SMART TOURISM GUIDE

Backend Architecture Design Document

Spring Boot Modular Monolith — v1.0

Project Team

Enes Can Bilgiç

Akif Karakoç

Abdulkadir Kırıcı

Computer Engineering — Graduation Thesis

İzmir, 2025

Table of Contents

1. Project Overview

Smart Tourism Guide is an AI-powered mobile and web application designed to enhance the tourism experience in İzmir, Turkey, with planned expansion to neighboring cities including Manisa, Aydın, and Muğla. The system combines location-based services, gamification, conversational AI, and computer vision into a unified platform.

This document defines the architecture, module structure, technology decisions, and implementation guidelines for the Spring Boot backend. It is the single source of truth for all backend development decisions made by the team.

1.1 Core Features

- Nearby place discovery powered by Google Places API with intelligent caching
- Quest system with GPS and photo-based verification for gamified exploration
- User-created ToDo lists for personal trip planning
- Predefined routes in specific areas with radius-based suggestions
- AI chatbot with hybrid RAG architecture for travel recommendations
- Computer vision for landmark identification and quest verification
- Role-based admin dashboard for content management and moderation
- Push notification system via Firebase Cloud Messaging

1.2 Pilot Region

The initial deployment targets İzmir, Turkey. The computer vision and landmark recognition features are restricted to pilot zones where sufficient training data exists. This constraint is enforced at the API level via coordinate boundary checks.

1.3 Scope Decisions

The following items were explicitly evaluated and excluded from the current scope:

- Foursquare API integration — Google Places API covers all required functionality. Maintaining two external place APIs adds deduplication complexity and dual cost without clear user value.
- Background push notifications for iOS — iOS restricts background location tracking severely. This feature is architecturally planned but deferred. In-app banners cover the use case for MVP.

2. System Architecture

2.1 Architectural Pattern

The backend is a modular monolith deployed as a single Spring Boot application on port 8080. Modules communicate directly via service-layer calls rather than HTTP. The sole exception is the Python AI Microservice, which is a separate process communicating over HTTP REST.

This pattern was chosen over a microservices architecture for the following reasons:

- Team size is 3 developers — microservices introduce distributed system complexity that is not justified at this scale
- Shared PostgreSQL schema simplifies data consistency without distributed transactions
- Deployment and debugging are significantly simpler for a graduation project context
- The modular package structure allows future extraction into microservices if required

2.2 Component Overview

Component	Technology	Responsibility
API Gateway	NGINX	SSL termination, reverse proxy, rate limiting, CORS
Core Backend	Spring Boot 3.x	Business logic, REST API, security, data access
AI Microservice	Python FastAPI	LLM inference, RAG pipeline, image recognition
Primary Database	PostgreSQL 15 + PostGIS	Persistent data storage, spatial queries
Cache Layer	Redis 7.x	Session cache, geohash cache, chat context
Object Storage	MinIO	User-uploaded images, place photos
Monitoring	Prometheus + Grafana + Loki	Metrics, dashboards, log aggregation
Push Notifications	Firebase Cloud Messaging	Mobile push notifications

2.3 Client Applications

- Flutter Mobile App (iOS/Android) — primary tourist-facing interface
- React Web App — admin dashboard only (Superadmin, Content Editor, Moderator)

The React web app does not serve tourist users. Tourist web access, if required in the future, should be added as a separate frontend module.

2.4 External APIs

API	Usage	Notes
Google Places API	Nearby place discovery, place details	Cache aggressively, 30-day TTL max per ToS
Google Cloud Vision API	Fallback image recognition	Used only when custom model confidence is low
Firebase Cloud Messaging	Push notifications	Architecturally included, iOS background limits apply

2.5 Google Places API — Terms of Service Compliance

Google Places API Terms of Service restrict permanent caching of certain data. The following strategy ensures compliance:

- Place coordinates, name, and category: stored permanently in the places table as our own enriched record
- Google-owned rich content (photos, reviews, real-time hours): cached with 30-day TTL, re-fetched on expiry
- Our enriched content (Turkish descriptions, quest links, route notes): stored permanently, not subject to Google ToS

Academic project note: Google does not enforce ToS against academic deployments. However, the TTL-based approach is production-ready and should be documented in the project report as a deliberate architectural decision.

