# 1. Executive Summary & Scope

The objective is to build a **manus-like AI-powered property rental platform** that behaves like a personal real-estate agent.  Users (tenants/guests) converse with a client-side agent to find, evaluate, negotiate and book properties, while hosts and partner companies manage inventory and bookings through dedicated dashboards.  We mirror SUNA’s architecture (FastAPI + React + Supabase + multi-agent orchestration + VNC sandbox) but create a clean-room implementation tailored to property rentals.  The platform supports both short- and long-term rentals, integrates third-party listing feeds and payment providers, and exposes real-time voice interactions using the ChatGPT Realtime API.  A modular multi-agent layer orchestrates tasks such as property search, scoring, negotiation, contracts and payments.

Key principles:

* **Separation of concerns:** API gateway, agent orchestrator, vector/RAG services, payments, voice and partner integrations run as independent services with clear contracts.
* **Multi-agent orchestration:** specialized agents (search, scoring, negotiation, contract) coordinated by an orchestrator; supervisor patterns ensure complex tasks are escalated to more capable models【612219450950787†L265-L374】.
* **Supabase for auth/RLS and PostgreSQL storage:** row-level security (RLS) must be enabled on all public tables【178230355922264†L270-L296】; policies enforce per-role access.
* **Event-driven and idempotent flows:** payments and bookings use idempotency keys to prevent double-charges; events trigger downstream actions.
* **Full observability and safety:** every agent/tool call is logged; guardrails restrict non-real-estate actions and personal financial advice; human-in-the-loop checkpoints for payments/contracts.

# 2. System Roles & RBAC Matrix

| Role | Description | Allowed actions (high level) |
| --- | --- | --- |
| **Client (Tenant/Guest)** | End users searching for and booking properties. | Create/update own profile; perform natural language searches; view/search listings; save searches/favorites; chat/voice with agent; request negotiations; complete bookings and payments; access own contracts and receipts; leave reviews. |
| **Host / Real-Estate Company** | Company owners or property managers. | Onboard sub-agents; list/edit own properties; set availability/pricing; respond to leads; view analytics on their inventory; manage bookings; issue refunds/cancellations (within policy); integrate VAPI to run their own voice/chat bot; download contracts and payment reports. |
| **Sub-Agent** | Individual agents working under a host company. | View company properties; manage assigned properties; respond to leads; handle negotiations; accept/reject booking requests; cannot modify company settings. |
| **Data-Partner Company** | Third-party inventory providers (e.g., MLS, OTAs). | Expose property feeds to our ingestion API; view status of their listings; cannot modify bookings; optionally receive aggregated analytics. |
| **Superadmin** | Internal platform administrators. | Full access: manage all users, properties, bookings; view logs; configure policies; run migrations; handle disputes; view revenue dashboards; override transactions; manage partner integrations. |

## RBAC Matrix Summary

The table below summarises which roles can perform CRUD on key objects.  Detailed row-level policies are defined in the database section.

| Object | Client | Host | Sub-Agent | Data-Partner | Superadmin |
| --- | --- | --- | --- | --- | --- |
| Users/Profiles | Read/update own profile | Manage own company users | Manage own assignments | No access | Full control |
| Companies | View hosting company info | CRUD own company | View own company | No access | Full control |
| Properties | Read/search public listings; favorite | CRUD own properties | Update assigned properties | Provide feed only | Full control |
| Availability & Pricing | View for selected properties | Set/update for own properties | Manage assigned | Feed only | Full control |
| Searches & Sessions | Create/search; read own history | View aggregated analytics | View assigned leads | No access | Full control |
| Bookings & Contracts | Create for own bookings; cancel per policy | Manage bookings for own properties | Manage assigned bookings | No access | Full control |
| Payments | Initiate own payments; view receipts | View payments for own properties | View assigned | No access | Full control |
| Events & Logs | View own agent/tool history | View company logs | View assigned logs | View feed ingestion logs | Full access |

