Burger King Receipt-Based Loyalty System on Azure

Version 2 – Architecture aligned with the latest diagram (Static Web Apps, Azure AD B2C, Blob Storage, Cosmos DB, Document Intelligence, Azure Functions, Email/SMS notifications, Azure Monitor / Application Insights / Log Analytics).

1. Functional Overview

This document describes a standalone loyalty program for a Burger King (BK) franchise that works purely from receipts
and does not integrate with the Aloha POS database. The system is hosted entirely on Microsoft Azure and can be
developed locally, then deployed through GitHub-based CI/CD.

High-level goal: convert a paper receipt into loyalty points and simple rewards, with a flow that is easy for customers,
easy for staff, and completely independent from the cash register.

1.1 Core Principles

• Customers earn points when they submit a valid receipt.

• Points are based on the money spent (example rule: 1 MAD = 1 point).

• At 100 points, customers can redeem a simple reward (e.g., free Sundae or free drink).

• Customers access everything from a QR/link printed on the receipt.

• Staff validate rewards from a separate web app (no POS integration).

• Identities (accounts) are stored in Azure AD B2C, while loyalty profiles and data live in Cosmos DB.

1.2 Customer Journey

1. Customer buys food at BK and receives a standard paper receipt.

2. The bottom of the receipt contains a QR code or short URL to the loyalty site.

3. The customer scans the QR and opens the customer web app (hosted in Azure Static Web Apps).

4. The customer signs up or logs in via Azure AD B2C (email/password or social login).

5. They upload a photo of the receipt.

6. The backend sends the image to Azure Document Intelligence (OCR) to extract total, date, time, and store info.

7. Azure Functions validate the receipt (no duplicate, recent enough), calculate points, and update the customer profile in Cosmos DB.

8. The customer sees their updated balance (example: 185 MAD spent → 185 points).

9. When the balance reaches the reward threshold (e.g., 100 points), the customer clicks “Redeem”.

10. The backend creates a reward entry in Cosmos DB and returns a QR reward code displayed in the app.

11. The customer presents this reward QR code during a future visit.

1.3 Staff Journey

1. Staff open the staff web app (also in Azure Static Web Apps) on a smartphone or tablet in the restaurant.

2. They choose “Scan reward”. The app uses the device camera to scan the customer’s QR reward code.

3. The staff app sends the reward identifier to the backend API (Azure Functions).

4. The backend checks Cosmos DB to verify that the reward exists, is pending, and not expired.

5. If valid, the backend marks the reward as USED and returns a clear message (e.g., “Valid reward – Free Sundae”).

6. Staff manually add the free item in Aloha, exactly as they would for a standard manual discount/goodwill gesture.

7. No change or integration is required on the POS side.

2. Azure Architecture (Aligned with Diagram)

The architecture is organized into logical groupings similar to the diagram you created: Client Side, Frontend Resource Group,
Identity Resource Group, Data & AI Resource Group, Backend Resource Group, Notifications Resource Group, and Monitoring Resource Group.
All resources live inside an Azure subscription dedicated to this project.

2.1 Client Side

• Customer Browser – mobile or desktop browser used by customers to interact with the loyalty program.

• Staff Browser – mobile or tablet browser used by BK staff to validate rewards.
Both connect to Azure via HTTPS only.

2.2 Frontend Resource Group – Azure Static Web Apps

• Azure Static Web Apps hosts two frontends:

– Customer Web App: public-facing app linked from the QR on receipts.

– Staff Web App: internal app for scanning and validating rewards (can be protected by specific AD B2C user flows or roles).

• Static Web Apps expose `/` for the frontend, and `/api` routes that proxy to Azure Functions.

• Static Web Apps are integrated with Azure AD B2C for authentication (OAuth2 / OpenID Connect).

2.3 Identity Resource Group – Azure AD B2C

• Azure AD B2C stores user accounts and handles signup/signin flows.

• It issues ID tokens and access tokens consumed by Static Web Apps and Azure Functions.

• Users are identified by their `sub` (subject) or `oid`, which is used as `userId` in Cosmos DB.

• B2C user flows or custom policies can support email/password and social providers (Google, etc.).

2.4 Data & AI Resource Group – Cosmos DB, Blob Storage, Document Intelligence

• Cosmos DB (Core API / SQL) stores all loyalty data:

– `Users` container for loyalty profiles and points balances.

– `Receipts` container for each submitted receipt.