3. Module Design

Each module is self-contained within its package. Modules expose their functionality through @Service classes. Direct repository access across module boundaries is prohibited — cross-module data access must go through the owning module's service layer.

3.1 Auth Module

Responsibility

Handles user registration, login, JWT token generation and validation, and role-based access control.

Key Design Decisions

- JWT tokens contain the user role as a claim. Every request is validated by JwtFilter before reaching any controller.
- The users table stores role as an enum (TOURIST, MODERATOR, CONTENT_EDITOR, SUPERADMIN). A separate roles table is unnecessary overhead for this role model.
- Superadmin account is protected at the database level via a non-deletable constraint, not just application-level code. Application-level checks alone are insufficient.
- Frontend role checks (React menu visibility) are UX-only. Security is enforced exclusively at the Spring Security layer.

Security Endpoint Mapping

Path Pattern	Allowed Roles
/api/v1/auth/**	PUBLIC
/api/v1/places/**	TOURIST
/api/v1/quests/**	TOURIST
/api/v1/chat/**	TOURIST
/api/v1/admin/superadmin/**	SUPERADMIN
/api/v1/admin/editor/**	SUPERADMIN, CONTENT_EDITOR
/api/v1/admin/moderator/**	SUPERADMIN, MODERATOR

3.2 User Module

Responsibility

Manages user profiles, EXP and leveling system, and favorite places.

Key Design Decisions

- EXP points and level are stored directly on the users table (exp_points, level columns). A separate gamification table is not justified — these are simple integer fields updated on quest completion.
- Favorites are a junction table (user_id, place_id, saved_at). Favorite management lives in this module, not the place module, because favorites are a user-centric concept.
- User location history (user_location_history table) is write-only from the application perspective. It is populated for analytics and logging purposes only. No business logic reads from it — adding such logic would introduce unnecessary complexity.

3.3 Place & Location Module

Responsibility

Manages place discovery, Google Places API integration, caching strategy, and spatial queries.

Cache Strategy

- On each user location update, a Geohash (precision 6, ~1.2km cells) is calculated.
- Redis is checked first: key pattern query:geohash:{hash}. Cache hit returns results with <10ms latency at zero API cost.
- Cache miss triggers Google Places API call. Results are saved to PostgreSQL (permanent) and Redis (TTL: 3600 seconds).
- Location updates are only processed if the user has moved more than 100 meters OR 1 minute has elapsed. This prevents redundant cache lookups on minor GPS drift.

Spatial Queries

PostGIS ST_DWithin is used for all proximity queries (nearby places, route radius checks). Standard SQL distance calculations are not used — they are both slower and less accurate for geographic coordinates.

Data Ownership

Google Places data is mapped into our own places schema on first retrieval. The external_id and source_api columns maintain traceability to the original source. This decouples our system from Google's data model — if Google changes an ID or removes a listing, our internal references remain intact.

3.4 Quest Module

Responsibility

Manages admin-created quests, user quest progress, and the three-layer verification system.

Three-Layer Verification

- Layer 1 — GPS Check: User must be within 50 meters of the target location. If not, reject immediately without proceeding to image analysis.
- Layer 2 — Image Recognition: Photo is sent to Python microservice /vision/verify endpoint with the target place as context. Each quest step has a configurable confidence threshold (set by Content Editor, default 0.80, range 0-1). If confidence >= threshold, auto-approve. If confidence < threshold, push to moderator queue.

- Layer 3 — Moderator Review: Moderator sees the photo and location data, manually approves or rejects. On rejection, user is notified and may resubmit.

Context-Aware Classification

The image recognition model receives both the photo and the quest target location. Because the user has already selected a specific quest, the model only evaluates against the expected landmark rather than performing open-world classification. This significantly increases accuracy and is referred to as context-aware classification in the project report.

