**All-Serve — Full Developer Specification**

**Summary (one-sentence)**

All-Serve is a mobile-first marketplace (Flutter mobile app for customers + Flutter Web provider portal + Flutter Web admin dashboard) backed by Firebase Firestore, Storage, Auth and Cloud Functions, enabling customers in Zambia to find, book, and review verified local service providers; providers can manage profiles and bookings and optionally link to their own website; admins manually verify provider documents and moderate the platform.

**1. Tech stack & third-party services**

* Frontend (Customer mobile): **Flutter (Dart)** — Android-first (iOS optional later).
* Frontend (Provider & Admin portals): **Flutter Web** (keeps single codebase) — or React if your team prefers.
* Backend-as-a-service: **Firebase**
  + **Auth** — email/password and phone OTP (optional).
  + **Firestore** — primary data store (NoSQL).
  + **Storage** — for images and verification documents.
  + **Firebase Cloud Functions**  — booking validation, review validation, notifications, server-side logic.
  + **Cloud Messaging (FCM)** — push notifications.
  + **Hosting** — host admin/provider web if using Flutter Web.
* Geolocation: **geolocator** (client) and **geohash** or geoflutterfire for proximity queries.
* Dev tools: VS Code, Android Studio, Git/GitHub.

**2. High-level architecture**

[Customer Flutter App] [Provider Flutter Web] [Admin Flutter Web]

│ │ │

└────────────── HTTPS ────────────────┼────────────────────────────────┘

│

[ Firebase Project ]

├─ Firestore (collections below)

├─ Storage (images, docs)

├─ Auth (users/providers/admin)

├─ Cloud Functions (server logic)

└─ FCM (notifications)

All clients use the same Firebase project. Role determines which UI is shown and which operations are allowed.

**3. Roles & authorizations**

* customer — uses mobile app to search, book, review.
* provider — uses provider portal to manage profile, services, availability, accept/decline bookings.
* admin — uses admin dashboard to manually verify providers, moderate reviews/bookings, manage categories.

Assign role in the users/{uid} document after registration. Firestore rules enforce role-based access.

**4. Firestore data model (collections + detailed fields)**

All timestamps are stored as Firestore Timestamp. Use consistent field names.

**users/{uid}**

* uid (string) — same as Auth uid
* name (string)
* email (string)
* phone (string)
* role (string) — "customer" | "provider" | "admin"
* profileImageUrl (string)
* defaultAddress (map: {address, lat, lng})
* deviceTokens (array) — FCM tokens
* createdAt (timestamp)

**providers/{providerId}**

* providerId (string)
* ownerUid (string) — user id of provider owner
* businessName (string)
* description (string)
* categoryId (string)
* services (array of maps) — each {serviceId, title, priceFrom, priceTo, durationMin}
* logoUrl (string)
* images (array of strings)
* websiteUrl (string) — optional, to redirect customers
* lat (double), lng (double)
* geohash (string) — for spatial indexing
* serviceAreaKm (number)
* ratingAvg (double)
* ratingCount (int)
* verified (bool) — set by admin after manual verification
* verificationStatus (string) — "pending"| "approved"| "rejected"
* documents (map) — {nrcUrl, businessLicenseUrl, otherDocs...}
* status (string) — "active"|"suspended"|"inactive"
* createdAt (timestamp)

**categories/{categoryId}**

* name (string)
* iconKey (string) or iconUrl
* description (string)
* isFeatured (bool)

**bookings/{bookingId}**

* bookingId
* customerId
* providerId
* serviceId
* address (map address, lat, lng)
* scheduledAt (timestamp)
* requestedAt (timestamp)
* status (string) — "requested"|"accepted"|"rejected"|"completed"|"cancelled"
* notes (string)
* createdAt (timestamp)

**reviews/{reviewId}**

* reviewId
* bookingId
* customerId
* providerId
* rating (int 1–5)
* comment (string)
* flagged (bool)
* flagReason (string)
* createdAt

**verificationQueue/{id} (optional)**

* providerId
* ownerUid
* submittedAt
* status "pending"|"approved"|"rejected"
* adminNotes
* docs (map of storage URLs)

**adminAuditLogs/{id}**

* actorUid
* action (string)
* detail (map)
* timestamp

**5. Client UI pages & behavior**

I’ll describe the pages and the precise behaviors (including navigation, what to fetch, and acceptance criteria).

