Project - High Level Design

on

Haven AI

Course Name: Agentic AI

Institution Name: Medicaps University – Datagami Skill Based Course

Sr no	Student Name	Enrolment Number
1.	Rudra Sthapak	EN22CS301828
2.	Rajvardhan Singh Rathore	EN22CS301786
3.	Ronak Bhawsar	EN22CS301824
4.	Rupendra Singh Tomar	EN22CS301833
5.	Raj Kumawat	EN22CS301779
6.	Priyanka Kumari	EN22CS301758

Group Name: 04D7 Project

Number: AAI-27

Industry Mentor Name:

University Mentor Name: Prof. Ajeet Singh Rajput

Academic Year: 2025-2026

Table of Contents

1. Introduction

This document describes the High-Level Design (HLD) of the Real Estate Support Triage Agent. The system is designed to automate customer query handling in the real estate domain using conversational AI integrated with backend services and a database layer.

The HLD focuses on architectural structure, system modules, integration points, data flow, and key technical decisions.

1.1. Scope of the Document This

document defines:

- Overall system architecture
- Major modules and responsibilities
- Data flow and process flow
- API definitions
- Data design and storage strategy
- Non-functional requirements
- Security and performance considerations

This document does not include low-level implementation details such as code logic or internal function definitions.

1.2. Intended Audience

This document is intended for:

- Industry Mentor
- University Mentor
- Technical Review Panel (Datagami Members)
- Development Team Members
- System Architects

1.3. System Overview

The Real Estate Support Triage Agent is a web-based AI-powered system that:

- Accepts user queries in natural language
- Extracts structured property parameters
- Performs filtered database search
- Returns contextual conversational responses

The system consists of:

- Frontend Application (Next.js)
- Backend API Layer (FastAPI)
- AI Orchestration Layer (Gemini via LangChain)
- Database Layer (MongoDB)

2. System Design

The system follows a modular layered architecture:

Presentation Layer
Handles UI interactions and API communication.

Application Layer
Handles authentication, chat processing, and business logic.

AI Layer
Responsible for intent detection and entity extraction.

Data Layer
Stores properties, users, and OTP data.

Architecture Pattern Used:

- Client-Server Architecture
- RESTful API Communication
- AI Tool-Calling Integration Pattern

2.1. Application Design

The backend application is structured into modules:

1. Authentication Module ○ Signup ○ Login ○ OTP Reset
2. Chat Processing Module ○ User Query Handling ○ LLM Integration ○ Tool Invocation
3. Property Search Module ○ MongoDB Query Builder ○ Filter Application

2.2. Process Flow

Chat Flow:

User → Frontend → Backend API → LLM → Tool Call → MongoDB → LLM → Backend → Frontend → User

Authentication Flow:

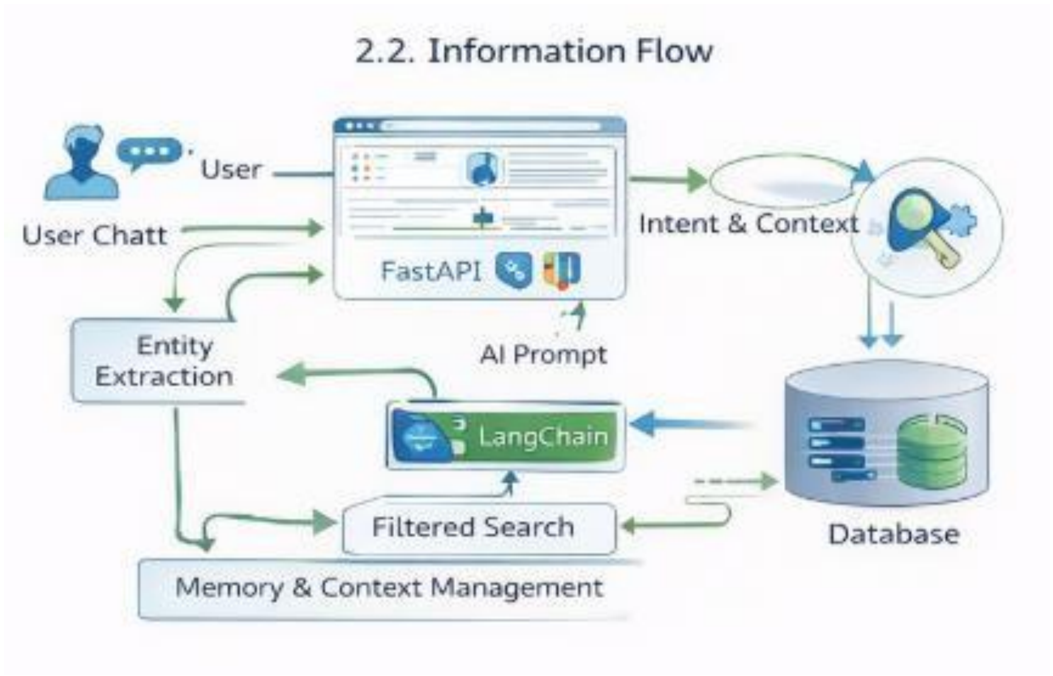
User → Signup/Login → Password Hashing → Database Validation
Password Reset → OTP Generation → TTL Validation → Reset Password

