CaughtMyEye

# ***With the rapid advancement of artificial intelligence (AI) and the widespread adoption of image recognition and location-based services, today’s travelers increasingly demand intelligent tools that streamline trip planning through instant, personalized recommendations. The convergence of computer vision, geolocation technology, and real-time data processing has enabled a new generation of mobile applications capable of interpreting visual content—such as photos of landmarks—and transforming them into actionable travel insights. These context-aware systems go beyond traditional search-based planning, offering dynamic suggestions for flights, accommodations, and activities based on real-world imagery. As a result, AI-powered travel assistants are no longer a novelty but an expectation, reshaping how users discover destinations, plan itineraries, and interact with their surroundings. This shift underscores the need for seamless integration between visual input, AI-driven analysis, and travel service APIs to deliver a cohesive and intuitive user experience.***

# introduction

With the rapid advancement of computer vision, artificial intelligence, and location-based technologies, the way we interact with our physical surroundings is changing faster than ever. Travelers today are more curious, more mobile, and more connected. They no longer just visit places; they document, share, and want to immediately learn about them in real time. When someone stumbles upon an unfamiliar landmark or scenic view and takes a picture, their next instinct is often to ask: “What is this place?” or “What else is nearby that I can explore?”

While several tools currently exist to help answer these questions, they often fall short of delivering a complete, intuitive experience. Some applications can identify a landmark or provide directions, others can find hotels or attractions, but few can do all of it in one place. Most require users to switch between multiple apps, manually enter location data, or sift through scattered information. This fragmentation creates unnecessary friction in the user experience and makes spontaneous exploration less enjoyable than it could be.

To solve this, we introduce CaughtMyEye, an AI-powered travel planning application that bridges the gap between visual curiosity and actionable information. The core idea is simple: take a photo or scan a location, and instantly receive a set of personalized travel recommendations, including the landmark’s name, nearby accommodations, flight options, and a mapped overview of the area. By combining deep learning models for image classification with real-time data from location and travel APIs, we aim to deliver a unified and context-aware experience.

Our system uses convolutional neural networks (CNNs), trained on diverse datasets of global landmarks, to analyze and recognize images with high accuracy. Once a landmark is detected, we use its geolocation data (latitude and longitude) to query mapping and travel APIs, pulling together flights, hotels, and local information in a seamless interface. The end result is a single-page, user-friendly experience that answers the traveler’s most pressing question: “What is this place, and how can I get there?” without ever needing to leave the app.

CaughtMyEye is designed not just to provide answers, but to encourage discovery. Whether a user is exploring locally, planning a bucket-list trip, or simply browsing travel inspiration from their phone, this tool transforms a moment of curiosity into a clear path forward.

# problem statement

Tourists and casual travelers often find themselves in unfamiliar locations, curious to know more about the places they encounter. They may take a photo of a building, statue, or natural landmark, hoping to learn its name, significance, or discover what attractions lie nearby. However, the tools currently available to assist them are often limited or inefficient when it comes to providing this kind of contextual information.

Apps like Google Lens, Yelp, or TripAdvisor offer partial solutions. Some can recognize landmarks, others provide reviews or suggest nearby spots, but most require manual input or jumping between separate platforms. This leads to a disjointed experience where the user must piece together information from multiple sources. The lack of a streamlined, all-in-one solution makes spontaneous exploration harder than it should be and reduces the excitement of discovering a new place.

The real challenge is building an intelligent system that can do three things effectively. First, it must accurately classify an image of a location based on visual features. Second, it needs to identify the geographical identity of that location with precision. Third, it should present useful recommendations such as nearby hotels, attractions, and travel options, all within a single, easy-to-navigate interface. Achieving this requires advanced methods in image recognition, semantic interpretation, geolocation analysis, and smooth integration with various APIs, all while ensuring that the system remains fast, reliable, and user-friendly.

CaughtMyEye aims to solve this exact problem. By using pre-trained convolutional neural networks to identify and categorize landmark images, and combining those predictions with accurate GPS coordinates and third-party travel APIs like Google Maps and Amadeus, we are able to generate context-aware, visual-first travel recommendations. The goal is to provide users with instant, intelligent guidance based on nothing more than a photo or a scan, helping them make informed decisions and explore the world with confidence.

