ChatGPT

PiePay-Style E-Commerce Offer Platform: Technical Specification & Implementation Guide

The goal is to build a MERN/Firebase web application that lets buyers without credit cards leverage card offers on Amazon/Flipkart by pairing them with volunteer cardholders. Buyers submit product links or details, the system scrapes metadata (title, image, price) from Amazon/Flipkart, and then notifies eligible cardholders via push notification (Firebase Cloud Messaging). Cardholders can accept the deal, make the payment with their credit card (possibly split with the buyer's gift card), and earn a commission, while the buyer gets the discount. The discount is split (e.g. 60% to buyer, 30% to cardholder, 10% to platform) via an internal wallet system, and workflows are fully automated (no manual approvals). The app uses **React** (frontend) on Vercel, **Node.js/Express** (backend) on Render, **MongoDB Atlas** (database), and **Firebase** (Authentication + FCM) under free-tier plans. The overall high-level architecture is illustrated below [1] [2].

High-level architecture of the MERN stack with Firebase services. React (front-end) communicates with an Express backend; the backend uses MongoDB Atlas for data and Firebase Authentication + Messaging for user auth and notifications [1] [2]. The backend also scrapes Amazon/Flipkart pages for product info. Hosting is on free tiers (Vercel for React, Render for Express, Atlas M0 cluster).

Architecture & Components

- **Frontend (React on Vercel):** A Single-Page App (SPA) built with React (or Next.js) provides login (Firebase Auth), deal requests, wallet balance, receipt upload, and (optionally) admin views. It communicates with the Express API and subscribes to push notifications via FCM. Vercel offers free hosting with automatic HTTPS and CDN support [3]; connecting your Git repo to Vercel enables one-click deployment of the React app [3].

- **Backend API (Node.js/Express on Render):** The Express server exposes REST endpoints for deal requests, deal matching, payments, wallet updates, receipts, etc. It verifies Firebase JWTs for authentication. Key backend tasks include: orchestrating deal matching, sending push notifications via Firebase Admin SDK, scraping product pages, and splitting payments. Render's free web service tier allows deploying a Node app from GitHub with auto-deploy on each push [4] [5].

- **Database (MongoDB Atlas Free Tier):** A cloud MongoDB cluster (Atlas M0 free tier, 512 MB storage [6]) stores collections for Users, Deals/Requests, WalletTransactions, GiftCards, and Receipts. For example:

- **Users**: `{_id, name, email, role (buyer/cardholder/admin), firebaseUid, fcmToken, walletBalance}`,

- **Deals/Requests**: `{_id, buyerId, productInfo (title, price, image, link), discountPercent, status, matchedCardholderId, timestamps}`,
- **WalletTransactions**: `{_id, userId, amount, type ("buyer_reward"/"cardholder_commission"/"platform_fee"), dealId, timestamp}`,
- **GiftCards**: `{code, initialAmount, remainingAmount, assignedToUserId}`, etc.

Atlas's free cluster can support basic workloads for development and demos [6].

- **Authentication (Firebase Auth):** Use Firebase's Spark (free) plan for user authentication (email/password, Google login, etc.). The Firebase Spark plan covers "Authentication (most options)" at no cost [7]. Users log in via Firebase; the React app gets an ID token which the backend verifies using Firebase Admin. This avoids building your own auth system.

- **Real-time Notifications (Firebase Cloud Messaging):** Use FCM to push browser notifications. Each client (buyer or cardholder) obtains an FCM registration token and sends it to the backend to store in the user's record [8]. To notify cardholders of a new deal, the backend uses the Firebase Admin SDK (`npm install firebase-admin`) to send a message to the cardholders' tokens [9] [8]. On the React side, include Firebase Messaging and a `firebase-messaging-sw.js` service worker (see **Service Worker setup** below) so that even backgrounded browsers show notifications [10].

- **Product Scraping (Amazon/Flipkart):** When a buyer enters a product link or ID, the backend fetches the page HTML and parses it for metadata. For static content, libraries like **Axios + Cheerio** can scrape the HTML for title, price, and image [11]. If the site loads data dynamically, a headless browser (Puppeteer) can render the page first [12]. Example code: use `axios.get(url)` then `cheerio.load(html)` to extract e.g. `$('meta[property="og:title"]').attr('content')`. If scraping fails (site change or block), the API can return a mock or cached product info so the app remains responsive. Puppeteer can also be used (via `await puppeteer.launch()…`) to ensure dynamic content is captured [12].

