of the solution and how beneficial it is to BAC.

Cost summary

Current month costs [Info](#)

\$49.33

Down 3% over last month

Forecasted month-end costs [Info](#)

\$298.66

Down 85% over last month

Figure 43 Takhatta - cost summary