**5.A Customer Mobile App — main screens**

**1. Splash / WidgetTree**

* On launch check Auth.currentUser. If no user => show Login/Register. If logged in => load role from users/{uid} and route to Customer Home.

**2. Login / Register**

* Email/password; optionally phone OTP (implementing phone OTP requires Firebase Phone Auth).
* After registration, prompt for name and default address (map picker or manual).

**3. Home Screen (the design you provided)**

* Search bar (free-text)
* "View Categories" link — navigates to Categories List screen.
* "Book an Appointment Instantly" — shows 3 categories pulled randomly from categories collection (randomized each app open).
  + Implementation: client fetches all featured categories (or all categories if none featured), shuffles locally, displays first 3.
* "Suggested Providers" — interactive list based on search criteria:
  + By default: list providers matching no filter => sorted by proximity then rating.
  + Each provider card shows: logo/avatar, businessName, distance (calculated on client), ratingAvg and ratingCount, verified badge, short description, button "Book", and “Visit Website” if websiteUrl exists.
  + Tapping provider card => Provider Detail screen.
  + Tapping "Visit Website" => open provider's websiteUrl using url\_launcher or in-app WebView.

**4. Categories List screen**

* Fetch all categories (paged).
* Tap a category => Category Providers screen.

**5. Category Providers screen**

* Query providers where categoryId == selected and status == active.
* Sort: compute distance and then rating. Implementation detail below (Search algorithm).

**6. Provider Detail screen**

* Full provider profile: name, images carousel, services list, availability (read-only), contact phone (call intent), "Book Now" button, "Visit Website" link (open external).
* Reviews list (paginated).
* "Save/Favorite" toggle writes to users/{uid}.favorites.

**7. Booking Flow**

* Customer picks service, time slot, address (use default or new), optional notes.
* On "Request Booking" call Cloud Function createBooking — function validates provider active and writes booking with status: requested.
* Show a "booking requested" screen with live status updates (via Stream listening to booking doc).

**8. My Bookings screen**

* List all bookings by the current customer, showing status and allow cancellation where policy permits.

**9. Profile screen**

* Display & edit name, phone, profile image, saved addresses.
* Show booking history and review history.

**5.B Provider Web Portal (Flutter Web recommended to reuse Dart code)**

**Main screens**

1. Provider Signup / Login — after login, provider may be required to complete business profile including document uploads.
2. Provider Dashboard
   * Quick stats: pending bookings, upcoming jobs, rating.
   * Verification status: shows pending documents upload state and message from admin.
3. Profile edit screen
   * Manage business name, description, categories (one or more), add / remove services (title, price range, duration), add logo and gallery images, add websiteUrl.
   * Set location on a map (lat/lng) and serviceAreaKm radius.
4. Availability Calendar
   * Manage weekly availability and blocked dates.
   * Calendar UI to mark slots not available.
5. Bookings Management
   * Requests tab: accept/reject with optional message.
   * Accepted tab: list upcoming jobs and mark completed after service.
6. Reviews
   * View reviews and optionally reply (optional).
7. Settings: contact details, payout info (future), account status.

**Provider website link behavior**

* Providers can enter websiteUrl on their profile.
* On mobile app provider card and detail, display "Visit Website" button opening websiteUrl:
  + Use url\_launcher to open in external browser.
  + Optionally, if you prefer to keep users in-app, open a WebView with the URL (consider security & privacy).

**5.C Admin Dashboard (Flutter Web)**

**Main screens**

1. Login (admin accounts created in Auth console).
2. Verification Queue
   * List pending providers with preview of uploaded documents (images/PDF).
   * Buttons: Approve / Reject (with reason).
   * Approve sets providers/{id}.verified = true and verificationStatus = approved; log action in adminAuditLogs.
3. Providers Management
   * Search providers, change status (suspend/activate), edit categories, remove inappropriate content.
4. Reviews moderation
   * View flagged reviews (flagged by CF or users), remove or accept.
5. Announcements
   * Create broadcast message delivered via FCM.
6. Dashboard / KPIs
   * Bookings per day, provider acceptance rates, top categories.
7. Audit log & user management
   * RBAC: admin accounts have audit logs of actions.