- **Wallet & Discount Splitting:** Each user has a wallet balance. When a deal completes, the discount amount is calculated and split. For example, on a 100 ₹ discount: the buyer's wallet is credited 60₹, the cardholder's wallet 30₹, and 10₹ goes to the platform's reserve. These percentages are configurable. The server creates corresponding WalletTransactions entries to record each user's share. (PiePay's model similarly "saves money" for the buyer and "earns commission" for the cardholder [13].) In code, after verifying a successful purchase (see next), the backend runs `buyer.walletBalance += (discount * 0.6)`, `cardholder.walletBalance += (discount * 0.3)`, etc., and saves transactions.

- **Gift Cards & Split Payment:** Buyers can apply a gift card to cover part of the product cost, with the remainder charged to the cardholder's credit card. For instance, if a 1000₹ product has a 100₹ discount, and the buyer uses a 200₹ gift card, only the 800₹ balance is charged to the card. The 100₹ discount is then split proportionally (60% to buyer's wallet, etc.) only on that 800₹ portion. This matches Amazon's policy of allowing a gift card plus one credit card [14]. The app must record the gift card code and remaining balance; e.g. in the GiftCards collection, track `remainingAmount`.

- **Invoice/Receipt Upload:** After the cardholder makes the purchase, the buyer (or cardholder) uploads the order invoice or receipt image to confirm. The React app can send the image (or a file

URL) to the backend, which stores it (e.g. in Firebase Cloud Storage or as base64 in Mongo). Firebase Storage offers a generous free tier (5 GB, daily quotas) [15] for storing images, so it can host receipts if desired. The upload triggers the final step: after the e-commerce return-period (e.g. 7 days for Amazon), the system automatically reimburses the cardholder and distributes the discount via wallets. All these steps happen without human intervention (Refunds and reimbursements can be scheduled via a cron job or similar at 7-day intervals).

- **Notifications Workflow:** When a buyer creates a deal request, the backend pushes an FCM message to all cardholders (or a subset matching the deal) [9] [16]. For example: `"New Deal: [Product Name] at [Price] (100₹ off) – accept within 15 min!"`. In React, the service worker (`firebase-messaging-sw.js`) listens for these background messages and calls `showNotification()` [10] to display them. (If the app is in the foreground, the onMessage handler can display an in-app alert.) This real-time alert lets cardholders see and accept deals quickly. PiePay advertises that "you are matched with the right Cardholder and the order is placed within 15 mins" [16], so implement a 10–15min timeout: if no one accepts the deal by then, mark it expired.

- **Automation & Matching:** There is no admin approval step. The matching algorithm can simply notify all available cardholders (or those who opted in), and the first to accept wins. Backend routes like `POST /api/deals/:id/accept` are protected so only registered cardholders can call them, and a simple check `deal.status === "pending"` ensures only one accept. Once accepted, the backend sets `deal.status = "matched"` and records `deal.cardholderId = userId`. It then notifies the buyer (optional) that a cardholder was found. All processes (matching, reimbursements, wallet credit) run automatically according to triggers (incoming accept, receipt upload, elapsed time).

Implementation Details

Frontend (React): Use Create-React-App or Next.js. Set up Firebase SDK (v9 or v8) for Auth and Messaging. In `index.js`, initialize Firebase with your config, and request notification permission. Implement a `firebase-messaging-sw.js` file in `public/` with code like:

```
importScripts("https://www.gstatic.com/firebasejs/9.0.0/firebase-app-compat.js");
importScripts("https://www.gstatic.com/firebasejs/9.0.0/firebase-messaging-compat.js");
firebase.initializeApp({ /* Your config */ });
const messaging = firebase.messaging();
messaging.onBackgroundMessage(payload => {
  const {title, body, icon} = payload.notification;
  self.registration.showNotification(title, { body, icon });
});
```

This follows best practices [10]. On the client, after login, get `const token = await messaging.getToken()`, then POST it to your backend (e.g. `/api/users/fcm-token`) so the server can send notifications to this device [8]. Use React Router to manage pages: a page for **Deal Request** where

the buyer enters a product URL (or ASIN) and optional gift card; a **Deals List** page for cardholders showing available deals; a **Wallet** page showing balance and transactions; and a **Receipts** page where buyers upload images of receipts. Use UI libraries (e.g. Material-UI or Bootstrap) for a clean, responsive layout.