2.3. Information Flow

Input:
Natural language user query

Processing:
Entity extraction → Parameter validation → Database filtering

Output:
Structured property results converted into conversational response



2.4. Components Design

Frontend Component:

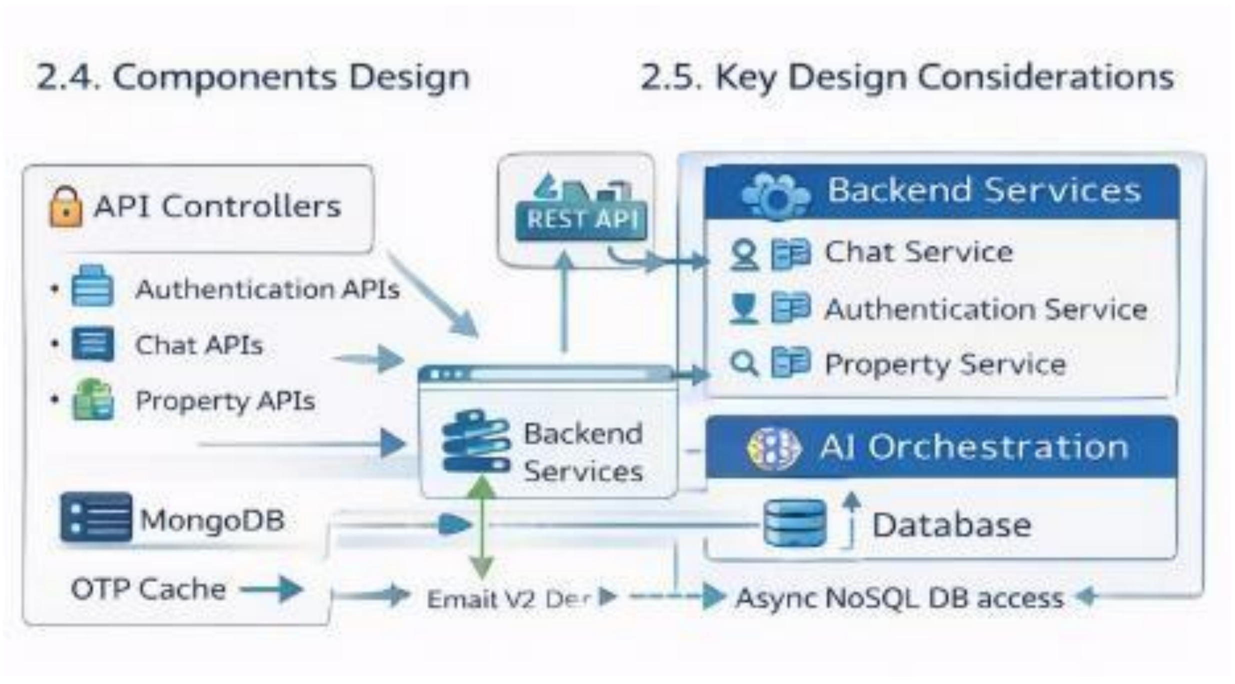
- Chat Interface
- Authentication Forms

Backend Components:

- API Controllers
- Service Layer
- LLM Integration Service
- Database Access Layer

Database Components:

- Users Collection
- Properties Collection
- OTP Collection



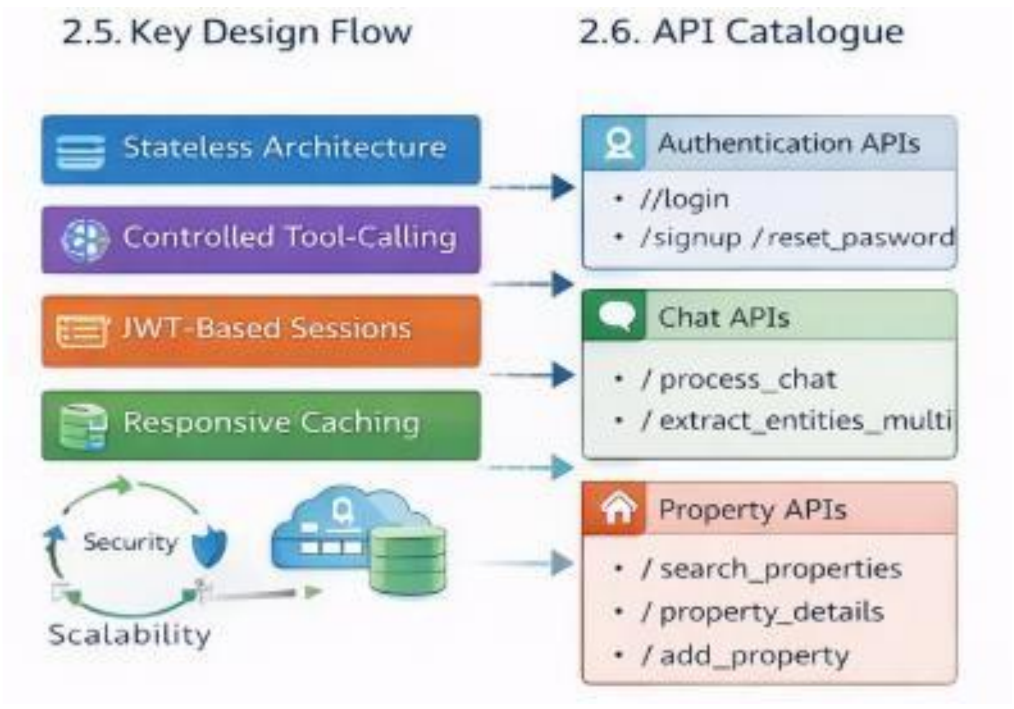
2.5. Key Design Considerations

- Modular architecture for scalability
- Asynchronous processing using FastAPI
- Separation of AI logic and business logic
- Secure password storage • Flexible NoSQL schema
- Tool-calling architecture for structured queries

2.6. API Catalogue

POST /signup
POST /login
POST /forgot-password
POST /verify-otp
POST /reset-password POST
/chat

All APIs follow REST standards and return JSON responses.



3. Data Design

Data stored in MongoDB:

Users:

- user_id
- email
- hashed_password
- created_at Properties:

- property_id
- location
- bhk
- price
- property_type
- description

OTP:

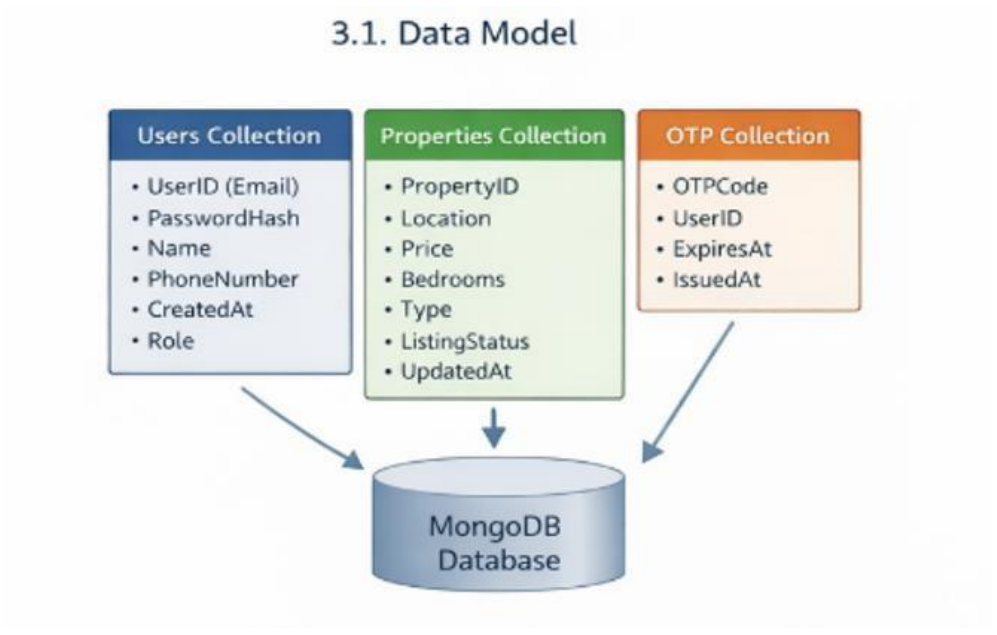
- email
- otp
- expiry_time

3.1. Data Model

The system uses a NoSQL schema design allowing flexible property attributes.

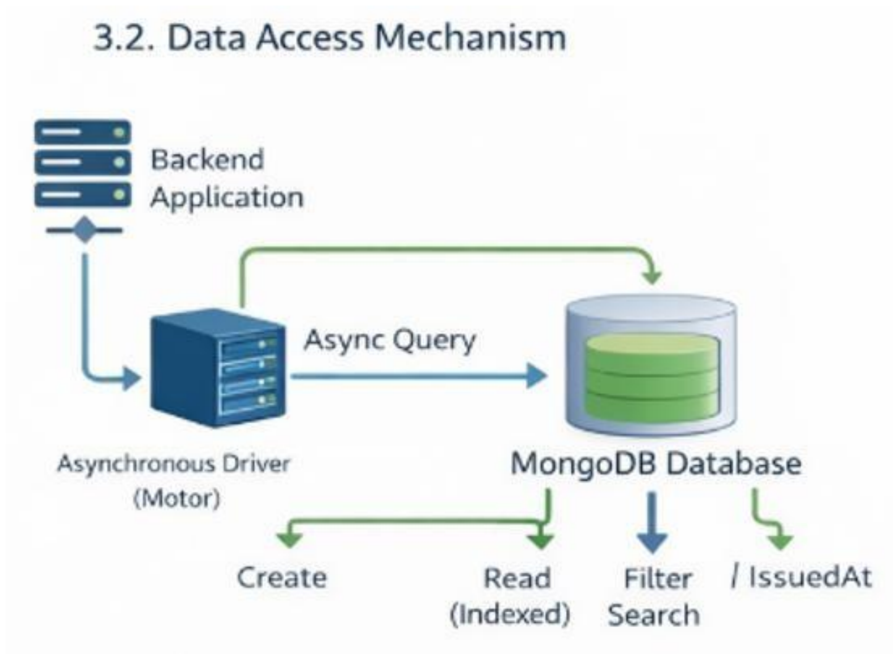
Primary Relationships:

- Users linked via email
- OTP linked to user email
- Properties independent entity