# methodologies

The Home screen welcomes users with an elegant gradient background and a concise description highlighting the application's core functionality: detecting landmarks from user-uploaded images and automatically generating comprehensive travel plans complete with relevant flight and hotel options. For user authentication, we implemented a robust system using Firebase Authentication, offering multiple secure sign-in methods to accommodate different user preferences. This includes traditional email/password authentication, new account creation with email verification, and seamless OAuth integration through GoogleAuthProvider for one-tap sign-in with Google accounts.

All authentication data is securely stored and managed through Firebase's built-in services - when users sign in with Google, their credentials are automatically validated and stored in Firebase Authentication, while manually created accounts trigger additional steps where user details are systematically organized in the Firebase Realtime Database under their unique user ID. This dual-path authentication system ensures both flexibility for users and reliable data management, with Firebase handling critical security aspects like password hashing, session management, and token verification behind the scenes while maintaining a smooth user experience across all entry points to the application.

         To ensure secure and authenticated image uploads, the app implements a multi-step verification system that ties each upload to a specific user session. When a user wants to upload a landmark photo from their mobile device, the web app generates both a QR code and a unique 6-digit alphanumeric pairing code through the QRGenerator.jsx component. This QR code contains three critical pieces of information: the logged-in user's unique Firebase ID, a randomly generated session token valid for 5 minutes, and the web app's base URL for proper routing.

The mobile app scans this QR code or manually enters the pairing code, which initiates a secure handshake process with Firebase Realtime Database - the system verifies that the pairing code matches an active session associated with the user's account before establishing the connection. This dual-authentication approach (QR code + manual code entry) serves multiple security purposes: it prevents unauthorized devices from uploading to a user's account, ensures each upload is properly attributed, and maintains session integrity by expiring both the QR code and pairing code after a set period. The implementation uses Firebase's security rules to validate that only devices presenting a valid, unexpired pairing code can write to the user's designated storage path, while also logging each upload event with timestamp and device metadata for audit purposes. This system effectively balances security with usability, allowing legitimate users to seamlessly connect their devices while protecting against unauthorized access attempts.

The QR code and dynamically generated pairing code serve as a secure authentication handshake, creating a temporary, encrypted channel between the mobile device and the web application. This two-step verification system—first scanning the QR code to establish a session, then entering the time-sensitive 6-digit pairing code—ensures that only authorized devices can transmit images to the backend for landmark detection. By leveraging Firebase’s Realtime Database, each pairing code automatically expires after 5 minutes, mitigating the risk of unauthorized access.

From a user experience perspective, this approach balances convenience and security: scanning a QR code requires minimal effort (most mobile cameras natively support QR detection), while the pairing code adds a deliberate verification step to prevent accidental uploads. Architecturally, this decouples the web and mobile apps—they communicate exclusively through Firebase rather than direct API calls, allowing both platforms to function independently while maintaining data integrity. The system also logs each successful pairing attempt with metadata (timestamp, device type, and user ID), enabling audit trails for security monitoring. Importantly, this layered authentication acts as a gatekeeper before resource-intensive processes like Google Vision API calls execute, preventing abuse and ensuring only verified requests trigger backend analysis.

         For capturing the landmark and detecting its name, the app initiates a multi-step process that begins with the user pairing their mobile device through a QR code scan and entering a unique pairing code. Once connected, the app requests permission to access the device's camera, and upon approval, presents a real-time viewfinder interface with guidance to frame the landmark clearly. When the user captures the photo, they're prompted to confirm or retake the image, ensuring optimal quality for processing. The confirmed image is then securely transmitted to Firebase Storage, where it's temporarily stored with metadata including timestamp, device information, and user ID. From there, the image undergoes analysis through Google Vision API's advanced machine learning models, which employ a combination of convolutional neural networks (CNNs) and geolocation pattern recognition trained on millions of landmark images.

This system not only identifies prominent visual features but also cross-references them with geographical data to determine the most probable landmark match, along with its precise coordinates. The entire process, from photo capture to identification, typically completes within 8-12 seconds, with results displayed on the web app's dashboard alongside confidence scores indicating the accuracy of the detection. For ambiguous cases where confidence falls below 85%, the system prompts the user to manually verify or correct the suggested landmark, creating a feedback loop that improves future recognition while maintaining reliability.