Backend (Express): Structure code with separate routes/controllers (e.g. `deals.js`, `users.js`). Use Mongoose for schema models. Example route flow:

- **POST** `/api/deals` (Buyer sends new deal request): Validate product URL, scrape metadata (with Cheerio or Puppeteer as needed [11] [12]), create a Deal document (`status="pending"`), and then use the FCM admin SDK to send a "new deal" message to all active cardholder tokens [9] [8].
- **GET** `/api/deals` (Cardholder polls for available deals): Return all pending deals. Alternatively, have the frontend rely on push notifications and then fetch details via API.
- **POST** `/api/deals/:id/accept` (Cardholder accepts): Check deal still pending, set it to `matched` and record `cardholderId`. Respond so the front-end can navigate to a payment/invoice step.
- **POST** `/api/deals/:id/complete` (Buyer uploads receipt): Store the receipt (e.g. in Mongo using GridFS or in Firebase Storage) and trigger the finalization logic. At completion (or via a scheduled job 7 days later), compute the discount split and update wallets: 60% to buyer, 30% to cardholder, 10% to admin/account. Save WalletTransaction records accordingly.

Also include routes for **gift cards**: e.g. **POST** `/api/giftcards` to register a gift card (admin or buyer action), and **POST** `/api/giftcards/:code/use` to deduct an amount. The checkout logic should deduct from gift card first, then create a payment record for the remainder (simulated).

For payments, you could integrate a gateway (Stripe, etc.) for actual processing, but since PiePay's model assumes off-platform card use, a simpler approach is: record an invoice, trust that the cardholder has paid, then mark the transaction done once the receipt is verified. (In a real app, you might embed Amazon's API or direct the cardholder to place the order themselves.)

Database Schema Example:

Collection	Key Fields (examples)
`Users`	`{ _id, firebaseUid, email, name, role:"buyer"/"cardholder"/"admin", fcmToken, walletBalance }`
`Deals`	`{ _id, buyerId, productTitle, productPrice, productImage, discountPct, giftCardUsed:bool, status:"pending"/"matched"/"completed", cardholderId, createdAt, expiresAt }`
`WalletTransactions`	`{ _id, userId, dealId, amount, type:"buyer_reward"/"cardholder_commission"/"platform_fee", timestamp }`
`GiftCards`	`{ code, initialAmount, remainingAmount, assignedToUserId }`
`Receipts`	`{ dealId, imageUrl, uploadedAt }`

This schema allows tracking all necessary data. The free-tier Atlas supports storing these collections comfortably [6].

Key Workflows & Flows

1. **Requesting a Deal (Buyer Flow):** The buyer logs in via Firebase Auth and submits a product URL (or selects an item). The frontend calls `POST /api/deals` with `{ url, giftCardCode? }`. The backend scrapes the page for title/image/price [11] and creates a Deal. The backend immediately broadcasts a push notification to cardholders using FCM Admin [9]. The entire request-to-notification cycle happens in real time, aiming for under 1-2 seconds so the buyer gets matched "within 15 mins" [16].

2. **Accepting a Deal (Cardholder Flow):** A logged-in cardholder receives a browser notification (FCM) saying, e.g., "New deal: [Product Name] – ₹100 off". The cardholder clicks into the app, sees the deal details (via a `GET /api/deals` call) and clicks "Accept". The frontend calls `POST /api/deals/:id/accept`. The backend verifies the deal is still pending, marks it matched to this cardholder, and replies success. The buyer can be notified (UI alert) that a cardholder has accepted.

3. **Ordering & Payment:** Once matched, the cardholder purchases the item on Amazon/Flipkart using their credit card (possibly combining with the buyer's gift card). For example, if the product is ₹1000 with ₹100 discount, and the buyer had a ₹200 gift card: ₹200 is paid via gift card, ₹800 via credit card. Only the ₹800 portion generates the ₹100 discount (since gift cards have no offers) [14]. The cardholder then uploads the invoice/receipt (or sends it to the buyer), who uploads it to the app.