Scope Limitation

The system cannot definitively determine whether a user was physically present — a user could submit an internet-downloaded photo. The system detects whether the correct landmark is visible in the photo, not whether the user took it. This limitation is acknowledged and accepted. The GPS pre-check adds a meaningful barrier. Full anti-spoofing is out of scope for this project.

3.5 ToDo Module

Responsibility

Manages user-created personal travel lists. No verification, no EXP — purely organizational.

Key Design Decisions

- Simple CRUD operations. No external dependencies.
- Users can attach a place reference (place_id FK) or create free-text items without a place reference.
- Chatbot can call ToDo service methods via LangChain function calling — users can say 'add this place to my list' in the chat interface.

3.6 Route Module

Responsibility

Manages admin-created routes and provides radius-based route suggestions.

Two Interaction Patterns

- Passive discovery: When a user enters a route's radius, the system triggers a suggestion. This is delivered as an in-app banner (not a push notification — see Section 3.11). The radius check uses PostGIS ST_DWithin against the route's center coordinate.
- Active discovery via chatbot: When a user asks the chatbot for route recommendations in an area, the chatbot queries the route service and returns matching routes. The user does not need to be physically present.

EXP Integration

Completing a full route awards EXP. Individual point-based EXP (user enters a GPS zone) is handled by the Quest module, not the Route module. These are distinct concepts and must not be conflated.

3.7 Chat Module

Responsibility

Manages chatbot sessions, message persistence, and communication with the Python AI microservice.

Hybrid RAG Architecture

The chatbot uses a hybrid approach combining retrieval-augmented generation with the LLM's general knowledge:

- Intent detection first classifies the user query. Place/route/quest queries trigger database retrieval. General travel knowledge queries use the LLM directly.
- For database-backed responses, relevant records are retrieved and injected into the LLM prompt as context. The LLM generates a natural language response from this context.
- This prevents hallucination on local data (the LLM only states what is in our database for place-specific queries) while maintaining fluency for general questions.

Conversation History Strategy

Active sessions use a sliding window + Redis approach:

- While a session is active, conversation history is stored in Redis (key: chat:session:{id}). Redis provides fast read/write with no database I/O on each message.
- The last 10 messages are included in each LLM context window. Older messages are truncated — they are not summarized in MVP. Summarization requires an extra LLM call and adds latency.
- On session end, the full conversation is persisted to PostgreSQL (chat_sessions, chat_messages tables) for permanent storage and admin review.

LLM Strategy — Current and Future

Current implementation uses Llama 3 8B via Ollama running locally on the development machine. The Python microservice abstracts this — Spring Boot is unaware of which model is running.

Future migration to a fine-tuned model: The team will fine-tune Llama 3 8B on İzmir tourism data and Turkish-language content. This constitutes 'our own model' as required by the academic supervisor. The fine-tuning process and Spring Boot integration are entirely within the Python microservice boundary. No changes to Spring Boot are required for this migration.

Multilingual Support

The system supports Turkish, English, and German. Llama 3 8B is multilingual but Turkish performance is weaker than English. Fine-tuning with Turkish-weighted data mitigates this. Language detection is handled on the Python side.

3.8 Image Recognition Module

Responsibility

Handles image upload, validation, storage, and orchestration of the recognition pipeline.

Unified Model Architecture

A single computer vision model serves both use cases:

- Landmark identification (/api/v1/images/identify): User submits a photo. The model returns the most likely landmark with a confidence score. Used for tourist curiosity — 'what is this?'

- Quest verification (/api/v1/images/verify): User submits a photo with a target place context. The model returns whether the target landmark is present with a confidence score. Used by the Quest module for verification.

Both endpoints call the same Python model. The difference is the request payload and response interpretation, not the underlying model.

Three-Layer Fallback for Recognition

- Layer 1 — Custom Model: Our fine-tuned model runs inference. If confidence >= threshold, return result.
- Layer 2 — Google Cloud Vision API: If custom model confidence is below threshold or the service is unavailable, fall back to Google Cloud Vision. This adds cost but maintains availability.
- Layer 3 — Moderator Queue: If Google Vision also fails or returns low confidence, route to manual moderator review.

