**Software Requirements Specification (SRS) for Workjunction**

**1. Introduction**

**1.1 Purpose**

The purpose of this document is to provide a comprehensive, unambiguous, and detailed outline of the functional and non-functional requirements of Workjunction, a modern service-on-demand digital platform.

This SRS serves as the single source of truth for all stakeholders. It specifies:

* What the system will do.
* Who will use the system.
* How the system will behave under different conditions.
* The constraints under which the system must operate.

The SRS benefits all stakeholders in the following ways:

* **Project Managers**: Define scope, allocate resources, track progress, and evaluate deliverables.
* **Developers**: Reference requirements to design and implement system features, including dependencies such as Razorpay integration and JSON Web Token (JWT) authentication.
* **Quality Assurance Testers**: Develop test plans to ensure all requirements are met.
* **Business Stakeholders**: Understand system capabilities, limitations, and expected business value.

By detailing these requirements in advance, the SRS minimizes miscommunication, scope creep, and rework during development.

**1.2 Product Scope**

Workjunction is a web-based service marketplace that connects customers with verified service providers (workers). It aims to provide transparency, trust, and convenience in booking professional and domestic services.

The scope includes the entire service lifecycle:

* **User Onboarding**: Secure registration, login, and profile management for customers, workers, service agents, and administrators.
* **Service Discovery**: Customers can browse and search for workers based on type of service, location, reviews, and pricing.
* **Booking Management**: Workflow-based booking process with statuses such as Pending, Confirmed, In Progress, Completed, or Cancelled.
* **Worker Verification**: Aadhaar, live selfie, and police verification uploads for trust building. Verified workers become visible to customers.
* **In-App Communication**: Secure chat between customers and workers.
* **Payments**: Integration with Razorpay for both pre-service and post-service payments.
* **Administrative Oversight**: Centralized admin panel for managing agents, users, services, and reports.
* **Service Agent Oversight**: Local managers assist with onboarding, verification, and worker support.

The system will use a scalable cloud-based architecture to support future growth.

**1.3 Intended Audience**

* **Project Managers**: For planning and resource allocation.
* **Developers**: For detailed technical implementation.
* **Quality Assurance Teams**: For validation and verification.
* **Business Stakeholders/Investors**: For market alignment and ROI evaluation.
* **Operations and Support Teams**: For deployment and maintenance.

**2. Overall Description**

**2.1 Product Perspective**

Standalone cloud-native system with these subsystems:

* Customer System — register, search, book, chat, pay, feedback.
* Worker System — register, verify, set availability, manage bookings, upload portfolio.
* Service Agent System — regional onboarding, verification handling, local support.
* Admin System — global governance, user/service/agent management and reports.

**2.2 User Classes and Characteristics**

* Customers: simple UI, trust & transparency.
* Workers: profile control, verification flows, earnings view.
* Service Agents: assist non-technical workers, review documents.
* Administrators: manage global settings, fraud prevention, reporting.

**2.3 Operating Environment**

* Frontend: React.js (responsive).
* Backend: Node.js + Express (REST APIs).
* Database: MongoDB.
* File Storage: Cloudinary.
* Auth: JWT.
* Payments: Razorpay.
* Hosting: Cloud (AWS/GCP/Azure).

**3. System Features and Functional Requirements**

Each requirement below preserves your **Input** and **Output** text and adds a **Process** section that clearly explains how the system converts the input into the output. I also add common validation, errors, and data persistence notes where helpful.

**3.1 Customer Module**

**REQ-CU-001 – Registration and Login**

**Inputs:**

* Registration: Full Name, Email Address, Password, Mobile Number (for OTP verification).
* Login: Email Address, Password.
* OTP Verification: OTP sent to the mobile number.

**Process :**

1. **Registration flow**
   * User submits registration form.
   * Server validates required fields (non-empty, email format, password strength, mobile format).
   * Server checks for existing accounts with same email or mobile (uniqueness).
   * If new, server generates a random OTP and calls OTP gateway (SMS provider) to send OTP to the mobile number.
   * Server temporarily stores registration data with a pendingVerification flag and stores OTP hash + expiry.
   * User enters OTP → server verifies OTP hash and expiry.
   * On success, server hashes the password (bcrypt/argon2) and creates a new user record in database with role customer and status active.
2. **Login flow**
   * User submits email + password.
   * Server validates input format.
   * Server loads user by email; if not found → return invalid credentials.
   * Server compares password hash. If match → generate JWT (claims: userId, role, issuedAt, expiry) and return it. Also update lastLogin timestamp.
   * If account locked/suspended → return appropriate error (account status message).
3. **Security**
   * Rate-limit registration & OTP attempts.
   * Lock account after repeated failed login attempts.
   * Enforce HTTPS; JWT expiry and refresh policies.