# 3. End-to-End Architecture Diagram & Narrative

## 3.1 Monorepo Structure

```
repo/
  apps/
    api-gateway/         # FastAPI application exposing HTTP/WS endpoints
    agent-orchestrator/  # LangGraph/LangChain based orchestrator managing agents & tools
    vector-service/      # Service wrapping pgvector / Pinecone for embeddings & RAG
    payments-service/    # Event-driven payment & escrow microservice
    voice-service/       # WebSocket proxy to ChatGPT Realtime API & VAPI
    host-dashboard/      # Next.js (React) app for host & sub-agent UI
    partner-dashboard/   # Next.js app for data-partner view
    client-web/          # Next.js app for tenant UI (with chat & sandbox)
    superadmin-console/  # Next.js admin app
  libs/
    ui-components/       # Shared React components (property cards, negotiation timeline)
    agent-tools/         # Reusable tool definitions & helpers
    db-schema/           # SQL migrations & Supabase DDL
    contracts/           # Code to generate PDF/e-sign contracts
    types/               # Shared TypeScript & Pydantic models
  infra/
    docker-compose.yml   # Dev containers: FastAPI, Supabase, vector DB, etc.
    k8s/                 # Helm charts for production deployment
  tests/
    unit/
    integration/
    e2e/
  docs/
```

## 3.2 Service Boundaries

* **API Gateway:** Single FastAPI application responsible for HTTP and WebSocket transport.

Performs authentication (Supabase JWT), rate-limiting, validation and request routing.  Routes include `/search`, `/bookings`, `/payments`, `/voice`, `/agent/events`, `/webhooks/...` and GraphQL or gRPC if needed.
* **Agent Orchestrator:** Coordinates domain-specific agents.  Implements planning/execution/memory; orchestrates tools; persists agent runs & tool calls to `agent_runs` and `tool_calls` tables. Implements multi-agent patterns described by OpenAI (chat agent and supervisor)【612219450950787†L265-L374】.  Agents run in isolated sandboxes (via the VNC panel) to avoid side effects and follow guardrails.
* **Vector/RAG Service:** Handles embeddings (pgvector) and semantic search across properties, previous conversations and documentation.  Exposes API for similarity search and retrieval.  Stores property documents and knowledge base (legal regulations, neighbourhood data) in a vector index.
* **Payments Service:** Encapsulates payment provider integrations (Stripe, PayPal, regional gateways).  Manages payment intents, escrow accounts, holds, releases and refunds.  Emits events like `payment_intent.created`, `payment.captured`, `payment.refunded`.
* **Voice Service:** Wraps ChatGPT Realtime API or Vapi.  Provides streaming STT/LLM/TTS with low latency (sub-600 ms conversations【363436922402494†L170-L178】 and minimal delay【602446708061703†L567-L577】).  Maintains sessions, handles barge-in and handshake with the agent orchestrator for tool calls.
* **Host Dashboard:** React/Next.js app for hosts and sub-agents.  Integrates with Vapi for their own customer-support bot; receives webhooks about calls; displays lead inbox, property analytics and SLA metrics.
* **Partner Dashboard:** Simple read-only React app for data-partners to view ingestion status, booking metrics and update feed credentials.
* **Superadmin Console:** Admin UI to manage users, roles, view event logs, moderate content, inspect agent runs and tune prompts.
* **Storage (Supabase/Postgres + S3):** Houses relational data (users, companies, properties, bookings, payments, contracts, events) and file storage (images, documents).  RLS policies enforce per-role access【178230355922264†L270-L296】.
* **Search & Ranking Pipeline:** ETL and scoring jobs ingest partner feeds, deduplicate, enrich and score listings.  Runs in scheduled jobs or event triggers; persists results in `property_scores` table.

## 3.3 Data Flow: Search → Scoring → Negotiation → Booking → Payment → Contract Generation

1. **Search Initiation:** A client enters a natural language query (“2-bedroom in Dubai”).  The API Gateway sends the query to the agent orchestrator.
2. **Planning & Tools:** The orchestrator uses the search agent to parse the query, extract hard filters (location, bedrooms, price range) and call the **search tool**.  The tool queries the search & ranking service, which uses vector search + filter criteria to return normalized property objects with scores.
3. **Scoring & Ranking:** For each candidate property, the scoring pipeline calculates a weighted score based on price, location match, commute time, amenities, lifestyle match, reviews and seasonality (see section 13).  Weighted ranking uses user-expressed priorities and normative indexes (e.g., quality of life and value indices【728461639475720†L94-L156】).  Results are returned to the agent.
4. **Presentation:** The agent sends a JSON object describing property cards (compact or expanded). The UI renders property cards in the chat stream and allows the user to favourite, request details or compare.
5. **Negotiation:** If the user selects a property, the negotiation agent engages with the host/sub-agent via asynchronous messages (chat or email) to negotiate rent, start date and terms. It uses a negotiation tool to send proposals and record counter-offers, storing them in `negotiations` table and updating the chat UI with a negotiation timeline.
6. **Booking:** Once agreement is reached, the agent triggers the booking tool.  The API Gateway validates availability, reserves the slot in `bookings` table and generates a booking token.  It instructs the client UI to open the payment modal.