Pilot Zone Enforcement

Image recognition is only available within defined pilot geographic zones. The ImageController checks the user's submitted coordinates against active pilot zones before processing. Outside zones, a clear error response is returned: 'This feature is not yet available in your region.'

Image Storage

Uploaded images are stored in MinIO object storage, not the filesystem. MinIO uses the S3-compatible API, making future migration to AWS S3 trivial. File metadata (URL, uploader, upload time) is stored in the place_images table.

3.9 Review Module

Responsibility

Manages user reviews for places and moderator review queue.

Key Design Decisions

- Reviews are submitted with a status of PENDING by default.
- Moderators see all PENDING reviews in their dashboard and can APPROVE or REJECT.
- Only APPROVED reviews are visible to tourist users.
- avg_rating on the places table is updated via a database trigger or scheduled job after review approval — not on every review submission, to avoid performance overhead.

3.10 Badge Module

Responsibility

Manages badge definitions and user badge awards. Badges are created and managed by Content Editors, not hardcoded. This allows the badge catalog to evolve without code changes.

Key Design Decisions

- badges table stores badge definitions: id, name, description, icon_url, is_active. Content Editors can create, update, and soft-delete badges via the admin panel.

- user_badges is a junction table: user_id, badge_id, earned_at. A unique constraint on (user_id, badge_id) ensures each badge can only be earned once per user.
- Each quest optionally references a badge_id (nullable FK). When a quest is completed, QuestService checks if a badge is attached and awards it via BadgeService. BadgeService ignores duplicate awards silently — it does not throw an error if the user already has the badge.
- Badge management lives in its own module. QuestService calls BadgeService — never BadgeRepository directly. This respects the cross-module access rule.

Database Schema Addition

Two new tables are required:

- badges: id (UUID PK), name, description, icon_url, is_active, created_at, updated_at
- user_badges: id (UUID PK), user_id (FK), badge_id (FK), earned_at — UNIQUE(user_id, badge_id)
- quests table: badge_id (UUID FK, nullable) — added as an optional column

3.11 Notification Module

Responsibility

Manages in-app and push notifications for quest results, route suggestions, and system events.

Key Design Decisions

- Firebase Cloud Messaging (FCM) handles push delivery to both iOS and Android.
- A notifications table persists all notifications for in-app inbox display, regardless of whether FCM delivery succeeded.
- Users can manage notification preferences (opt-in/out per notification type).

iOS Limitation

iOS severely restricts background location tracking. Route radius-based suggestions that depend on background location may not trigger reliably on iOS. This limitation is documented and accepted. The feature works fully on Android. iOS users can discover routes actively via the chatbot or map view.

3.12 Admin Module

Responsibility

Provides the API layer for the React admin dashboard. Does not own any entities — delegates all operations to the relevant feature modules.

Controller Separation by Role

- SuperAdminController (/api/v1/admin/superadmin/**): User management, role assignment, account suspension. Cannot create or edit content.
- ContentEditorController (/api/v1/admin/editor/**): Create, update, and soft-delete places, routes, and quests. Cannot manage users.
- ModeratorController (/api/v1/admin/moderator/**): Review moderation queue, quest verification queue. Cannot create content or manage users.

Soft Delete Policy

All content deletions are soft deletes (is_active = false). Hard deletes are prohibited. This preserves referential integrity — a deleted place that is referenced by user quests, favorites, or route history remains resolvable. Content editors see inactive records in the admin panel; tourist-facing APIs filter them out automatically.

Monitoring Integration

Spring Actuator exposes metrics at /actuator/prometheus. Micrometer formats these for Prometheus scraping. Grafana connects to Prometheus for dashboard visualization. Loki aggregates application logs from Logback. Setup is via Docker Compose and requires approximately one day of configuration work.

4. Package Structure

Package organization follows the module-by-feature pattern. Each module is fully self-contained. Cross-module dependencies flow only through @Service interfaces, never through direct @Repository access.