4. **Receipt & Settlement:** The buyer uploads the receipt image via `POST /api/deals/:id/complete` or a dedicated endpoint. The backend stores the receipt (e.g. in Firebase Storage – Spark plan allows 5 GB free, with daily free quotas [15]). After verifying delivery (optionally automated by waiting a week or listening for "delivered" status from an e-commerce API), the system "completes" the deal: it calculates the total discount and splits it 60/30/10 into wallets. For instance, if ₹100 discount, buyer gets +₹60, cardholder +₹30, platform +₹10 in their internal wallets. These are recorded in `WalletTransactions`. All of this is automatic – no human admin action. (Optional: if a return is initiated, reverse transactions accordingly.)

5. **Notifications and Timers:** The backend sends push notifications for key events: new deals to cardholders, deals accepted to the buyer, and perhaps deal completed. We use FCM topics or direct tokens; e.g., all cardholders could subscribe to a `"deal-requests"` topic, or we send individually. The React app's service worker displays these using `showNotification()` [10]. Since the session should be quick, unaccepted deals expire (status set to "expired") after ~15 minutes [16].

6. **Full Automation:** There is no manual approval. For example, when a buyer enters a deal, the system automatically handles matching and notifications. When a receipt is uploaded, the system automatically splits wallets after a delay. If anything goes wrong (no accepts, no receipt), the backend marks the deal failed and notifies the buyer. All business logic (commission calculations, refunds) is coded in backend services.

Technical Setup & Deployment

- **Dev Environment:** Develop locally with Node.js (v18+) and React. Use environment variables for sensitive data (store Firebase service account JSON on the server, `MONGO_URI`, etc.). Use Git for version control.

- **Firebase Setup:** In the Firebase console, create a new project. Enable Email/Password (and any OAuth) in Authentication. Enable Cloud Messaging. Download a Service Account JSON for server usage. Note: Firebase Spark plan is free with full Auth and FCM support [7].

- **MongoDB Atlas:** Create an M0 free cluster on MongoDB Atlas. Get the connection URI (store in Render's environment variables). The M0 tier offers 512 MB storage (sufficient for prototyping) [6].

- **API Implementation:** Install necessary packages: `express`, `mongoose`, `firebase-admin`, `axios`, `cheerio`, `node-fetch` (or `puppeteer`), `cors`, `body-parser`, etc. Initialize Firebase Admin with the service account to send FCM. Example snippet:

```
const admin = require('firebase-admin');
const serviceAccount = require('./serviceAccount.json');
admin.initializeApp({
  credential: admin.credential.cert(serviceAccount)
});
// Later: admin.messaging().send({token, notification: {...}});
```

- **Push Notifications (Server):** When notifying cardholders, query the Users collection for all cardholders' `fcmToken`s and call `admin.messaging().sendMulticast()` or individual sends [9] [8].

- **Hosting:**

- *Frontend:* Connect your GitHub repo to Vercel. Vercel auto-detects a React app (`npm build`) and deploys a CDN-backed site with free SSL [3]. Any push to main triggers a preview and production deploy.

- *Backend:* On Render.com, create a new "Web Service" from Git. Choose Node, provide the build command (`npm install`) and start command (`node index.js`). Set environment variables in the Render dashboard (e.g. `MONGO_URI`, `FIREBASE_PROJECT_ID`, etc.). On push to main, Render will auto-deploy the new version [4] [5]. Render provides a free TLS-certified public URL (`your-app.onrender.com`).

- **Hosting Limitations:** All chosen services have free tiers. Firebase Auth and FCM are free [7]. Firebase Storage has daily free quotas [15] (good for receipts). Mongo Atlas M0 is free forever [6]. Vercel and Render both have free plans. Thus the entire stack can run at zero cost (aside from domain costs if you add a custom domain, but even that can use a free `onrender.com` or `vercel.app` domain).

- **CI/CD and Monitoring:** You can set up GitHub Actions for testing on push. Render provides basic logs and health checks (free). For admin analytics, integrate Google Analytics (free) in the React app, or use MongoDB Atlas charts (paid beyond M0). For a quick admin UI, use a React dashboard template (e.g. Material-UI admin templates, Chart.js or Nivo for charts showing metrics like total deals, commissions, user counts).