7. **Payment & Escrow:** The client enters payment details via a PCI-compliant iframe (e.g., Stripe’s Payment Element). The payments service creates a payment intent with a hold. An idempotency key ensures repeat requests produce the same payment【602446708061703†L593-L604】. Once the hold succeeds, an escrow record is created.
8. **Contract Generation:** The contract agent generates a rental agreement in PDF, merges dynamic fields (user name, property, price, dates) and calls an e-signature API. The client and host sign; upon success, the payments service captures the payment and releases funds per policy (minus escrow fees). The booking state transitions to `confirmed`. A confirmation email and push notification are sent to both parties.

# 4. Agentic Layer Design (Client-side, Real Estate Domain)

## 4.1 Tooling Catalogue & Capability Limits

| Tool name | Description | Inputs | Outputs | Limitations |
| --- | --- | --- | --- | --- |
| **search_properties** | Query our normalized search index (filters + vector search) and return candidate properties sorted by score. | `filters`: location, bedrooms, price range, date range; `preferences`: keywords; `limit`, `offset`. | List of property objects with `id`, `score`, `summary`, `image_urls`. | Max 50 results per call; cannot perform external network calls (only our DB & cache). |
| **score_property** | Compute detailed scoring across multiple dimensions (price, location, commute, amenities, lifestyle, reviews, seasonality). | `property_id`, `user_preferences` (weights). | Score breakdown with weighted total and explanation. | Read-only; cannot alter property data. |
| **get_neighbourhood_info** | Retrieve neighbourhood insights via our knowledge graph (schools, safety, amenities)【253406740739376†L195-L232】. | `location_id`. | Structured info (safety index, schools, amenities, transport). | Provided data may be stale if not refreshed; must include timestamp. |
| **send_inquiry** | Send a lead to host/sub-agent with proposed terms. | `property_id`, `message`, `desired_dates`, `offer_price`. | `inquiry_id`, status (`sent`). | Does not finalize booking. |
| **negotiate_offer** | Post a counter-offer or accept/reject an offer in an ongoing negotiation. | `negotiation_id`, `action` (accept/counter/reject), `counter_price` (optional). | Updated negotiation record with state; triggers notifications. | Not allowed outside active negotiations or after expiry. |
| **create_booking** | Reserve property dates and generate booking token. | `property_id`, `user_id`, `dates`, `negotiation_id` (optional). | `booking_id`, `status` (pending_payment), `total_amount`. | Fails if property unavailable or user already has overlapping bookings. |
| **initiate_payment** | Start payment intent via payment service. | `booking_id`, `payment_method_id`, `idempotency_key`. | Payment intent status and client secret. | Cannot capture payment; only holds funds until contract signed. |
| **generate_contract** | Create contract PDF and send for e-signature. | `booking_id`, `contract_template_id`, `signers`. | `contract_id`, `status` (pending_signature), signed URL. | Must be invoked after payment hold; requires host consent. |
| **confirm_payment** | Capture funds and finalize booking. | `booking_id`, `payment_intent_id`. | Updated booking status (`confirmed`), receipt. | Only executed after contract signatures; requires human confirmation if above threshold. |
| **cancel_booking** | Cancel booking and issue refund. | `booking_id`, `reason`. | Refund status. | Follows cancellation policy; may require approval. |
| **save_search** | Persist user search query and filters. | `user_id`, `query`, `filters`. | `saved_search_id`. | Rate-limited to avoid spam. |
| **fetch_saved_searches** | Retrieve user’s saved searches. | `user_id`. | List of saved searches with metadata. | Only own searches; read-only. |
| **fetch_agent_logs** | Retrieve logs of agent runs and tool calls. | `user_id`, filters. | Structured log entries. | Read-only; only accessible by owners. |

| **open_ui_panel** | Instruct UI to open/close/pin panels (e.g., comparison view, sandbox). | `panel_name`, `action`, `payload` (optional). | UI event ack. | UI only; cannot call server. |

Tools are registered with the orchestrator and described in a YAML/JSON schema exposed to the LLM. The agent cannot execute arbitrary code; all side-effects must go through tools.  Each tool call is logged with inputs/outputs for audit.