The app utilizes a dedicated CameraPage**.**jsx and it starts the camera using asynchronous function named start camera which uses the getUserMedia API to enable the camera and show it on screen as well as show the live feed from the camera so the user can see how the picture of the landmark they are taking is outputted**.** When the user takes the image the asynchronous function capturePhoto is executed and that is what converts the image of the landmark to a data URL so that it can be sent to the Google Vision API for further processing and analytics to detect location and coordinates**.**

The user can either continue with the image they have taken or they have an option to replace it with a new image and have it analyzed by the system**.** To store the user inputted image and data securely the uploadBytes import is used to upload the metadata to a cloud service such as firebase and then the getDownloadURL import is  used to retrieve the data from the firebase database, this generates a publicly accessible URL to the metadata**.** The landmark is detected using the Google Vision API functions which are defined within the LandmarkLocator.js file and it uses an API key to give the latitude and longitude coordinates for a certain landmark along with the landmark’s name**.**

         Once Google Vision detects that a landmark was passed into the firebase database the backend analyzes this metadata and the Google Maps API shows the actual location pin on the map and the latitude and longitude are passed to a map component**.** In the server.cjs file, the Google Maps places API is then used to detect hotels nearby given the coordinates of the landmark and check on filters based on rating, distance from the landmark, and affordability, this data is then shown to the user on a GUI that displays a certain number of hotels**.** In the TravelPlanner.jsx file, the asynchronous function fetchData using both SkyscannerAPI and Amadeus’ API sets a certain number of flights on the screen for the user to choose from which are based on filters of pricing and timings of the flight**.** The APIs that were used to gather information about the landmarks were Google Vision API, Google Maps API, and both the Sky Scanner API and Amadeus API**.**

         Users would select a hotel and a flight to take to both reach the landmark and stay somewhere after arriving there**.** The application would show the user this data once the user saves this trip on the database to view once they return back by login or they could choose to email this summary to themselves**.** When the trip is saved the information that is shown to the user is the landmark’s name and the timestamp that the trip was saved, the image’s metadata like the URL of the image, under flight details the user would see the airline, the flight number, the route by IATA code format from a departing area to an arrival area, and the cost of the flight**.**

The hotels’ saved information would show the name of the hotel, the address of where the hotel is located, the rating of the hotel, and the cost of the hotel stay**.** The trip data is associated with the user that is currently logged in and the saved trips are stored to the userTrips collection in firebase using the asynchronous function saveTripToDB taking the trip selected data as the parameter**.**

# results

To classify an image and find out where the image was taken an Output image is taken by the device the user is connected to with the web app’s pairing code. The user does this by scanning a QR code on the main web app and then being elicited to input the pairing code from the web app to utilize their camera. The output image is then sent to firebase and extracted to be shown on the main page of the web app where the Google Vision API detects a location for it in context of a map with a location pin and coordinates for the location. This is because Google Vision API is trained on with the use of machine learning models and a large dataset for landmarks built naturally or human-created throughout the world, and it would then classify the image to the most relatable landmark probably through supervised learning when it trains on 80% data with Landmarks and their labels.

A screenshot of a qr code

AI-generated content may be incorrect.

The next step would be to use the Amadeus API which is used to get information for flights, hotels, and other accommodations through the location passed to it as input. The coordinates are sent from the Google Vision API to the Amadeus API for travel information extraction, then the React JS application would utilize GET http request and use the API link to search reference-data database for airports using the latitude, longitude, and the radius that user sets in their device’s settings. The output for this information would typically be in a dictionary or JSON format where the IATA code is returned which is why in the application we saw various different airlines that had different options to depart from the device’s current location of Atlanta (ATL) and the location of the landmark (e.g., Taj Mahal) where the arrival is set to the closest international airport in Delhi (DEL). The airlines, flight number, price, timing of flight, and the amount of stops are all outputted when the GET request is made to the Amadeus API.

For the hotel data the same location name and coordinates are passed as input to the Amadeus API for which it would search the reference-data database for hotels using the latitude, longitude, and the radius and API would extract the hotel information in dictionary or JSON format to output hotel name, the address of the hotel, and the rating of the hotel which is displayed to the user using a POST request to the API using header and body content of what should be displayed on the UI.

A screenshot of a phone

AI-generated content may be incorrect.