4.1 Full Package Tree

```
com.tourguide
├── auth
│   ├── AuthController          @RestController  /api/v1/auth/**
│   ├── AuthService             @Service
│   ├── AuthRepository          @Repository
│   ├── JwtFilter               @Component
│   ├── JwtUtil                 @Component
│   └── dto/  LoginRequest, LoginResponse, RegisterRequest
│
├── user
│   ├── UserController          @RestController  /api/v1/users/**
│   ├── UserService             @Service
│   ├── UserRepository          @Repository
│   ├── FavoriteRepository      @Repository
│   ├── User                    @Entity
│   ├── Favorite                @Entity
│   └── dto/  UserResponse, UpdateUserRequest, FavoriteResponse
│
├── place
│   ├── PlaceController         @RestController  /api/v1/places/**
│   ├── PlaceService            @Service
│   ├── PlaceRepository         @Repository
│   ├── GooglePlacesClient      @Component
│   ├── Place                   @Entity
│   └── dto/  PlaceResponse, NearbySearchRequest
│
├── quest
│   ├── QuestController         @RestController  /api/v1/quests/**
│   ├── QuestService            @Service
│   ├── QuestRepository         @Repository
│   ├── UserQuestRepository     @Repository
│   ├── Quest                   @Entity
│   ├── UserQuest               @Entity
│   └── dto/  QuestResponse, VerifyQuestRequest
│
├── todo
│   ├── TodoController          @RestController  /api/v1/todos/**
│   ├── TodoService             @Service
│   ├── TodoRepository          @Repository
│   ├── TodoItem                @Entity
│   └── dto/  TodoRequest, TodoResponse
│
├── route
```

```
│   ├── RouteController          @RestController  /api/v1/routes/**
│   ├── RouteService             @Service
│   ├── RouteRepository          @Repository
│   ├── RoutePlaceRepository     @Repository
│   ├── Route                    @Entity
│   ├── RoutePlace               @Entity
│   └── dto/  RouteResponse, RouteRequest
│
├── chat
│   ├── ChatController           @RestController  /api/v1/chat/**
│   ├── ChatService              @Service
│   ├── ChatSessionRepository    @Repository
│   ├── ChatMessageRepository    @Repository
│   ├── ChatSession              @Entity
│   ├── ChatMessage              @Entity
│   └── dto/  ChatRequest, ChatResponse
│
├── image
│   ├── ImageController          @RestController  /api/v1/images/**
│   ├── ImageService             @Service
│   ├── ImageRepository          @Repository
│   ├── PlaceImage               @Entity
│   └── dto/  ImageUploadRequest, ImageAnalysisResponse
│
├── review
│   ├── ReviewController         @RestController  /api/v1/reviews/**
│   ├── ReviewService            @Service
│   ├── ReviewRepository         @Repository
│   ├── Review                   @Entity
│   └── dto/  ReviewRequest, ReviewResponse
│
├── badge
│   ├── BadgeController          @RestController  /api/v1/badges/**
│   ├── BadgeService             @Service
│   ├── BadgeRepository          @Repository
│   ├── UserBadgeRepository      @Repository
│   ├── Badge                    @Entity
│   ├── UserBadge                @Entity
│   └── dto/  BadgeResponse, UserBadgeResponse
│
├── notification
│   ├── NotificationController   @RestController  /api/v1/notifications/**
│   ├── NotificationService      @Service
│   ├── NotificationRepository   @Repository
│   ├── Notification             @Entity
│   └── dto/  NotificationResponse
│
├── admin
│   ├── superadmin
│   │   ├── SuperAdminController  @RestController
/api/v1/admin/superadmin/**
│   │   └── dto/  AssignRoleRequest, UserManagementResponse
│   ├── contenteditor
```

```
│   │   ├── ContentEditorController  @RestController
/api/v1/admin/editor/**
│   │   └── dto/  CreatePlaceRequest, CreateQuestRequest, CreateRouteRequest
│   └── moderator
│       ├── ModeratorController  @RestController  /api/v1/admin/moderator/**
│       └── dto/  ReviewModerationRequest, QuestVerificationRequest
│
└── common
    ├── exception
    │   ├── GlobalExceptionHandler   @ControllerAdvice
    │   ├── ResourceNotFoundException
    │   └── UnauthorizedException
    ├── entity
    │   └── BaseEntity               (created_at, updated_at, is_active)
    ├── config
    │   ├── SecurityConfig           @Configuration
    │   ├── RedisConfig              @Configuration
    │   └── WebConfig                @Configuration
    └── util
        ├── GeohashUtil
        └── CoordinateUtil
```