- **Wallet & Admin Dashboard:** An optional admin page (protected by an "admin" role) can show summary statistics (total users, total deals, total commissions, active deals). Use chart libraries (e.g. Chart.js or Recharts) to display these. MongoDB Atlas allows setting up simple charts on free tier (automatic refresh every 4h) [17], but a full admin page built into React is more flexible.

Best Practices & Resume-Ready Tips

- **Code Quality:** Organize code with clear folders (e.g. `routes/`, `models/`, `controllers/`). Use linter (ESLint) and formatter (Prettier). Comment tricky logic. Use TypeScript if comfortable, for type safety.
- **Security:** Do **not** store actual credit card numbers anywhere. Only manage metadata and tokenized auth. Use HTTPS (both Vercel and Render provide this by default). Implement CORS so only your front-end can call your API. Sanitize all inputs to prevent injection. Verify Firebase tokens on each API call to protect endpoints. Store secrets (Firebase service account, DB URI) in environment variables only.
- **Testing:** Write unit tests (Jest or Mocha) for critical functions (e.g. discount-splitting logic, scraping functions). Automate tests on each PR with GitHub Actions.
- **Error Handling & Logging:** Use try-catch around async calls. Return meaningful HTTP status codes. Log errors (console or use a free logging service). On Render, view logs in the dashboard; for frontend errors consider integrating Sentry (free tier).
- **Workflow Reliability:** Use transaction or two-phase commit patterns if needed when splitting wallets to avoid inconsistencies. Consider idempotency (if the same webhook hits twice). Use a TTL or cron to expire stale deals (15 min rule) and to finalize reimbursements.
- **Documentation & Demo:** Write a detailed README explaining setup, APIs, and how to use the app. Provide screenshots or a demo video. A clear UI with guided flow (signup → request deal → see notifications → complete purchase) will impress reviewers. Highlight that all features run on free tiers.

By following this plan—using React/Express/Mongo with Firebase Auth/FCM, implementing scraping with Axios/Cheerio (or Puppeteer), designing robust data schemas for users/deals/wallets, automating all matching and payouts, and deploying on free platforms—you'll create a fully operational PiePay-like system. This project not only demonstrates full-stack skills but also systems design and integration expertise, making it resume/demo-ready.

Sources: Relevant documentation and examples used include PiePay's app description [13] [18] [19], tutorials on React+Node architectures [20], Node.js scraping techniques [11] [12], Firebase Cloud Messaging setup [10] [9], and platform pricing details [7] [6] [15]. These guided the design of the system components, workflows, and use of free services.

---

1 Containerized React Web App with Node.js API and MongoDB on Azure - Code Samples | Microsoft Learn
https://learn.microsoft.com/en-us/samples/azure-samples/todo-nodejs-mongo-aca/todo-nodejs-mongo-aca/

2 20 Using NodeJS with React.js | Perfect Stack for Modern App Development
https://www.itarch.info/2023/07/using-nodejs-with-reactjs-perfect-stack.html

3 Vercel
https://vercel.com/solutions/react

4 5 Deploy a Node Express App on Render – Render Docs
https://render.com/docs/deploy-node-express-app

6 17 Pricing | MongoDB
https://www.mongodb.com/pricing

7 15 Firebase pricing plans | Firebase Documentation
https://firebase.google.com/docs/projects/billing/firebase-pricing-plans

8 9 Push Notification using firebase and node-js. | by Bisal Rajan Padhan | Medium
https://medium.com/@Bisal.r/push-notification-using-firebase-and-node-js-7508f61fa25c

10 Firebase Cloud Messaging in React: A Comprehensive Guide | by Vanshita Shah | Simform Engineering | Medium
https://medium.com/simform-engineering/firebase-cloud-messaging-in-react-a-comprehensive-guide-b5e325452f97

11 12 Develop Amazon Product Scraper Using Node.js
https://www.realdataapi.com/develop-amazon-product-scraper-using-nodejs.php

13 16 18 19 PiePay: Avail Card Offers - Apps on Google Play
https://play.google.com/store/apps/details?id=com.piepay&hl=en_US

14 Split payments: Using two cards for one transaction | CreditCards.com
https://www.creditcards.com/education/split-payment-transaction-online-two-cards/