**Manual verification process**

* Admin opens provider profile -> examines providers/{id}.documents (image URLs).
* Admin can click Approve/Reject. On Approve:
  + Set verified = true and verificationStatus = "approved".
  + Notify provider by FCM and email (if configured).
* On Reject: set status to rejected and include adminNotes; provider can re-submit.

**6. Search & matching algorithm (no AI)**

Goal: Given user input (keyword + optional category + user location), return providers ordered by:

1. Keyword relevance (provider name, business description, service titles, tags)
2. Proximity to user (distance)
3. Reputation (ratingAvg, ratingCount)

**Data prep (provider documents)**

* Each provider doc must include keywords array (lowercase tokens) computed when provider profile is created/edited. Example tokens: businessName words, category name, service titles, common synonyms.
* Include lat, lng, and geohash (geohash for spatial indexing).

**Query approach**

1. **Keyword matching**:
   * If user typed keyword, search providers using array-contains on keywords (exact token). For multi-word, split query into tokens and get union results (client-side combine). Firestore doesn’t support full-text search — consider Algolia / Meilisearch later for fuzzy search.
2. **Geospatial**:
   * Use geohash-based bounding box search (geoflutterfire) to get providers within radius. Implementation path:
     + Use client to compute bounding boxes using geohash prefixes and run where('geohash', '>=', g1).where('geohash', '<=', g2) queries.
     + Or use Cloud Function to perform server-side search (fetch all providers in bounding box and sort by distance).
3. **Sorting & ranking**:
   * After fetching candidate providers (keyword and/or nearby), compute distance (Haversine) on client or server and then sort:
     + Primary: distance (ascending)
     + Secondary: ratingAvg (descending)
     + If tie, ratingCount (descending)
   * Return top N (e.g., 10).

**Pseudocode (client)**

// 1. get userLatLng

// 2. if keyword:

resultsByKeyword = query providers where keywords arrayContains token

// 3. else:

resultsNearby = geohashQueryAround(userLatLng, radiusKm)

// 4. combine results (dedupe)

// 5. compute distance for each doc

// 6. sort by distance asc then ratingAvg desc then ratingCount desc

// 7. return top N

**Haversine distance (Dart)**

double haversine(lat1, lon1, lat2, lon2) {

const R = 6371; // km

final dLat = \_deg2rad(lat2 - lat1);

final dLon = \_deg2rad(lon2 - lon1);

final a = sin(dLat/2)\*sin(dLat/2) + cos(\_deg2rad(lat1))\*cos(\_deg2rad(lat2))\*sin(dLon/2)\*sin(dLon/2);

final c = 2 \* atan2(sqrt(a), sqrt(1-a));

return R \* c;

}

Note: For large scale and fuzzy search, integrate with Algolia/Meilisearch later. For MVP, keyword + geohash + client sorting is fine.

**7. Cloud Functions — contract & examples**

We’ll use Cloud Functions as server-side guards and helper endpoints (HTTP or callable). Keep heavy logic server-side for trustable operations (booking create/accept, review validation).

**Functions (recommended list)**

1. createBooking (callable)
   * Input: { customerId, providerId, serviceId, scheduledAt, address }
   * Server validations:
     + Provider exists and status == active and verified == true.
     + Service exists for provider.
     + No conflicting accepted booking for same slot (transactional check).
     + Write bookings/{id} with status: "requested" and notify provider (FCM).
   * Response: bookingId, status.
2. updateBookingStatus (callable)
   * Inputs: { bookingId, userUid, action } action = accept / reject / complete / cancel
   * Validations: only provider owner can accept/reject a booking for that provider; only provider or customer with correct status can mark completed/cancelled.
   * Updates booking doc; sends notifications.
3. postReview (callable)
   * Inputs: { bookingId, rating, comment }
   * Validations:
     + Booking exists and status == completed.
     + Booking.customerId == callerUid.
     + No existing review for this booking.
   * Writes review doc and updates provider's ratingAvg and ratingCount using Firestore transaction.
4. flagReview (callable)
   * Allows either user or admin to flag a review for manual moderation.
5. fetchProvidersNearby (HTTP)
   * Params: lat, lng, radiusKm, categoryId?, keywords?
   * Server computes geohashes, fetch candidates, calculates distances, sorts, returns paginated list.
   * Use if you want server-side radius filtering.
6. sendAnnouncement (admin-only callable)
   * Inputs: { title, message, audience }
   * Pushes FCM to selected audience.
7. adminApproveProvider (admin callable)
   * Inputs: { providerId, approve: true|false, notes }
   * Sets providers/{id}.verified and verificationStatus; writes to adminAuditLogs.