– `Rewards` container for generated rewards and their status.

• Blob Storage stores:

– `receipts-raw` container: original ticket images uploaded by customers.

– `qrcodes` container: optional QR code images for rewards if you choose to generate actual images.

• Azure AI Document Intelligence processes the ticket images:

– Functions send Blob URLs.

– Document Intelligence returns structured fields (totalAmount, dateTime, storeId, etc.).

2.5 Backend Resource Group – Azure Functions

Azure Functions implement all server-side logic. Typical HTTP-triggered endpoints include:

• `POST /api/register-or-sync-user` – creates/updates a user profile in Cosmos DB after successful AD B2C login.

• `POST /api/upload-receipt` – receives metadata and/or Blob reference for a newly uploaded receipt image.

• `POST /api/validate-receipt` – runs OCR via Document Intelligence, checks duplicates, calculates points, and updates the user balance.

• `GET /api/user-balance` – returns the user's points and available rewards.

• `POST /api/create-reward` – checks if the user has enough points, creates a reward document, and generates a unique reward ID/QR value.

• `POST /api/redeem-reward` – called by the staff web app with a reward ID, verifies eligibility, marks reward as used, and returns the reward description to display to staff.

Functions use managed identities or connection strings to access Cosmos DB, Blob Storage, Document Intelligence, and the notification service.

2.6 Notifications Resource Group – Email/SMS

• A notification service (SendGrid or Azure Communication Services) is used to send:

– Welcome emails.

– "You earned X points" notifications.

– "You unlocked a reward" messages with QR code or link.

– Win-back campaigns if no receipt has been processed for N days.

• Azure Functions call this service when certain events occur (first signup, new receipt, new reward, etc.).

2.7 Monitoring Resource Group – Azure Monitor, Application Insights, Log Analytics

• Azure Monitor and Application Insights collect metrics and logs from Static Web Apps and Azure Functions.

• Log Analytics centralizes logs for queries and dashboards (e.g., number of receipts per day, number of valid vs rejected rewards, errors).

• This allows you to track performance, troubleshoot issues, and observe adoption of the loyalty program.

3. Data Model (Cosmos DB)

3.1 Users Container

Example fields:

• `id` – same as `userId`. Typically derived from AD B2C subject claim.

• `email` – customer email (optional if stored in B2C but convenient for queries).

• `pointsBalance` – current available points.

• `tier` – e.g., Bronze, Silver, Gold (future extension).

• `createdAt` – ISO timestamp when first profile was created.

• `lastVisit` – date of last valid receipt (used for win-back campaigns).

3.2 Receipts Container

Example fields:

• `id` (receiptId) – unique identifier for the receipt record.

• `userId` – foreign key to Users container.

• `storeId` – BK store identifier (e.g., BK#30743).

• `dateTime` – receipt date and time.

• `totalAmount` – numeric total spent.

• `imageBlobUrl` – link to the uploaded ticket image in Blob Storage.

• `uniqueHash` – hash from (storeId + date + time + totalAmount) to prevent duplicates.

• `pointsAwarded` – number of points given for this receipt.

• `status` – e.g., NEW, PROCESSED, REJECTED (bad image, outdated, duplicate).

3.3 Rewards Container

Example fields:

• `id` (rewardId) – unique reward identifier; also encoded into the QR value.

• `userId` – owner of the reward.

• `rewardType` – e.g., FREE_SUNDAE, FREE_DRINK, DISCOUNT_10_PERCENT.

• `qrCodeValue` – string encoded in the QR code (could be same as rewardId or include additional signature).

• `status` – PENDING, USED, EXPIRED.

• `createdAt` – timestamp when reward was created.

• `usedAt` – timestamp when reward was marked as used.

• `expiresAt` – optional expiry date for the reward.

4. Business Logic

4.1 Earning Points

Example base rule: `pointsEarned = floor(totalAmount)`.

Additional rules can be added later, such as double points in off-peak hours or welcome bonuses, but the minimal version keeps a simple 1 MAD = 1 point conversion.

4.2 Validating Receipts

Steps for validating a receipt:

1. Customer uploads the receipt image; it is stored in Blob Storage (`receipts-raw`).

2. Azure Functions call Document Intelligence with the Blob URL.

3. Document Intelligence returns structured fields (total, date, time, store).

4. Function builds a `uniqueHash` and queries the Receipts container to ensure this hash does not already exist.

5. If duplicate or too old (e.g., older than 48 hours), the receipt is rejected.

6. If valid, the receipt document is written to Cosmos DB, `pointsEarned` is calculated, and the user's `pointsBalance` in Users is updated.