**Outputs:**

* Registration: Successful account creation with encrypted password. A unique user ID is generated. A success message is displayed. An OTP is sent to the provided mobile number.
* Login: A JSON Web Token (JWT) is returned for a successful login. An error message is displayed for invalid credentials.

**REQ-CU-002 – Profile Management**

**Inputs:** Full Name, Email Address, Phone Number, and detailed Address (street, apartment, city, state, pincode).

**Process :**

1. User (authenticated via JWT) submits profile update request.
2. Server validates field formats (email, phone, pincode).
3. If email/phone changed, check for uniqueness and optionally trigger re-verification (OTP).
4. Server updates user document in DB, sets updatedAt timestamp, and writes audit entry.
5. If address changed, optionally trigger geocoding (to get lat/long) for location-based searches.
6. Return updated profile object.

**Outputs:** Updated profile information is stored in the database. A success message is displayed to the user.

**REQ-CU-003 – Service Search and Browsing**

**Inputs:**

* Search query (e.g., "Plumber")
* Service Type, Location (city, pincode)
* Ratings (min/max), Pricing (min/max)
* Worker Name, Worker Phone No.

**Process :**

* Client sends search API request with filters and pagination params.
* The server receives the request and, using the authentication token (JWT), **retrieves the customer's default address (street, city, pincode) from their profile data.**
* The server validates filters and sanitizes inputs to avoid injection.
* The server constructs the database query:
  + Match service type (exact or category) and skills.
  + Find workers who are geographically near the customer's retrieved address. This involves a geospatial query ($nearSphere in MongoDB) using the latitude and longitude of the customer's location and a specified search radius.
  + Filter by rating range and pricing range.
  + Support partial matches for name or phone (with rate limiting).
* The server sorts results (relevance, rating, distance from customer's location, price) and applies pagination.
* The server maps the database worker document fields to a response DTO (excluding sensitive fields).
* Return a paginated list.

**Outputs:**

* A paginated list of verified workers matching the search and filter criteria. The list includes worker names, ratings, services offered, and pricing information. An "No results found" message is displayed if no workers match the criteria.

**REQ-CU-004 – Booking History**

**Inputs:** User ID (implicit from JWT).

**Process :**

1. Authenticated request uses JWT to identify user.
2. Server queries bookings collection for records where customerId = userId.
3. Optionally join worker profile for each booking to include worker name & phone.
4. Sort results (most recent first) and paginate if needed.
5. Format booking status and relevant timestamps for UI.

**Outputs:** A list of all bookings associated with the customer's account. Each booking record includes: Service Name, Worker Details (Name, Phone No.), Booking Date and Time, and current Booking Status.

**REQ-CU-005 – Ratings and Reviews**

**Inputs:** Booking ID, Rating (e.g., 1-5 stars), Review Text.

**Process :**

1. Verify booking exists and is associated with the authenticated customer.
2. Check booking status is Completed (only completed services can be reviewed) and that review window is within allowed days.
3. Validate rating range and sanitize review text (remove profanity).
4. Save review record (customerId, workerId, bookingId, rating, reviewText, timestamp).
5. Recalculate the worker's average rating (aggregate query: avg of ratings, update worker document avgRating, ratingCount). Use database transaction or eventual consistency approach to avoid race conditions.
6. Notify worker about the new review.

**Outputs:** The new rating and review are stored and associated with the worker's profile. The worker's average rating is updated. A success message is displayed.

**REQ-CU-006 – In-App Chat**

**Inputs:** Sender User ID, Receiver User ID, Message Content (text).

**Process :**

1. Authenticate sender via JWT; verify the sender has permission to message the receiver (e.g., they share a booking or connection).
2. Validate message content (length, no malicious payload).
3. Persist message in messages collection with fields: senderId, receiverId, content, timestamp, readStatus (false).
4. Push notification to receiver (websocket or push) if online; update unread count.
5. Optionally run message moderation or store attachments in Cloudinary with signed URLs.