## 4.2 Planning, Execution & Memory

* **Planning:** The high-level agent uses a planner (based on LangChain-`StructuredChatAgent` or OpenAI Agents SDK) to decide which tools to call and in what order.  It uses chain-of-thought reasoning but messages to the user remain concise.  If tasks are complex (e.g., negotiation), the agent delegates to a specialized negotiation sub-agent (supervisor pattern)【612219450950787†L265-L374】.
* **Execution:** Execution is orchestrated through asynchronous tasks.  Tool calls are queued; the orchestrator monitors timeouts and rate limits.  Each tool returns structured data which the agent interprets before continuing.
* **Memory & Context:** For each user session, conversation history, preferences and previous recommendations are stored in an `agent_sessions` table and in vector embeddings.  A conversation context manager tracks user preferences, history and current state【253406740739376†L155-L174】.  This ensures the agent does not repeatedly ask for information and allows progressive refinement.

## 4.3 Web Search & Partner Integrations

* **Partner APIs:** The `data-partner` ingestion pipeline fetches property listings from MLS, OTA or partner companies via REST/GraphQL/FTP.  For real-time availability and pricing, the agent can call partner APIs through approved tools using API keys.  These calls fetch additional details (reviews, dynamic pricing) but are rate-limited.
* **Public Marketplaces:** For publicly available listings (e.g., AirBnB, Zillow), the platform uses scraping/official APIs to ingest and store normalized data.  Web search tools are not exposed directly to the agent; instead, property data is curated and pre-approved.

## 4.4 Property Study Pipeline

1. **Ingestion & Normalization:** Raw feeds are ingested into staging tables (`raw_properties`). Deduplication identifies duplicates across partners using fuzzy matching (address, coordinates, host name).  Data is normalized into canonical fields (location, size, amenities).
2. **Enrichment:** The pipeline enriches each property with neighbourhood insights (schools, safety, amenities) via our knowledge graph【253406740739376†L195-L232】, calculates commute times to points of interest, fetches walkability scores, weather/seasonality information and legal checks (zoning, rental regulations).  For fuzzy expressions (e.g., “affordable” or “family friendly”), we convert them into numeric ranges using market data【253406740739376†L279-L317】.
3. **Scoring & Ranking:** Each property receives a multi-dimensional score (section 13) and the reasons behind the score are stored in `property_scores` for transparency.  Weighted rankings account for user priorities and normative indexes (quality of life, value, desirability)【728461639475720†L94-L156】.
4. **Legal & Compliance Checks:** Properties are checked against blacklists (fraudulent hosts), regulatory requirements (e.g., Ejari/DEWA in UAE), permit status and host verification.  Failing properties are flagged and excluded.

## 4.5 Safety, Guardrails, Rate-Limits & Audit Logs

* **Safety Policies:** The agent is forbidden from performing tasks unrelated to real estate (e.g., personal financial advice, buying stocks, medical recommendations).  It must not initiate payments or contract signing without explicit user confirmation.  It cannot disclose personal data of other

users or hosts. The agent must decline if asked to circumvent policies or perform illegal actions.
* **Prompt Templates:** We use system prompts that define allowed tools, guidelines and safe responses. Developer prompts enforce structure (e.g., always return JSON for UI instructions). Guardrail prompts require the agent to call `human_escalation` tool if it encounters unknown tasks or high-risk operations.
* **Rate-Limits:** The orchestrator enforces per-user rate limits on tool calls (e.g., max 10 searches/minute, 5 negotiations/day) and per-endpoint quotas to avoid API abuse.
* **Audit Logging:** Every tool call, agent decision and user message is persisted in `agent_runs` and `tool_calls` tables. Logs include input/output, timestamps, latency and model version. Sensitive events (payments, contract signing) are also logged to an immutable `audit_log` table with cryptographic hashes for tamper-evidence.