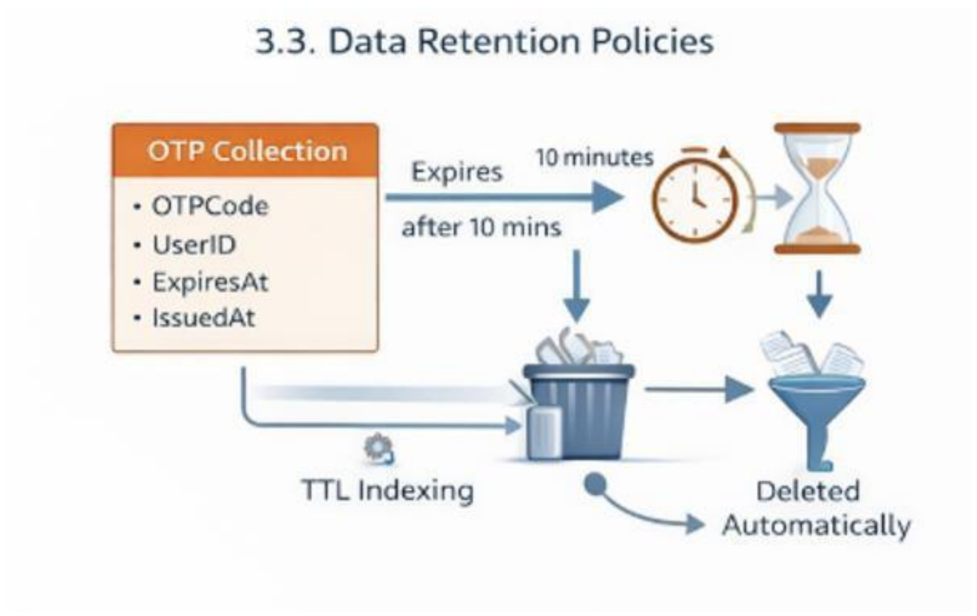
3.2. Data Access Mechanism

- Async MongoDB driver (Motor)
- Non-blocking queries
- Filter-based search
- Indexed search on location and price



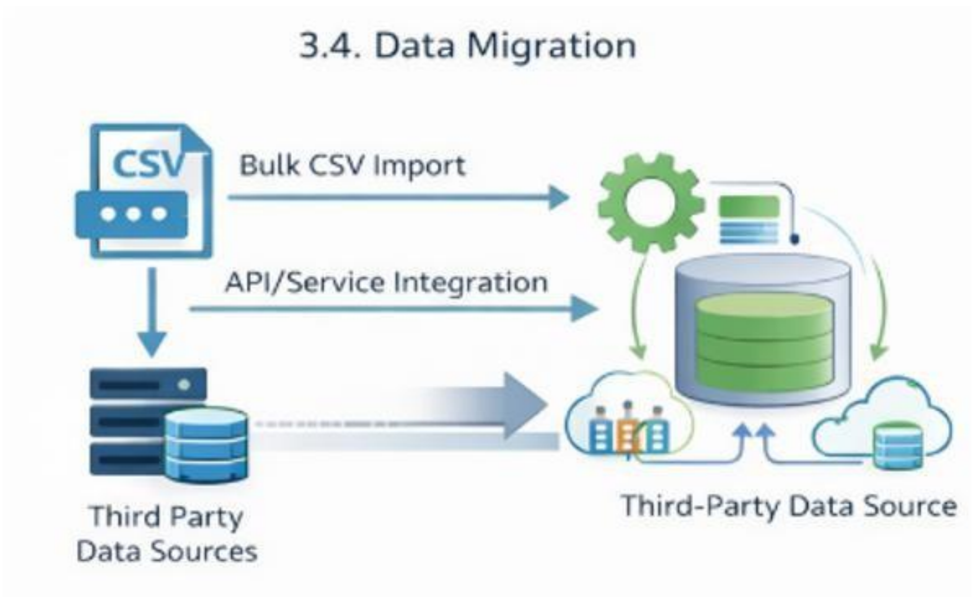
3.3. Data Retention Policies

- OTP records auto-expire via TTL index
- User accounts retained unless deleted
- Property listings updated dynamically



3.4. Data Migration

Currently not required.
Future integration may include bulk CSV property import.



4. Interfaces

The Real Estate Support Triage Agent communicates through multiple well-defined interfaces that ensure modularity, scalability, and clear separation of responsibilities across system components. The primary interface exists between the frontend application and the backend API layer, where communication is handled using RESTful principles over HTTP/HTTPS. All requests and responses are exchanged in structured JSON format, ensuring standardized data transfer and easy extensibility. This interface remains stateless and follows proper HTTP status conventions to maintain consistency and reliability.

Beyond client-server communication, the backend interacts with the Large Language Model through an abstraction layer implemented using LangChain. This interface is responsible for prompt construction, entity extraction, and structured tool-calling. Instead of allowing the LLM to directly access the database, the system restricts it to predefined backend tools such as property search. This controlled interaction improves reliability, prevents unsafe operations, and ensures predictable system behavior.

The backend also communicates with the MongoDB database through an asynchronous data access layer using the Motor driver. This interface supports non-blocking operations, dynamic query construction, and indexed search mechanisms to ensure optimal performance. Internally, backend modules interact through clearly defined service-layer contracts, maintaining loose coupling and improving maintainability. Overall, the interface design ensures that each component of the system can evolve independently without disrupting other layers, which is essential for scalable and production-ready architecture.