**Outputs:** The message is sent to the recipient.

**REQ-CU-007 – Multi-Language Support**

**Inputs:** Language preference (e.g., 'en' for English, 'hi' for Hindi).

**Process :**

1. User sets a language preference in their profile or browser settings.
2. Server (or client) loads localized resource bundles for the selected language.
3. API responses that include UI strings are provided in the chosen language (or keys are returned and frontend renders text).
4. Save preference in user profile for subsequent sessions.
5. Fallback to English for missing translations.

**Outputs:** The user interface text and labels are rendered in the selected language.

**3.2 Worker Module**

**REQ-WO-001 – Worker Registration**

**Inputs:** Full Name, Email Address, Password, Mobile Number (for OTP verification), detailed Address, Skills, Services Offered.

**Process :**

1. Worker submits registration form.
2. Server validates fields, checks uniqueness for email/phone.
3. Send OTP for mobile verification and store pendingVerification state.
4. Create worker account with status Unverified. Store skills/services as structured array objects.
5. Optionally collect initial expected pricing or hourly rates.
6. Provide next-step instructions for verification document upload.

**Outputs:** A new worker account is created. The worker's status is set to 'Unverified'. An OTP is sent to the mobile number.

**REQ-WO-002 – Profile Management**

**Inputs:** Full Name, Email, Phone, Address, Skills, Services, and Pricing details.

**Process :**

1. Authenticated worker requests update.
2. Validate formats, ensure pricing is numeric and in allowed range.
3. If changing email/phone, trigger re-verification steps.
4. Save updates and update updatedAt timestamp.
5. Send confirmation notification.

**Outputs:** Updated profile information is stored. A success message is displayed.

**REQ-WO-003 – Availability Management**

**Inputs:** Availability status (e.g., 'Available', 'At Work', 'Off Duty'), specific dates/times.

**Process :**

1. Worker sets availability via UI (day slots, hours, one-off availability).
2. Validate timeslot format and conflicts.
3. Update worker availability in DB with timezone-aware timestamps.
4. Update search visibility and booking acceptance logic (only show workers available for requested slot).
5. Optionally integrate calendar sync (Google/Outlook) if worker opts-in.

**Outputs:** The worker's availability is updated in the database. This status affects their visibility to customers.

**REQ-WO-004 – Service Request Queue**

**Inputs:** Booking ID, Action (e.g., 'Accept', 'Decline').

**Process :**

1. Worker views queued booking requests in UI.
2. On Accept: system verifies the booking is still pending and the worker is available; then updates booking status to Confirmed, sets timestamps, sends notification to customer, and blocks that timeslot.
3. On Decline: system records reason (optional), sets booking status to Cancelled or Declined, and notifies customer.
4. Update worker's response metrics and performance dashboard.

**Outputs:** The booking status is updated to 'Accepted' or 'Declined'. A notification is sent to the customer.

**REQ-WO-005 – Service Portfolio/Gallery**

**Inputs:** Image files (e.g., JPG, PNG).

**Process :**

1. Worker uploads images via the UI (multipart form).
2. Server validates file type, size, and runs basic checks (image dimensions).
3. Upload files to Cloudinary (secure, signed upload or server-side upload).
4. Cloudinary returns secure URLs; server stores the URLs and metadata (public\_id, size, createdAt) in worker profile gallery.
5. Thumbnails/optimized versions generated by Cloudinary for responsive display.

**Outputs:** The image files are uploaded to Cloudinary. The URLs are stored in the worker's profile, making them visible to customers.

**REQ-W-006 – Multi-Language Support**

**Inputs:** Language preference.

**Process :** Same pattern as customer multi-language; worker UI text changes and preference saved in profile.

**Outputs:** The worker interface is rendered in the chosen language.

**3.3 Worker Verification Module**

**REQ-WV-001 – Document Upload**

**Inputs:** Aadhaar Card image/PDF, Live Selfie image, Police Verification Certificate image/PDF.

**Process :**

1. **Document types and expectations**
   * **Aadhaar Card**: clear front image or PDF showing Aadhaar number & name; system masks portions when displayed to other users; used for identity verification.
   * **Live Selfie**: real-time selfie capture or recent clear photo used for face-match against Aadhaar photo; anti-spoof checks recommended (liveness detection).
   * **Police Verification Certificate**: scanned certificate or document issued by local police authority; proof of background clearance.
2. **Upload flow**
   * Worker uploads each document via secure multipart request.
   * Server validates file type (jpg, png, pdf), size limits, and runs basic OCR / format checks (optional).
   * Optionally perform face-match: send selfie + Aadhaar photo to face-compare service to compute similarity score.
   * Store documents in Cloudinary with restricted access (private bucket / signed URLs). Save document metadata (file URL, file type, upload timestamp, uploaderId) in DB.
   * Set worker verification status to Pending.
   * Trigger notification to assigned Service Agent and Admin for review.
3. **Privacy & Security**
   * Encrypt sensitive metadata stored in DB.
   * Mask sensitive identifiers in UI for agents (show only required fields).
   * Retain documents per data-retention policy and enable worker-initiated deletion workflows where required.