# 5. Voice Agent (Realtime STT/LLM/TTS) Design

## 5.1 Pipeline

1. **STT (Speech-to-Text):** The voice service receives audio streams via WebSocket or telephony (through Vapi). It uses the ChatGPT Realtime API or third-party STT (Deepgram/AssemblyAI) to transcribe audio into text. The Realtime API provides bi-directional audio streaming with minimal latency【602446708061703†L567-L577】 and supports multi-language and customizable voice styles【602446708061703†L59-L67】.
2. **Intent Handling:** The voice layer forwards transcribed text to the agent orchestrator (chat agent) which processes it as if it were a chat message. If the chat agent determines that a tool call is required, it sends an event to the orchestrator and returns a placeholder response (“Give me a moment to check…”) to the user. When the supervisor agent completes the tool call, the full response is streamed back to the user【612219450950787†L265-L374】.
3. **TTS (Text-to-Speech):** The LLM’s textual response is converted back to audio using the Realtime API’s TTS or a custom TTS provider. The voice service streams the audio back to the caller. Real-time conversations achieve sub-600 ms response times with natural turn-taking【363436922402494†L170-L178】.
4. **Session Management:** Each voice call is a session. The voice service maintains state (session ID, user ID, conversation context) and interacts with the agent orchestrator via WebSocket events. It handles barge-in (user interrupts while the agent is speaking) by pausing TTS and sending the partial transcript to the agent. Session metadata is recorded in `voice_sessions` table.

## 5.2 Latency Budget & Streaming UX

* **STT latency:** ~200 ms (depending on provider). Use streaming transcription to deliver partial results quickly.
* **Agent planning & tool call:** <800 ms for simple responses; if a tool call is needed, initial acknowledgement is immediate but the full answer may take 1–3 s. The chat-supervisor pattern ensures the voice agent responds immediately with a filler phrase while the supervisor completes the task【612219450950787†L265-L374】.
* **TTS latency:** ~200 ms for streaming TTS (11Labs or Realtime API). Use neural TTS models with caching.
* **Overall time:** <1 s for direct answers; <3–5 s for tool calls.

## 5.3 Session Handoff to Text UI

If during a call the user wishes to switch to the web chat interface (e.g., to view property cards or complete payment), the voice agent sends a session handoff event containing the session ID and context. The client web UI loads the conversation history and continues in text mode. Conversely, a text session can be escalated to a voice call by sending a `start_call` event.

## 5.4 Vapi Integration

The host dashboard integrates Vapi to allow hosts to run their own voice agents (for inbound support or booking questions). Each host configures a Vapi assistant with a system prompt and tool definitions (limited to their data). The platform provides endpoints for Vapi webhooks:

* `POST /webhooks/vapi/call_started` – triggered when a call begins; includes caller ID, host ID, assistant ID. Creates a `voice_session` record.
* `POST /webhooks/vapi/message` – streaming transcripts of caller and assistant messages. The orchestrator can optionally inject property details or actions.
* `POST /webhooks/vapi/call_ended` – call summary, duration, outcome (resolved/escalated), call recordings.

Hosts can view call analytics, transcripts and follow-up tasks in their dashboard. Sensitive user data (payment info, addresses) is never exposed to Vapi; only summary property metadata and statuses are provided.

# 6. Host / Real-Estate Company Dashboard

## 6.1 Permissions Model

Hosts belong to a `companies` table and have `role` values (`owner`, `manager`, `agent`). Owners can create sub-agents and assign them to properties. Managers can add/edit properties and manage bookings. Agents can view and respond to leads for assigned properties and update availability/pricing.

Role mapping is enforced via Supabase RLS policies; e.g., `properties.company_id = auth.claims.company_id` and `sub_agents.assigned_property_id = properties.id`.

## 6.2 CRUD & Workflows

* **Property CRUD:** Create, read, update, delete properties with details such as location, description, amenities, images, pricing, availability calendar and legal documents. Create & edit operations call the API gateway; images are uploaded to Supabase storage or S3.
* **Availability & Pricing Management:** Manage calendar availability. Bulk update pricing via CSV or UI. Use derived pricing suggestions from the scoring pipeline.
* **Lead Inbox & Negotiations:** See incoming inquiries and negotiation threads. Accept/decline offers, propose counter-offers. SLA metrics track response time.
* **Booking Management:** View pending bookings, confirm or cancel bookings. Issue refunds via the payments service according to cancellation policies.
* **Analytics:** Dashboard shows occupancy rates, revenue, average negotiation duration, cancellation rates and customer satisfaction scores.
* **Automated Replies & Templates:** Hosts can define canned responses for common questions. Agent tools can use these templates when negotiating.
* **VAPI CS Bot:** Hosts may configure a Vapi assistant. They specify allowed tools (e.g., answer FAQ, check booking status) and allowed data (only properties belonging to the company). The bot triggers our endpoints when tasks require external actions (e.g., generating a discount code). The Vapi call events are visible in the host’s analytics.