The user also gets an option to view the location on google maps in the situation that the user does not know how to get there, and it would allow the user to see the directions while driving, walking, through bus transit, or other options. The filters that are available for the flight data were sorted by price to filter by order of low to high or high to low and sort by travel time which allows the user to choose to only display the flights that are operating at certain times only. The filters that are available for the Hotel data were to sort by rating, distance, or affordable pricing of user’s budget.

Screenshot of a screenshot of a travel schedule

AI-generated content may be incorrect.

The user has an option to save this trip if they have selected a hotel and a flight first and that saved trip would appear to the user to revisit when they login to their account at the bottom of the dashboard screen. The addDoc function from firestore sdk is utilized so that the user can store the data database which contains the current user’s id and the savedTrips collection which stores information such as the creation date, destination name, flight details (like airline, arrival, departure), hotel details (like address, mapsLink, name, and rating), and landmark information such as landmark name and latitude, longitude coordinates.

A screenshot of a phone

AI-generated content may be incorrect.

The saved results are displayed through the SavedTrips.jsx file and it does this by calling the fetch data function which would first check all database results where the user id corresponds to that of the current user logged into the application, then it would use the JavaScript function of getDocs that would retrieve information about the saved information such as the name of landmark with the date below it then the image of the landmark is displayed, finally one container displays Flight details to the user and another container displays Hotel details to the user. The SavedTrips function in SavedTrips.jsx is the one that displays the landmark name, image, flight, and hotel details in an AccordianItem which is to show the details of the saved trip when the user clicks on the drop down menu.

A screenshot of a phone

AI-generated content may be incorrect.

# analysis of results

1. **Evaluating the Full Workflow and AI Reliability**

The foundational vision of CaughtMyEye was to transform spontaneous moments of inspiration—whether capturing a photo of an intriguing landmark while traveling or scanning a QR code on-the-go—into fully realized travel plans with minimal effort. By focusing on visual input as the primary trigger, the system aimed to eliminate the traditional friction of trip planning, where users typically juggle multiple apps and tabs to research flights, hotels, and itineraries.

Instead, the goal was to create a seamless, almost magical experience: a user snaps a photo, and within seconds, they receive a curated selection of flights, accommodations, and logistical details tailored to that location. To bring this vision to life, the application relied on a carefully orchestrated workflow combining AI-powered image analysis, geolocation services, and real-time travel data APIs. Each component had to not only function independently with high accuracy but also integrate smoothly into a cohesive user journey. For instance, the Google Vision API needed to correctly identify landmarks from varied photo conditions (e.g., different angles, lighting, or obstructions), while the Amadeus API had to deliver relevant travel options based on those coordinates.

Meanwhile, the React frontend had to present this data intuitively, with filters and sorting options to accommodate diverse user preferences. To evaluate the system’s effectiveness, we tested each phase rigorously assessing the AI’s reliability in landmark detection, the APIs’ responsiveness in fetching travel data, and the UI’s clarity in presenting options. We also examined how these components interacted under real-world conditions, such as poor network connectivity or ambiguous visual inputs, to identify bottlenecks and areas for improvement. This end-to-end analysis revealed both the strengths of the integrated approach and the challenges inherent in balancing automation with user control, speed with accuracy, and simplicity with depth of information.

The first step in this pipeline involved using the Google Vision API to process user-submitted images and identify potential landmarks. This stage was essential because if the app couldn’t reliably identify where the photo was taken, none of the travel planning features would be triggered. During our testing, the Vision API delivered high accuracy when analyzing high-resolution, well-lit images of iconic landmarks. Landmarks such as the Eiffel Tower, Burj Khalifa, and the Sydney Opera House were consistently recognized and matched with precise map coordinates.

However, we observed a significant drop in accuracy when dealing with photos taken in less-than-ideal conditions. Dimly lit or nighttime images, photos with partially obstructed views of the landmark, or images featuring less photographed or niche locations were far more challenging for the model. Recognition was also more error-prone with photos taken at odd angles or heavily zoomed-in shots. These limitations seem to reflect the nature of the Vision API’s training data, which likely consists of more traditional, well-framed, and popular landmark images. For future versions of *CaughtMyEye*, integrating a system that either prompts users for better input or allows for manual correction could help address these limitations and make the app more robust in less predictable scenarios.