**Where to place functions**

* Cloud Functions folder: functions.
* Use callable functions to avoid CORS issues and easily authenticate callers.

**Example: createBooking**

exports.createBooking = functions.https.onCall(async (data, context) => {

const uid = context.auth.uid;

if (!uid) throw new functions.https.HttpsError('unauthenticated', '...');

const { providerId, serviceId, scheduledAt, address } = data;

// fetch provider

// validate provider active/verified

// check service exists

// transactionally check availability and create booking

// send FCM to provider

return { bookingId: newBookingId, status: 'requested' };

});

**8. Firestore Security Rules (examples / starting point)**

Put these in your firestore.rules. These are illustrative — adapt to your fields and collections.

rules\_version = '2';

service cloud.firestore {

match /databases/{database}/documents {

// Users - each user can read/write own doc

match /users/{userId} {

allow read: if request.auth != null;

allow write: if request.auth != null && request.auth.uid == userId;

}

// Providers - public read allowed for listing

match /providers/{providerId} {

allow read: if true;

allow create: if request.auth != null && request.auth.token.email\_verified == true && request.auth.uid == request.resource.data.ownerUid;

// Only provider owner can edit profile fields

allow update: if request.auth != null && request.auth.uid == resource.data.ownerUid;

// Only admin/CF should set verified

allow write: if false; // disallow generic writes (handle via more specific rules or functions)

}

// Bookings

match /bookings/{bookingId} {

allow create: if request.auth != null && request.resource.data.customerId == request.auth.uid;

allow read: if request.auth != null && (request.auth.uid == resource.data.customerId || request.auth.uid == resource.data.providerId);

allow update: if request.auth != null && (

// provider can accept/complete booking for their providerId

(request.auth.uid == resource.data.providerId) ||

// customer can cancel their own booking

(request.auth.uid == resource.data.customerId)

);

}

// Reviews: only Cloud Function should create reviews (or enforce booking completed check via rule)

match /reviews/{reviewId} {

allow read: if true;

allow create: if false; // created via callable function only

allow update, delete: if request.auth != null && request.auth.token.admin == true;

}

// Admin logs - admin only

match /adminAuditLogs/{id} {

allow read, write: if request.auth != null && request.auth.token.admin == true;

}

}

}

Important: For any sensitive operation (set verified, create review), prefer executing them from Cloud Functions with admin credentials (server side), not from client.

**9. Storage rules (for verification docs & images)**

Put documents under secure paths and restrict write/read:

service firebase.storage {

match /b/{bucket}/o {

match /providers/{providerId}/docs/{file} {

allow read: if request.auth != null && (

request.auth.uid == resource.metadata.ownerUid ||

request.auth.token.admin == true

);

allow write: if request.auth != null && request.auth.uid == request.resource.metadata.ownerUid;

}

match /providers/{providerId}/images/{file} {

allow read: if true;

allow write: if request.auth != null && request.auth.uid == request.resource.metadata.ownerUid;

}

}

}

When uploading, include metadata such as ownerUid to enable rules.