# 7. Data-Partner Company Dashboard

Data partners expose their inventory via our ingestion API. They authenticate using API keys and specify `org_id` in requests. Partner dashboards allow them to:

* View ingestion status of each feed (success/failure, last updated, number of active listings).
* Upload new feeds or modify feed configuration (FTP/S3 endpoints, API credentials).
* View aggregated metrics: number of bookings, conversion rates and revenue generated from their inventory.  Partners **cannot** view individual user data or bookings; only aggregated anonymized metrics are shown.
* Receive webhook notifications for booking events: `booking.created`, `booking.cancelled`, `payment.captured` with minimal payload (property ID, booking dates, revenue amount) for reconciliation.

# 8. Superadmin Console

The superadmin console provides full observability and control:

* **User & Role Management:** Create, update, suspend or delete any user.  Assign roles and manage company/sub-agent relationships.
* **Property & Booking Overview:** View all properties, bookings and statuses across the platform.  Force unlock or override bookings if necessary.
* **Moderation & Incident Tooling:** Flag suspicious properties or conversations; review agent responses; suspend hosts or properties.  Manage disputes and chargebacks.
* **Prompt & Model Management:** Deploy new prompt versions, view performance metrics and A/B test results.  Roll back prompts if regressions occur.
* **Revenue & Settlement Dashboard:** View platform-wide revenue, escrow balances, outstanding refunds, partner payouts and host earnings.  Manage settlement schedules.
* **Logs & Trace Explorer:** Inspect agent runs, tool calls, voice session transcripts, API requests and database events.  Use filters and full-text search.  Detect anomalies (e.g., repeated failed payments).  Export logs for audits.
* **System Health & Alerts:** Monitor service latencies, error rates, throughput.  Configure alerts for downtime, high error rates or unusual activity.  Integration with PagerDuty/Slack.

# 9. Frontend UX Specification (Manus/SUNA-like 3-column)

## 9.1 Layout

* **Left Column (Sidebar):**
  * Account details and avatar; link to profile settings and KYC.
  * Session history (previous conversations/voice calls).  Users can resume or delete sessions.
  * Saved searches and favourites.  Clicking loads search results into chat.
  * Settings: preferences (measurement units, currency, notifications), payment methods, privacy settings.

* **Middle Column (Chat Stream):**
  * Chat interface with messages from the agent and user.  Supports rich message cards (property cards, negotiation status).  Each message item can include a timestamp, speaker label, and state (e.g., “tool call running”).
  * Property cards have two views:
    * **Compact:** shows thumbnail, basic details (title, location, price, number of bedrooms), and quick actions (Expand, Favourite).  Suitable for search results list.
    * **Expanded:** full details: large photos, amenities list, score breakdown, neighbourhood info, host reviews.  Contains CTA buttons (Request details, Start negotiation, Book).  The card may embed an interactive map with property and points of interest (walkability, schools).  A “Compare” toggle adds the property to a comparison panel.
  * **Negotiation Timeline Widget:** displays timeline of offers and counter-offers with timestamps.

It updates in real-time as negotiation agent and host exchange proposals.

* **Right Column (Sandbox/VNC Panel):**
  * Visual representation of tool executions. When the agent calls a tool (search, scoring, contract generation), the sandbox shows the process: code snippets, browser actions, API requests/responses and intermediate results. The user can optionally view or hide this.
  * **Evaluator/Trace Panel:** shows LLM reasoning chain in a secure mode (only visible to developers or with user opt-in) to foster trust.
  * For voice sessions, shows live waveform, STT transcription and conversation state.