**Outputs:** The documents are securely uploaded to Cloudinary. The worker's verification status is set to 'Pending'. Notifications are sent to assigned Service Agent.

**REQ-WV-002 – Verification Status**

**Inputs:** N/A (this is a system state).

**Process :**

1. System tracks verificationStatus field for each worker (Unverified, Pending, Verified, Rejected).
2. Status transitions:
   * Unverified → on document upload → Pending.
   * Pending → after Agent/Admin review → Verified or Rejected.
3. When Agent approves, server records approverId, approvalTimestamp and sets verifiedAt timestamp and visibleInSearch=true.
4. When rejected, server stores rejectionReason and notifies worker with steps for resubmission.
5. Maintain audit trail of each state change (who changed it and when).

**Outputs:** The worker's profile status changes based on the review process. The possible outputs are: 'Unverified', 'Pending', 'Verified', or 'Rejected'.

**REQ-WV-003 – Visibility Control**

**Inputs:** Worker's verification status (e.g., 'Verified').

**Process :**

1. Search and listing APIs include verified=true filter unless admin overrides.
2. When verificationStatus == 'Verified' → worker document has visibleInSearch = true and profile displays verification badges.
3. If not verified → either hide the worker or show limited profile with a note "Verification pending", depending on business rule.
4. Update caching / search indices when verification status changes.

**Outputs:** If the status is 'Verified', the worker's profile is included in search results and visible to customers. Otherwise, it is hidden.

**3.4 Booking Module**

**REQ-SB-001 – Worker Matching**

**Inputs:** Customer's Location, Requested Service Type, Customer's Booking Date and Time.

**Process :**

1. Validate requested time slot and service type.
2. Query available workers who:
   * Offer requested service type.
   * Are verified=true.
   * Have availability for requested date/time.
   * Are within acceptable distance from customer's location (geospatial filter).
3. Rank results by relevance: proximity, rating, response time, price.
4. Return top N workers. If none found, optionally expand radius or show agents.

**Outputs:** A list of available and verified workers who are located near the customer and offer the requested service.

**REQ-SB-002 – Booking Workflow**

**Inputs:**

* Customer: Service Type, Worker Selection, Timeslot, Location.
* Worker: Action ('Accept' or 'Decline').

**Process :**

1. **Create Booking (Customer action):**
   * Validate inputs and reserve the timeslot tentatively (create booking with status Pending, generate bookingId).
   * Notify worker with booking request details (time, location, price, contact).
2. **Worker Response:**
   * Worker accepts → verify availability and update booking status = Confirmed, record acceptedAt, and block timeslot in worker calendar. Notify customer.
   * Worker declines → set status = Cancelled (or Declined), record reason, and notify customer. Optionally re-run matching to find other workers.
3. **In-Progress & Completion:**
   * When worker starts service → mark In Progress with timestamp.
   * After service ends → worker or customer marks Completed. Capture actualAmount and allow customer to confirm before payment settlement.
4. **Cancellation & Refunds:**
   * If booking cancelled by customer or worker before service → enforce cancellation policy and process refunds if payment made (via Razorpay).
5. **Notifications & Audit:**
   * All status changes are pushed to both parties and stored in booking history.
6. **Edge cases:**
   * Double-booking prevention via transactions or calendar locks.
   * No-show handling with grace period and penalties.

**Outputs:**

* Initial Request: A new booking record is created with 'Pending' status. A notification is sent to the worker.
* Worker Acceptance: The booking status is updated to 'Confirmed'. A notification is sent to the customer.
* Worker Decline: The booking status is updated to 'Cancelled'. A notification is sent to the customer.

**3.5 Payment Module**

**REQ-PM-001 – Razorpay Integration**

**Inputs:** Payment Amount, Customer Payment details (e.g., credit card info, UPI ID), Booking ID.

**Process :**

1. Client creates payment request for bookingId and amount.
2. Server creates Razorpay order through Razorpay API (server-to-server), storing RazorpayOrderId and amount in DB.
3. Client uses Razorpay checkout or UPI flow to complete payment (tokens handled by Razorpay).
4. Razorpay sends webhook/callback to server on payment success/failure. Server validates webhook signature and updates payment record accordingly.
5. On success, update booking payment status and create transaction entries (for worker payout, platform commission). Trigger invoices and receipts.