1. **From Recognition to Action: Location Mapping and API Integration**

Once a landmark was successfully identified, the coordinates returned from the Vision API were used to retrieve real-time travel data, such as flights and hotels through integration with the Amadeus Travel API. While this process appeared straightforward in theory—passing coordinates to fetch relevant travel options—it introduced multiple layers of complexity that demanded meticulous error handling and data validation. To retrieve flight options, the application first had to translate the landmark's latitude and longitude into a usable airport reference by calling Amadeus's /reference-data/locations/airports endpoint, which filtered results by proximity (within a 100km radius) and returned the nearest major airport's IATA code (e.g., "DEL" for the Taj Mahal). This code then served as the destination input for the flight search API, which required additional parameters like the user's departure airport (automatically detected via geolocation or manually selected), departure date (defaulting to one week ahead to ensure results), and passenger count. The API responses, typically in JSON format, included granular details such as airline names, flight numbers, departure/arrival terminals and timestamps, layover durations, and dynamic pricing (with currency conversion handled client-side).

However, challenges arose when dealing with incomplete data—such as missing flight numbers or abruptly changed schedules—which necessitated client-side checks to filter out invalid entries and fallback mechanisms to display partial information when critical fields were absent. Similarly, hotel data retrieval relied on the same landmark coordinates but required radius-based searches (adjustable by the user) and real-time availability checks, with responses parsed to highlight key details like property ratings, distance from the landmark, and pricing tiers. Throughout this process, synchronization between the frontend and APIs had to account for latency, rate limits, and formatting inconsistencies to ensure a seamless user experience.

To make this data usable for end users, we designed an intuitive frontend interface with dynamic filtering capabilities that allowed users to sort flight options by both price (from lowest to highest or vice versa) and travel duration, including the ability to filter specifically for nonstop routes to minimize layovers. This filtering system was mirrored for hotel results, where we leveraged the same geographic coordinates from the landmark detection to query the Amadeus hotel search API, presenting users with comprehensive accommodation options that included not just basic information like hotel names and addresses, but also valuable metadata such as star ratings, guest reviews, price tiers (represented visually with dollar signs), and direct links to Google Maps locations for easy navigation.

The filtering implementation went beyond simple sorting - we created a tiered filtering system where users could first prioritize by their most important factor (such as highest-rated hotels), then further refine by secondary preferences (like maximum distance from the landmark or specific price ranges), effectively replicating the layered filtering experience found on major travel platforms. One of the most significant technical challenges emerged in coordinating these disparate data streams - the image recognition output, flight availability, and hotel listings - particularly when dealing with inconsistent API responses or remote locations with limited travel infrastructure.

When the flight API returned viable options but the hotel search came up empty (or vice versa), we implemented a robust fallback system that would: first attempt to expand the search radius automatically; then provide educational tooltips explaining the data gap; and finally present alternative nearby locations when possible, all while maintaining a cohesive user interface that prevented null states from breaking the layout.

For particularly remote landmarks where both flights and hotels were scarce, we developed a "Plan B" mode that would suggest the nearest well-serviced city along with transportation options to reach the actual destination, ensuring users always receive actionable travel advice rather than dead-end results. This comprehensive approach to data handling and user communication proved crucial in maintaining engagement, especially when dealing with the unpredictable nature of global travel data availability.

1. **Usability and Firebase Integration: Trip Saving and Retrieval**

Beyond just recognizing places and suggesting travel options, we designed CaughtMyEye to offer a seamless, personalized experience, allowing users to save trips they liked and revisit them anytime. This functionality was built using **Firebase Authentication** for secure user accounts and **Firebase Firestore** for structured data storage. When a user selected a preferred hotel and flight combination, clicking the "Save Trip" button triggered a backend process that created a new document in the Firestore database under the savedTrips collection. Each document stored critical details, including the user’s unique ID (to ensure data isolation between accounts), the landmark’s name and coordinates (for future reference), the selected flight’s details (airline, departure/arrival times, price), the chosen hotel’s information (name, address, rating, and Google Maps link), a server-generated timestamp (to sort trips chronologically), and a reference URL to the uploaded image in Firebase Storage.

To optimize performance, we implemented real-time data syncing via Firestore’s onSnapshot listener, ensuring that any newly saved trips would instantly appear in the user’s dashboard without requiring a manual refresh. When the user logged back in, the app queried Firestore for all documents matching their user ID, then dynamically rendered the trips in a visually intuitive layout. Each saved trip was displayed as a card featuring the landmark’s image and name, with expandable accordion sections—powered by react-accessible-accordion—for flight and hotel details. This design choice prioritized clarity and compactness, allowing users to quickly scan their saved trips while keeping the interface uncluttered.