7. Optionally, a notification is sent to the user summarizing points earned.

4.3 Creating Rewards

When a user has at least the threshold points (e.g., 100 points), the frontend shows a "Redeem reward" button.

On click:

1. The frontend calls `POST /api/create-reward` with the user's token.

2. The backend checks the user's `pointsBalance` from Cosmos DB.

3. If enough points, it subtracts the cost (e.g., 100 points), creates a Rewards document with status PENDING, and generates a `rewardId` / `qrCodeValue`.

4. The QR code (or alphanumeric code) is returned to the frontend and displayed to the user.

5. Optionally, a QR image is generated and stored in the Blob `qrcodes` container.

4.4 Redeeming Rewards (Staff View)

1. Staff open the staff web app and tap "Scan reward".

2. The app uses the device camera and a JavaScript QR reading library to extract the `qrCodeValue`.

3. The value is sent to `POST /api/redeem-reward`.

4. Azure Functions look up the Rewards container using the ID/value.

5. If reward does not exist, is already USED, or is EXPIRED, an error is returned.

6. If valid and PENDING, the function sets status to USED and stores `usedAt`.

7. The response includes a human-readable message, e.g., “Valid reward: Free Sundae – apply manually in POS”.

8. Staff manually apply the corresponding free item in Aloha. The POS system is not aware of or connected to the loyalty system.

5. Development and Deployment on Azure

5.1 Initial Azure Setup

1. Create or select an Azure subscription dedicated to this project.

2. Create a resource group, e.g., `rg-bk-loyalty` (you can later split into Frontend RG, Backend RG, Data & AI RG, etc., as shown in the diagram).

3. Set up Azure AD B2C tenant and link it to your subscription.

4. Create a Cosmos DB account (SQL API), Blob Storage account, and Document Intelligence resource.

5. Create an Azure Functions app (consumption or premium plan).

6. Create an Azure Static Web App for the customer frontend and another for the staff frontend (or a single app hosting both with different routes).

7. Configure a notification service (SendGrid or Azure Communication Services).

8. Configure Application Insights / Log Analytics for Azure Functions and Static Web Apps.

5.2 Local Development

Recommended local tooling:

• Node.js + a React/Next.js framework for the frontend.

• Azure Functions Core Tools for running and debugging Functions locally.

• Azure Static Web Apps CLI for emulating the combined frontend + API locally.

• Azurite (Storage emulator) and optionally Cosmos DB emulator if you prefer offline testing.

Typical dev flow:

1. Implement the React frontends (customer + staff).

2. Implement the Functions endpoints and test them with HTTP calls and mocked Document Intelligence responses.

3. Integrate with real Azure services once the logic is stable.

4. Use environment variables / config files to separate local and cloud configurations.

5.3 GitHub CI/CD

1. Create a GitHub repository containing frontend and backend code (monorepo is fine).

2. Connect the repo to Azure Static Web Apps to auto-generate a GitHub Actions workflow file.

3. Add another workflow for deploying Azure Functions (or use the Static Web Apps integrated API deployment pattern).

4. On push to main branch, GitHub Actions will build the frontend, package Functions, and deploy to Azure.

5. Use deployment slots or staging environments if you want to test changes before production.

5.4 Operations and Monitoring

• Use Application Insights to track slow requests, failures, and dependency calls (Cosmos DB, Blob, Document Intelligence).

• Create Log Analytics queries and dashboards for:

– Receipts processed per day.

– Points awarded per store.

– Rewards created vs used.

– Errors in OCR or validation.

• Configure alerts (email/SMS) for high error rates or anomalies (e.g., potential abuse patterns).

6. Summary

The system described here fully matches the architecture shown in your latest diagram:
client browsers connect to Azure Static Web Apps, which integrate with Azure AD B2C for authentication and with Azure Functions for business logic.
Functions coordinate data in Cosmos DB, store images in Blob Storage, call Azure Document Intelligence for OCR, send notifications via an email/SMS service,
and send telemetry to Azure Monitor / Application Insights / Log Analytics.
Rewards are validated by staff using only a browser-based app, with no POS integration required.
This PDF can be reused as a master specification for future development, debugging, and for initializing new AI conversations about this project.