**Outputs:** A secure payment session is initiated with Razorpay. Upon completion, a payment success or failure status is returned to the Workjunction system.

**REQ-PM-002 – Payments**

**Inputs:**

* Fixed: Service ID, Pre-defined price.
* Variable: Booking ID, Final amount agreed upon after service.

**Process :**

1. If fixed price: fetch service default price and generate Razorpay order.
2. If variable price: ensure final amount is confirmed by both parties before generating payment request.
3. Apply taxes, discounts, coupons, and platform commission rules.
4. Create ledger entries: customer charge, platform commission, worker's net payable.
5. Handle partial payments, advance payments, or escrow as per policy (hold funds until completion if required).

**Outputs:** A payment request is created and sent to Razorpay with the correct amount.

**REQ-PM-003 – Payment Status Tracking**

**Inputs:** Payment ID from Razorpay.

**Process :**

1. On Razorpay webhook, server validates signature and maps Razorpay paymentId to local Payment record.
2. Update status to Unpaid, Processing, Paid, or Failed. Attach metadata (paymentMethod, payer details, timestamps).
3. Trigger downstream actions: release payout to worker, email receipt to customer, update booking status to Paid if payment required pre-service.

**Outputs:** The payment status in the database is updated to 'Unpaid', 'Processing', 'Paid', or 'Failed' based on the Razorpay callback.

**3.6 Admin and Service Agent Modules**

**Admin Module**

**REQ-AD-001 – Manage Service Agents**  
**Inputs:** Service Agent Name, Email, Password, Assigned Area.

**Process :**

1. Admin creates agent profile; system validates email and area uniqueness.
2. Generate credentials and send invitation email (with secure password setup link).
3. Store agent with assigned area and permissions.
4. Track agent performance metrics (onboarding rate, verification turnaround).

**Outputs:** A new Service Agent account is created, updated, or suspended.

**REQ-AD-002 – Global User Management**  
**Inputs:** User ID, Action (e.g., 'Deactivate', 'Suspend', 'Delete').

**Process :**

1. Admin issues management action.
2. Server validates action (prevent deleting admin or critical accounts without two-step approval).
3. Apply action: deactivate sets isActive=false, suspend sets suspendedUntil, delete marks soft-delete or hard-delete based on policy.
4. Notify user and record audit trail.

**Outputs:** The specified user's status is updated.

**REQ-AD-003 – Service Listing and Updation**

* **Inputs:**
  + Category details (name, description).
  + Subcategory details (name, description, parent category).
  + Service details (serviceId, name, description, defaultPrice, estimatedDuration, requiredSkills).
  + Admin actions (Create, Update, Delete).
* **Process:**
  + **Admin creates or updates hierarchical categories** in the form of *Category → Subcategory → Service*.
  + Each service record includes:
    - serviceId (unique identifier)
    - name (service title)
    - description (short detail of service)
    - defaultPrice (base cost)
    - estimatedDuration (time estimate)
    - requiredSkills (skills needed for worker).
  + **System validates uniqueness** of categories and services to prevent duplicates.
  + **Database is updated** with new or modified service records.
  + **Search index and frontend menus are refreshed** so customers can easily browse and filter services.
  + **Cache invalidation is triggered** to ensure updated service data is visible immediately across the platform.
* **Outputs:**
  + A structured, hierarchical list of categories, subcategories, and services available for customers to browse.
  + Updated search filters and frontend menus reflecting the latest changes.
  + Confirmation message to Admin for successful creation, update, or deletion.

**Service Agent Module**

**REQ-SA-001 – Worker Management**  
**Inputs:** Worker ID, Action (e.g., 'Assist Onboarding', 'Handle Request').

**Process :**

1. Agent selects worker and performs allowed actions (edit profile fields, guide through verification steps).
2. Actions are validated and saved. Agents cannot change critical identity fields without admin approval.

**Outputs:** The Service Agent can access and modify a worker's profile details and assist with their requests.

**REQ-SA-002 – Verification Handling**  
**Inputs:** Worker ID, Document URLs, Action ('Approve' or 'Reject'), Rejection Reason (if applicable).