**10. Project file & code organization (recommended)**

/all\_serve

├─ /lib

│ ├─ /models

│ │ provider.dart, booking.dart, user.dart, review.dart

│ ├─ /services

│ │ firestore\_service.dart, auth\_service.dart, location\_service.dart, notification\_service.dart

│ ├─ /screens

│ │ auth/, customer/, provider/, admin/

│ ├─ /widgets

│ │ provider\_card.dart, category\_tile.dart, rating\_widget.dart

│ └─ main.dart

├─ /functions (Cloud Functions)

│ ├─ index.js

│ └─ package.json

├─ /web\_admin (optional separate project if desired)

├─ pubspec.yaml

└─ README.md

**11. Implementation step-by-step (developer workflow)**

**Phase A**

1. Create Firebase project and enable Firestore, Auth, Storage, Cloud Functions, Cloud Messaging.
2. Initialize Flutter project and add dependencies:
   * firebase\_core, firebase\_auth, cloud\_firestore, firebase\_storage, firebase\_messaging, geolocator, geoflutterfire (or geohash strategy), url\_launcher.
3. Create Firebase web config files (android: google-services.json, iOS: GoogleService-Info.plist).
4. Set up Firebase CLI and Functions folder.

**Acceptance**: App initializes Firebase and launches (blank home). Auth login + registration flows can create users/{uid} entries with role.

**Phase B**

1. Implement registration flows for customers and providers (provider registration includes document upload to Storage and create provider doc with verificationStatus: pending).
2. Create categories collection and admin UI to manage categories.
3. Implement provider profile editing (save lat/lng and compute geohash).

**Acceptance**: Providers can create profiles, upload docs; categories are visible in customer app.

**Phase C**

1. Implement category listing & provider listing pages.
2. Implement search widget:
   * Query by keywords (arrayContains).
   * For proximity, implement geohash queries (geoflutterfire) or client bounding box.
3. Implement provider detail screen and Visit Website link.

**Acceptance**: Customer can search text, pick category and see providers sorted by distance+rating. Click Visit Website opens external site.

**Phase D**

1. Booking form (service + slot + address).
2. createBooking callable function validating provider.
3. Provider portals show pending requests; provider can accept/reject.
4. Notifications via FCM for accepted/rejected.

**Acceptance**: Customer requests booking; provider receives notification and accepts; booking status updates in both UIs.

**Phase E**

1. After booking completed, customer leaves review.
2. postReview callable CF validates booked/completed status; update provider ratingAvg via transaction.
3. Add basic moderation flags (Cloud Function auto-flag rules).

**Acceptance**: Reviews created only after completed booking and ratingAvg reflects new rating.

**Phase F**

1. Admin UI to view verificationQueue.
2. Admin approves/rejects provider documents (Approve sets providers.{id}.verified).
3. Admin can flag/remove reviews & suspend providers.

**Acceptance**: Admin can verify providers and those providers become visible as verified: true in customer list.

**Phase G**

1. Add caching for categories and last search results (Hive).
2. Write unit and widget tests for critical components.
3. End-to-end manual tests: booking flows, provider verification, review submission.

**Acceptance**: App is stable and passes test checklist.

**12. Firestore indexes (suggested)**

Add composite indexes if necessary (Firestore may prompt). Some candidates:

* providers index: categoryId ASC, ratingAvg DESC
* providers index if you include filters: categoryId ASC, ratingAvg DESC, verified ASC
* If using server-side geohash queries, you may not need composite indexes for spatial search.

**13. Testing & QA checklist**

**Unit & Integration**

* Auth unit tests for registration/login.
* Firestore service unit tests (mock Firestore).
* Booking function tests (Cloud Functions emulator).

**Manual**

* Register as provider -> upload docs -> check admin verification workflow.
* Search by keyword & category -> inspect ordering.
* Book a service -> provider accepts -> customer sees accepted status.
* Leave review after completion -> rating updated.
* Provider website link opens correctly.

**Edge cases**

* Offline booking attempt -> queued properly.
* Duplicate review attempt -> rejected.
* Provider suspend/unverified should not appear in customer list (unless admin allows preview).

**14. Monitoring & Maintenance**

* Enable **Crashlytics** for crash reporting.
* Monitor Cloud Function invocations and Firestore read/writes (cost).
* Use **Cloud Logging** for server logs (CF).
* Regularly export Firestore backups (Firestore managed export to Cloud Storage or scheduled exports).