5. State and Session Management

The Real Estate Support Triage Agent follows a stateless architectural design, meaning that each request is processed independently without maintaining persistent server-side session storage. This approach simplifies horizontal scaling and reduces memory consumption on the server. Since no session state is stored between requests, the backend can efficiently handle concurrent users without synchronization issues.

Authentication is validated during each relevant request to ensure secure access to protected endpoints. Currently, chat interactions are processed independently, and conversation history is not persistently stored. This aligns with the project's defined scope and ensures lightweight backend processing. The stateless nature of the system also makes deployment and load balancing significantly easier.

However, the architecture is designed to support future enhancements such as JSON Web Token (JWT) based authentication. Token-based authentication would allow secure session validation without storing server-side state. Additionally, persistent chat context storage can be introduced to enable more personalized and context-aware conversations. Expiration policies, token refresh mechanisms, and inactivity timeout handling can be incorporated in future versions to strengthen both security and user experience while maintaining scalability.

6. Caching

Although caching is not implemented in the current version of the Real Estate Support Triage Agent, it is an important architectural consideration for performance optimization. Conversational AI systems frequently receive repeated or similar property queries. Without caching, each query requires full processing, including entity extraction, database filtering, and response generation, which increases latency and system workload.

Introducing a caching layer such as Redis between the backend and database would significantly reduce redundant database calls. Frequently requested property results could be temporarily stored and quickly retrieved for identical or similar queries. This would reduce response time and improve overall user experience. Additionally, caching LLM-generated responses for repeated structured queries could lower computational overhead and reduce external API costs.

Proper cache expiration policies would be required to ensure that cached property data remains consistent with the database. Time-based invalidation or version-based cache keys could be implemented to maintain data freshness. By integrating caching into the architecture, the system would achieve improved scalability, reduced server load, and faster response times. The current modular design ensures that a caching layer can be introduced in the future without major structural modifications.

7. Non-Functional Requirements

Non-functional requirements define the quality attributes of the Real Estate Support Triage Agent and ensure that the system performs efficiently, securely, and reliably under different conditions.

Scalability

The system is designed using a modular and stateless architecture, allowing it to handle increasing user traffic without performance degradation. Backend services can be scaled horizontally, and MongoDB supports data scaling as property listings grow.

Availability

The application is designed to remain accessible with minimal downtime. Stateless APIs and independent frontend-backend deployment allow the system to continue functioning even if one component temporarily fails.

Maintainability

The system follows a modular design with clear separation of authentication, chat processing, AI integration, and database operations. This structure makes the application easier to update, debug, and extend in the future.

Reliability

Input validation, structured tool-calling, and controlled database access ensure consistent system behavior. The architecture minimizes unexpected failures and maintains stable performance under normal operating conditions.

Security

Passwords are securely stored using bcrypt hashing, and OTP verification includes expiration control. Proper validation and separation between AI and database layers protect sensitive user information.

Usability

The chat-based interface allows users to search properties using natural language, making the system simple and user-friendly. Authentication processes are straightforward to ensure smooth user interaction.

7.1. Security Aspects

- bcrypt password hashing
- OTP-based password recovery
- Input validation
- Modular API protection
- Database access restriction

7.2. Performance Aspects

- Asynchronous request handling
- Efficient MongoDB filtering
- Reduced response latency
- Modular micro-service ready architecture

8. References

1. FastAPI Documentation. *FastAPI – High Performance Web Framework for Building APIs with Python.* Available at: https://fastapi.tiangolo.com/

2. MongoDB Documentation. *MongoDB Official Documentation.* Available at: https://www.mongodb.com/docs/

3. Motor Documentation. *Motor – Asynchronous Python Driver for MongoDB.* Available at: https://motor.readthedocs.io/

4. LangChain Documentation. *LangChain Framework Documentation.* Available at: https://python.langchain.com/

5. Google AI Documentation. *Gemini API Documentation.* Available at: https://ai.google.dev/

6. OWASP Foundation. *OWASP Top 10 – Web Application Security Risks.* Available at: https://owasp.org/www-project-top-ten/

7. Redis Documentation (for future caching reference). Available at: https://redis.io/documentation/

8. Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures (REST Dissertation).* University of California, Irvine.