Additionally, we added a delete function tied to each trip card, enabling users to remove outdated or unwanted plans with a single click, which triggered a Firestore deleteDoc operation. The integration proved robust during testing, handling concurrent saves and retrievals without latency, though we noted that users with large numbers of saved trips (50+) experienced slight delays in rendering, suggesting a need for pagination in future iterations. Overall, this system not only enhanced usability by preserving user preferences but also demonstrated Firebase’s reliability as a backend solution for travel applications requiring real-time data synchronization and secure user-specific storage.

One limitation in this area was the email-based itinerary feature, which remained incomplete. Our original intention was to let users generate a summary of their saved trip and have it emailed to them as a PDF or formatted message. However, integrating third-party email services like SendGrid or Firebase Cloud Functions turned out to be more complex and time-consuming than anticipated. This part of the feature set was scoped too late in the timeline, and we didn’t have enough time left to implement it properly. It's a clear candidate for future development and would add significant value to the trip planning experience.

1. **Technical Challenges and Timeline Reflections**

Throughout development, we faced a number of challenges that significantly shaped the project’s scope, timeline, and ultimate direction. Some hurdles were anticipated from the outset, such as API rate limits (particularly with the Amadeus test environment, which restricted us to 10 requests per minute) and data formatting inconsistencies between different services (e.g., Google Vision’s JSON responses versus Firebase’s document structure). However, other issues emerged unexpectedly during integration and testing, forcing us to pivot or rethink our approach.

For instance, the initial plan to use OpenAI’s GPT-4 for airport code generation failed due to model access restrictions, requiring a last-minute downgrade to GPT-3.5-turbo—a change that introduced minor accuracy tradeoffs. Similarly, Firebase’s permission errors, which only surfaced during multi-user testing, revealed gaps in our security rules that took days to debug. These challenges not only extended our timeline but also reinforced the importance of building flexible architectures, maintaining thorough documentation, and allocating extra buffer time for unforeseen obstacles—lessons that will inform future projects from their earliest planning stages.

One of the most persistent and impactful challenges we encountered was the system's heavy dependency on image quality, which created a critical bottleneck in the user experience. The Google Vision API, while remarkably powerful for identifying well-known landmarks under ideal conditions, demonstrated significant limitations when processing suboptimal images—particularly those that were grainy, poorly lit, or unusually cropped. This sensitivity meant that the quality of a user's initial photo input became a decisive factor in whether the system could successfully generate travel recommendations, creating frustration when blurry vacation photos or screenshots of landmarks failed to produce results.

The issue was compounded by environmental factors like nighttime photography or crowds obstructing landmarks, where even human-recognizable images sometimes returned no matches from the API. Similarly, our QR code pairing system, though generally reliable, revealed subtle but important reliability gaps during testing. While printed QR codes scanned effortlessly in good lighting conditions, we observed frequent failures with small, pixelated, or glare-covered codes—common scenarios when users attempted to scan codes from other device screens.

Certain mobile browsers, particularly iOS Safari with its restrictive camera permissions, introduced additional friction by requiring multiple authorization prompts or failing to initialize the camera altogether. These technical dependencies on image clarity and scanning conditions created inconsistent first-time user experiences that we mitigated through clearer onboarding instructions and a retry mechanism, though a more robust computer vision pipeline would be needed for truly universal reliability.

The rate limits imposed by the Amadeus API created significant constraints during development, as the free-tier service only allowed 10 requests per minute and 2,500 calls per day. This limitation forced us to implement careful request management strategies, including client-side caching frequent queries (like airport code lookups) and mock data implementations for development testing. During intensive build sessions with multiple developers working simultaneously, we frequently encounter bottlenecks where API quotas would be exhausted unexpectedly, halting progress until the rate limit reset.

To mitigate this, we established a shared testing schedule and implemented request throttling on the frontend, but these workarounds inevitably slowed our development velocity and made comprehensive testing more challenging. The situation was further complicated by the asynchronous nature of our architecture - while one developer might be testing flight searches, another could be debugging hotel queries, and both would consume from the same limited pool of API calls. These constraints highlighted the importance of either budgeting for paid API tiers in production or building more sophisticated mock services earlier in the development process.