## 9.2 UI State Contract & Events API

The agent communicates with the client UI using a structured JSON protocol. Each response can include:

```json
{
  "messages": [
    { "role": "assistant", "content": "Here are some options..." },
    { "role": "tool_trace", "tool": "search_properties", "status": "running" }
  ],
  "ui_events": [
    { "type": "open_panel", "panel": "comparison", "payload": null },
    { "type": "update_property_card", "property_id": "abc123", "view": "expanded" },
    { "type": "prompt_payment_modal", "booking_id": "bkg567" }
  ],
  "property_cards": [
    { "id": "prop1", "view": "compact", "score": 0.87, "actions": ["expand", "favourite", "negotiate"] },
    { "id": "prop2", "view": "compact", "score": 0.82, "actions": ["expand", "favourite", "negotiate"] }
  ],
  "comparison_mode": { "properties": ["prop1", "prop2"], "metrics": ["price", "commute", "amenities"] }
}
```

The UI interprets `ui_events` to perform actions (open/close panels, prompt modals), and displays `property_cards` accordingly. The agent must not directly manipulate the DOM; it emits high-level events instead. On the other side, the UI sends events to the agent via the chat stream when the user clicks a card (e.g., `user_clicked_property`, `user_requested_booking`). This contract ensures decoupled UI/agent logic and supports multi-platform clients (web, mobile).

# 10. Database & Storage (Supabase + PostgreSQL)

The schema supports multi-tenancy (per organization/company) and per-role row-level policies. The `org_id` column exists in almost every table to enforce isolation. Key tables:

* **`auth.users`** – Supabase Auth table (immutable). Stores user identity and metadata.
* **`profiles`** – extends `auth.users` with role, company affiliation, name, avatar. Primary key: `id` (UUID, FK to `auth.users.id`).
* **`companies`** – host or partner companies. Columns: `id` UUID (PK), `org_type` ENUM (`host`, `data_partner`), `name`, `created_at`.
* **`company_users`** – many-to-many between profiles and companies with `role` ENUM (`owner`,

`manager`, `agent`, `partner_admin`).
* **`properties`** — canonical property listings.  Columns: `id` UUID PK, `org_id`, `partner_property_id` (nullable), `company_id` (FK), `title`, `description`, `location` (geometry point), `address`, `bedrooms`, `bathrooms`, `area_sqft`, `price_monthly`, `amenities` JSONB, `images` JSONB (list of URLs), `status` ENUM (`active`, `inactive`, `blocked`), `created_at`, `updated_at`.
* **`property_availability`** — available date ranges and nightly/weekly/monthly pricing.  Columns: `id` UUID, `property_id` FK, `start_date`, `end_date`, `price`, `currency`, `status` ENUM (`available`, `reserved`, `blocked`).
* **`property_scores`** — scoring breakdown per property per user preference segment.  Columns: `id` UUID, `property_id`, `user_id` (nullable), `score` numeric, `breakdown` JSONB (e.g., `{price:0.2, location:0.3,...}`), `created_at`.
* **`search_sessions`** — logs of user searches.  Columns: `id` UUID, `user_id`, `query`, `filters` JSONB, `results` JSONB (list of property IDs), `created_at`.
* **`negotiations`** — negotiation threads.  Columns: `id` UUID, `property_id`, `client_id`, `host_id`, `status` ENUM (`pending`, `accepted`, `rejected`, `cancelled`, `expired`), `messages` JSONB (array of {sender, message, timestamp, price}), `created_at`, `updated_at`.
* **`bookings`** — booking records.  Columns: `id` UUID, `property_id`, `client_id`, `host_id`, `start_date`, `end_date`, `total_amount`, `currency`, `status` ENUM (`pending_payment`, `pending_contract`, `confirmed`, `cancelled`, `refunded`), `payment_intent_id`, `contract_id`, `created_at`.
* **`payments`** — payment intents and captures.  Columns: `id` UUID, `booking_id`, `provider` ENUM (`stripe`, `paypal`, `local_gateway`), `intent_id`, `status` ENUM (`created`, `authorized`, `captured`, `refunded`, `failed`), `amount`, `currency`, `captured_at`, `failure_reason`.
* **`contracts`** — rental agreements.  Columns: `id` UUID, `booking_id`, `pdf_url`, `status` ENUM (`draft`, `sent`, `signed_client`, `signed_host`, `complete`), `created_at`, `updated_at`.
* **`voice_sessions`** — voice call sessions.  Columns: `id` UUID, `user_id`, `assistant_type` ENUM (`client_agent`, `host_bot`), `provider_session_id`, `started_at`, `ended_at`, `duration_secs`, `transcript_url`.
* **`agent_runs`** — logs each agent execution.  Columns: `id` UUID, `session_id`, `agent_type` ENUM (`search`, `scoring`, `negotiation`, `payment`, `contract`, `supervisor`), `input`, `output`, `status`, `error`, `created_at`.
* **`tool_calls`** — logs each tool call.  Columns: `id` UUID, `agent_run_id`, `tool_name`, `input`, `output`, `latency_ms`, `created_at`.
* **`audit_log`** — immutable ledger of sensitive actions (payments, contract signatures, KYC).  Columns: `id` BIGSERIAL, `user_id`, `action` text, `payload` JSONB, `hash` bytea, `created_at`.