**Process :**

1. Agent reviews uploaded documents (Aadhaar, selfie, police certificate) in a secure viewer.
2. If Approve: set worker verificationStatus='Verified', update visibility and record approverId and timestamp.
3. If Reject: set verificationStatus='Rejected', store rejectionReason, and notify worker with instructions.
4. After 1month selfie is recheck
5. All decisions logged for audit.

**Outputs:** The worker's verification status is updated to 'Verified' or 'Rejected'. A notification with the result is sent to the worker.

**REQ-SA-003 – Worker Request Handling**  
**Inputs:** Worker ID, Booking ID, Action ('Accept' or 'Decline').

**Process :**

1. Agent acts on behalf of worker where permitted (e.g., worker is not tech-savvy).
2. System verifies agent's authority for the worker/area.
3. Perform accept/decline flow and update booking status and notify customer.

**Outputs:** The booking request status is updated on behalf of the worker.

**REQ-SA-004 – Area Statistics**  
**Inputs:** N/A (system-generated data).

**Process :**

1. System aggregates metrics for assigned area: verified workers count, completed bookings, pending verifications, average response times.
2. Metrics computed from bookings, workers, and agent logs using scheduled batch jobs or real-time aggregation.
3. Populate agent dashboard with charts, lists, and alerts (threshold-based notifications).

**Outputs:** A dashboard showing key metrics for the assigned area, such as number of verified workers, completed bookings, etc.

**5. Worker Verification Workflow — expanded & process-oriented**

**High-level Steps:**

1. **Registration:** Worker registers → verificationStatus = Unverified.
2. **Document Upload:** Worker uploads Aadhaar, Live Selfie, Police Verification Certificate. (See REQ-WV-001 for details & data stored.)
3. **Notification:** System notifies assigned Service Agent and Admin (push + email).
4. **Review:**
   * **Agent Review:** Agent inspects documents, checks face-match, and confirms authenticity.
   * **Admin Oversight:** Admin may spot-check or review flagged cases (random or high-risk).
5. **Decision:**
   * **Approve →** Verified → worker becomes visible to customers and gets verification badge.
   * **Reject →** Rejected with rejectionReason. Worker gets message with steps for re-upload.
6. **Post-Verification:** On Verified, worker can appear in search; platform schedules payouts and rating history starts collecting.

**Document descriptions (more descriptive as requested):**

* **Aadhaar Card (Identity Document):** Must be a clear front-side image or PDF. System requires readable Aadhaar number and name. Used for identity verification and must be masked when displayed publicly.
* **Live Selfie (Liveness & Face Match):** Recent selfie taken within the app or uploaded with liveness proof. Used to compare against Aadhaar photo using face-match services to ensure the person is genuine. High face-match threshold required for auto-approval.
* **Police Verification Certificate (Background Check):** Issued by local police authority or authorized background-check partner. Must include issuing authority, validity date, and worker name. Used to confirm no criminal record.

**Resubmission & Appeals:**

* If documents rejected, worker receives clear rejectionReason and allowed to resubmit. For disputes, worker can appeal to Admin with extra documents or in-person verification.

**Audit & Retention:**

* All decisions logged with agentId/adminId, timestamps, and versioned document history. Retention policy states documents stored for X years (configurable). Access controlled and encrypted

**5. Non-Functional Requirements**

**5.1 Performance**

* 10,000 concurrent users, API < 2s response time.
* Search < 3s for 95% of cases.
* Payments < 5s excluding gateway delays.

**5.2 Scalability**

* Horizontal scaling on cloud infra.
* MongoDB sharding & replication.
* Load balancing across backend instances.

**5.3 Availability & Reliability**

* SLA: 99.9% uptime.
* Auto-recovery with Kubernetes.
* Daily backups (RTO = 2h, RPO = 15m).

**5.4 Security**

* TLS 1.2+, JWT-based auth.
* Password hashing with bcrypt/argon2.
* Document encryption at rest.
* RBAC enforced for all roles.
* Audit logs maintained.

**5.5 Usability**

* Responsive UI, multi-language support.
* Accessibility: WCAG 2.1 AA compliance.

**5.6 Maintainability**

* Modular code, 12-Factor principles.
* Automated CI/CD with ≥ 80% test coverage.

**5.7 Compliance & Privacy**

* Indian IT Act, Aadhaar handling guidelines.
* GDPR-style consent & data deletion policies.

**5.8 Disaster Recovery**

* Multi-zone deployment.
* Incident response plan with escalation SLAs.