**15. Deployment & release steps**

**Firebase**

1. Deploy functions: firebase deploy --only functions
2. Firestore rules & indexes: upload via Firebase console or firebase deploy --only firestore:rules,firestore:indexes
3. Firebase Hosting (admin & provider web from Flutter Web builds): firebase deploy --only hosting:admin,hosting:provider after flutter build web --base-href /admin/

**Mobile**

1. Build Android debug/release: flutter build apk / flutter build appbundle
2. Test on devices/emulator, then publish to Play Store later.

**16. Security & privacy checklist (production readiness)**

* Use HTTPS all the way (Firebase uses HTTPS).
* Protect admin endpoints with request.auth.token.admin == true.
* Limit Storage access to owner/admin as shown in rules.
* Do not store verification docs publicly — set storage rules to allow admin access only.
* Rotate service account keys and store secrets in environment variables for Cloud Functions.
* Consider enabling **App Check** to prevent unauthorized clients from calling your backend.

**17. Data migration & sample seed data (example JSON for Firestore)**

// categories/doc1

{

"name": "Plumbing",

"iconKey": "plumbing",

"description": "Pipes, taps, toilets, leaks",

"isFeatured": true

}

// providers/provider1

{

"ownerUid": "uid123",

"businessName": "James Plumbing",

"categoryId": "plumbing",

"lat": -15.3875,

"lng": 28.3228,

"geohash": "s2d8...", // computed from lat/lng

"ratingAvg": 4.8,

"ratingCount": 112,

"verified": false,

"verificationStatus": "pending",

"websiteUrl": "https://jamesplumbing.example.com",

"createdAt": "<timestamp>"

}

**18. Example code snippets (key operations)**

**18.A Create provider keywords (Dart)**

List<String> buildKeywords(String businessName, String category, List<String> services) {

final tokens = <String>{};

final normalize = (String s) => s.toLowerCase().trim();

normalize(businessName).split(RegExp(r'\s+')).forEach(tokens.add);

normalize(category).split(RegExp(r'\s+')).forEach(tokens.add);

for (var s in services) normalize(s).split(RegExp(r'\s+')).forEach(tokens.add);

return tokens.toList();

}

**18.B Sample Firestore query for keyword + category (Flutter)**

final q = FirebaseFirestore.instance.collection('providers');

Query q2 = q;

if (categoryId != null) q2 = q2.where('categoryId', isEqualTo: categoryId);

if (keywordToken != null) q2 = q2.where('keywords', arrayContains: keywordToken);

final snapshot = await q2.get(); // then compute distance & sort on client

**18.C Open website URL (Dart/flutter)**

import 'package:url\_launcher/url\_launcher.dart';

Future<void> openWebsite(String url) async {

final uri = Uri.parse(url);

if (await canLaunchUrl(uri)) {

await launchUrl(uri, mode: LaunchMode.externalApplication);

} else {

throw 'Could not launch $url';

}

}

**19. Performance & cost considerations**

* Firestore charges by reads/writes. Avoid repeated get() calls: use snapshots() or cache.
* Denormalize ratingAvg, ratingCount to avoid computing on reads.
* Use pagination for provider lists.
* Image optimization & serve smaller thumbnails to reduce bandwidth.

**20. Deliverables & milestones for your supervisor**

* Week 0: Project scope & data model (this doc)
* Week 1: Basic auth + customer UI skeleton + categories + provider profile creation
* Week 2: Search & provider listing + provider website links + booking request
* Week 3: Provider portal (accept/reject) + reviews + admin verification queue
* Week 4: Polish, tests, documentation, demo & deployment to Firebase hosting (web) and APK build (mobile)

**21. Acceptance criteria (what “done” looks like)**

* Customer can register, search by category or keyword, view provider profiles, request bookings and see status updates.
* Provider can register, upload docs, add services, accept/reject bookings via web portal, and set a websiteUrl which customers can open.
* Admin can manually review pending provider documents and approve/reject — verified providers show verified on the app.
* Reviews can be left only after completed booking and update provider’s average rating.
* Search returns providers sorted by proximity and then rating (and supports keyword and category filtering).
* Security rules prevent unauthorized write access to verification flags and reviews.