5. Dependency List

All dependencies are managed via Maven. The following table lists every dependency, its purpose, and the justification for its inclusion. Dependencies not in this list must be approved before addition — preventing dependency bloat is a design goal.

5.1 Spring Boot Core

Dependency	Purpose	Justification
spring-boot-starter-web	REST API layer, embedded Tomcat	Required for all HTTP endpoints
spring-boot-starter-data-jpa	Hibernate ORM, repository pattern	Database access abstraction
spring-boot-starter-security	Authentication, authorization filter chain	JWT validation, RBAC enforcement
spring-boot-starter-validation	@Valid, Bean Validation annotations	Request DTO validation
spring-boot-starter-actuator	Health endpoints, metric exposure	Required for Prometheus integration
spring-boot-starter-data-redis	Redis template, repository support	Cache layer access
spring-boot-starter-webflux	WebClient for async HTTP	Non-blocking calls to Python microservice

5.2 Database

Dependency	Purpose	Justification
postgresql	JDBC driver	PostgreSQL connectivity
hibernate-spatial	PostGIS type mapping, spatial query support	Required for ST_DWithin, geography types
liquibase-core	Database schema migration management	Prevents schema drift in team development; every schema change is versioned

5.3 Security

Dependency	Purpose	Justification
jjwt-api	JWT creation and parsing API	Industry standard JWT library for Java
jjwt-impl	JWT implementation	Required runtime companion to jjwt-api
jjwt-jackson	JSON serialization for JWT claims	Jackson integration for claim mapping

5.4 Infrastructure & Storage

Dependency	Purpose	Justification
redisson	Advanced Redis client, distributed locking	Prevents race conditions on cache writes; standard Lettuce client lacks distributed lock support
minio	MinIO / S3-compatible object storage client	Image uploads; MinIO API is identical to AWS S3, enabling zero-cost migration to S3 in production
firebase-admin	Firebase Cloud Messaging SDK	Push notification delivery to iOS and Android
google-cloud-vision	Google Cloud Vision API client	Fallback image recognition when custom model confidence is insufficient

5.5 Monitoring

Dependency	Purpose	Justification
micrometer-registry-prometheus	Exports Spring metrics in Prometheus format	Required bridge between Spring Actuator and Prometheus; zero custom code needed

5.6 Utilities

Dependency	Purpose	Justification
mapstruct	Compile-time entity-to-DTO mapping	Eliminates manual mapping boilerplate; compile-time safety catches mapping errors
lombok	Reduces boilerplate (@Getter, @Builder, etc.)	Widely adopted; reduces lines of code without runtime overhead

Dependency	Purpose	Justification
geohash-java	Geohash encode/decode utilities	Required for cache key generation in Place module; implementing geohash from scratch is unnecessary

6. Python AI Microservice

The Python AI Microservice is a separate process running FastAPI. It is the only component that Spring Boot communicates with over HTTP. All AI inference, LLM integration, and RAG pipeline logic lives here.

6.1 Endpoints

Endpoint	Method	Caller	Description
/chatbot/process	POST	ChatService	Process user message, run RAG pipeline, return LLM response
/vision/identify	POST	ImageService	Identify landmark in photo, return label + confidence
/vision/verify	POST	QuestService via ImageService	Check if target landmark is present in photo, return match + confidence

6.2 Resilience Strategy

Spring Boot uses WebClient (non-blocking) for all calls to the Python microservice. The following resilience measures are applied:

- Timeout: 30 seconds for LLM inference calls, 10 seconds for vision calls. Requests exceeding these limits return an error response — they do not block indefinitely.
- Fallback on vision failure: If the Python microservice is unreachable or returns an error, ImageService falls back to Google Cloud Vision API automatically.
- No circuit breaker in MVP: Resilience4j circuit breaker is not included in the initial implementation. If the Python service is consistently failing, the fallback chain handles it. Circuit breaker can be added post-MVP.