Integrating the diverse ecosystem of services - including Google Vision for image recognition, Amadeus for travel data, Firebase for real time storage, and our React frontend for presentation - introduced numerous coordination challenges that spanned technical and operational dimensions. Each service returned data in different formats (JSON structures from Amadeus, protobuf from Google Vision, Firestore's document-based responses), requiring extensive transformation layers to normalize the data for our UI components. The asynchronous nature of these external calls created race conditions where flight data might arrive before the hotel results, or where a slow Google Vision response would delay the entire trip planning flow.

We implemented Promise.allSettled() to handle partial failures gracefully and added loading states for each service independently, but debugging these distributed interactions often required correlating timestamps across browser developer tools, Firebase logs, and API monitoring dashboards. Particularly problematic were edge cases where one service would fail silently - for instance, when the Amadeus API returned a 200 OK with an empty dataset for obscure locations - requiring us to build comprehensive validation checks at each integration point. These challenges underscored the complexity of modern web development where systems are increasingly composed of disparate cloud services, each with their own quirks and failure modes that must be orchestrated into a cohesive user experience.

Our initial plan to develop a cross-platform React Native version of the app encountered significant technical roadblocks that ultimately led us to pivot our approach. Several critical libraries we relied on for core functionality—particularly those handling image processing, Firebase authentication, and the Google Vision API integration—proved incompatible with the mobile environment or required extensive native module bridging that fell outside our project timeline.

After evaluating the substantial refactoring needed to resolve these conflicts (which would have delayed other key features), we made the strategic decision to prioritize optimizing the web application experience instead. This allowed us to focus our resources on enhancing the stability of the pairing system, improving the visual presentation of travel data, and refining the trip-saving workflow—all areas that directly impacted user experience. The tradeoff, while necessary, did limit our ability to offer native camera access and offline functionality that a mobile app could have provided. These technical constraints ultimately reinforced an important lesson about platform-specific dependencies early in the development process.

Similarly, the accelerated timeline and steep learning curve presented constant challenges that shaped our team's workflow. Mastering secure API request patterns, implementing proper token rotation for the Amadeus and Google APIs, transforming complex JSON responses into intuitive UI components, and debugging real-time synchronization between Firebase and our state management required countless iterations—often under tight deadlines.

Unexpected hurdles like API rate limiting, CORS policy conflicts, and asynchronous data flow issues consumed more development time than originally anticipated for what appeared to be "backend" tasks in our initial planning. However, these struggles proved invaluable for skill development; debugging sessions became impromptu knowledge-sharing opportunities, and the pressure to deliver forced us to adopt more efficient collaboration practices like pair programming and modular feature branching. While the compressed schedule meant sacrificing some ambitious features, the hands-on experience with full-stack troubleshooting, performance optimization, and user feedback integration ultimately delivered deeper technical growth than a perfectly smooth development process ever could have.

# conclusion statement

To classify an image and find out where the image was taken an Output image is taken by the device the user is connected to with the web app’s pairing code. The user does this by scanning a QR code on the main web app and then being elicited to input the pairing code from the web app to utilize their camera. The output image is then sent to firebase and extracted to be shown on the main page of the web app where the Google Vision API detects a location for it in context of a map with a location pin and coordinates for the location. This is because Google Vision API is trained on with the use of machine learning models and a large dataset for landmarks built naturally or human-created throughout the world, and it would then classify the image to the most relatable landmark probably through supervised learning when it trains on 80% data with Landmarks and their labels.

REFERENCE

1. D. V. Lindberg and H. K. H. Lee, “Optimization under constraints by applying an asymmetric entropy measure,” *J. Comput. Graph. Statist.*, vol. 24, no. 2, pp. 379–393, Jun. 2015, doi: 10.1080/10618600.2014.901225.
2. B. Rieder, *Engines of Order: A Mechanology of Algorithmic Techniques*. Amsterdam, Netherlands: Amsterdam Univ. Press, 2020.
3. I. Boglaev, “A numerical method for solving nonlinear integro-differential equations of Fredholm type,” *J. Comput. Math.*, vol. 34, no. 3, pp. 262–284, May 2016, doi: 10.4208/jcm.1512-m2015-0241.