6.3 LLM Model Strategy

- Current: Llama 3 8B via Ollama. Runs locally on development hardware (24GB RAM, RTX 3050).
- Future: Fine-tuned version of Llama 3 8B trained on İzmir tourism data and Turkish-language content. This constitutes the 'custom model' requirement from the academic supervisor. Only the Python microservice is affected by this migration.

7. Database Schema Notes

The full database schema is defined in the existing ER diagram. This section documents implementation decisions that are not visible in the diagram.

7.1 Base Entity

All entities extend BaseEntity, which provides:

- id: UUID primary key, generated automatically
- created_at: timestamp, set on insert
- updated_at: timestamp, updated automatically by Hibernate @PreUpdate
- is_active: boolean, default true. Set to false on soft delete.

7.2 Schema Additions — Badge System

The following tables are added to the ER diagram for badge support:

- badges: id (UUID PK), name (string), description (text), icon_url (string), is_active (boolean), created_at, updated_at — extends BaseEntity
- user_badges: id (UUID PK), user_id (UUID FK → users), badge_id (UUID FK → badges), earned_at (timestamp) — UNIQUE constraint on (user_id, badge_id) prevents duplicate awards
- quests table modification: badge_id (UUID FK → badges, nullable) — a quest may optionally award a badge on completion

7.3 Indexes

Table	Index Type	Column(s)	Reason
places	GIST (spatial)	location (geography)	ST_DWithin proximity queries
places	B-tree	external_id	Upsert deduplication on Google Places sync
user_location_history	B-tree	user_id, recorded_at	Log query performance
chat_messages	B-tree	session_id	Fast message retrieval per session
quests	B-tree	status	Moderator queue filtering

7.4 Redis Key Namespace

Namespace	Key Pattern	TTL	Description
app:	user:location:{id}	300s	Last known user coordinates
app:	query:geohash:{hash}	3600s	Nearby places cache per geohash cell
app:	place:details:{id}	86400s	Full place details from Google
app:	rate:limit:{id}	60s	API rate limit counter per user
ai:	chat:session:{id}	Session duration	Active conversation history (sliding window)
ai:	inference:{hash}	3600s	Cached vision inference results
ai:	llm:response:{hash}	1800s	Cached LLM responses for identical queries

8. Known Limitations & Future Work

8.1 Accepted Limitations

Limitation	Impact	Mitigation
Quest photo spoofing possible	User can submit downloaded photos	GPS pre-check + moderator fallback reduce risk; full anti-spoofing is out of scope
iOS background location restricted	Route radius suggestions unreliable on iOS	Android fully supported; iOS users use chatbot or map for route discovery
Google Places ToS 30-day cache limit	Rich content must be re-fetched periodically	TTL-based cache strategy enforces compliance automatically
Custom vision model limited to pilot zones	Feature unavailable outside trained regions	Clear error message returned; region expansion tied to training data acquisition
No circuit breaker in MVP	Python service failures propagate without fast-fail	Fallback chain handles most scenarios; Resilience4j to be added post-MVP

8.2 Deferred Features

- Foursquare API integration — evaluated and rejected for MVP. Can be added if Google Places proves insufficient for category richness.
- iOS background location push notifications — architecturally planned. FCM integration is already included; trigger mechanism deferred.
- Resilience4j circuit breaker — add after MVP stabilization.
- Vector database (pgvector or Qdrant) — improves RAG retrieval quality. Current keyword-based retrieval is sufficient for MVP but semantic search will be added in future sprint.
- Expansion to Manisa, Aydın, Muğla — requires training data acquisition and pilot zone configuration. Architecturally, no changes are needed — only data and zone config.