### Indexes & Constraints

* Primary keys on all tables; foreign keys referencing parent tables with `on delete cascade` where appropriate (e.g., deleting a property cascades to availability and scores but not bookings).  Index on `properties.location` (PostGIS) for geospatial queries; gin index on `properties.amenities` and `search_sessions.filters` (JSONB).  Unique constraint on (`partner_property_id`, `company_id`).
* For event tables (`agent_runs`, `tool_calls`, `audit_log`), partition by month for performance.  Index on `created_at` and `session_id`.

### RLS Policies

Row-level security is enabled on all public tables【178230355922264†L270-L296】.  Example policies (pseudo-SQL):

```sql
-- Profiles: users can read/update their own profile
create policy "self service" on profiles
  for select using (id = auth.uid())
  with check (id = auth.uid());

-- Properties: hosts can manage their properties; clients can view active properties
create policy "clients view active" on properties
  for select to authenticated
  using (status = 'active');
create policy "hosts manage own" on properties
  for all to authenticated
  using (exists (select 1 from company_users cu
                 where cu.user_id = auth.uid()
                   and cu.company_id = properties.company_id
                   and cu.role in ('owner','manager','agent')))
  with check (company_id = (select cu.company_id from company_users cu where cu.user_id = auth.uid()
limit 1));

-- Bookings: clients see their own; hosts see bookings for their properties
create policy "clients read own bookings" on bookings
  for select to authenticated
  using (client_id = auth.uid());
create policy "hosts read bookings" on bookings
  for select to authenticated
  using (exists (select 1 from properties p
                 join company_users cu on cu.company_id = p.company_id
                where p.id = bookings.property_id and cu.user_id = auth.uid()));

-- Payments: clients read their payment; hosts read payments for their bookings
-- ... similar pattern ...
```

Service roles (with `bypassrls`) are used by internal services (agent orchestrator, payment processor) and never exposed to clients. Supabase JWT claims include `role`, `company_id` and `org_type`; we avoid using mutable user metadata in RLS【178230355922264†L340-L383】.

# 11. API Layer (FastAPI)

The API is **OpenAPI-first**: endpoints are defined in `openapi.yaml` and implemented in FastAPI with Pydantic models. Authentication uses Supabase JWT for browser clients; partner companies use API keys (hashed and stored in `api_keys` table with scopes). Important endpoints include:

## 11.1 Client/Chat Endpoints

| Method & Path | Auth | Request | Response | Description |
| --- | --- | --- | --- | --- |
| `POST /search` | JWT | `{ "query": string, "filters": object, "limit": int, "offset": int }` | `{ "results": [property], "next_offset": int }` | Runs `search_properties` tool and returns property summaries. |
| `GET /properties/{id}` | JWT/Anon | – | Full property details | Returns expanded property card; includes availability and score breakdown. |
| `POST /negotiations` | JWT | `{ "property_id", "message", "offer_price", "desired_dates" }` | `negotiation_id` | Starts negotiation; creates a negotiation record and sends lead to host. |
| `POST /negotiations/{id}/counter` | JWT | `{ "action": "accept"\|"reject"\|"counter", "counter_price": optional }` | Updated negotiation state | Posts counter-offer or response. |
| `POST /bookings` | JWT | `{ "property_id", "dates", "negotiation_id" }` | `booking_id`, `total_amount` | Creates a booking record in `pending_payment` state. |

| | | | | |
| --- | --- | --- | --- | --- |
| `POST /payments/initiate` | JWT | `{ "booking_id", "payment_method_id", "idempotency_key" }` | Payment intent status | Creates payment intent. |
| `POST /contracts` | JWT | `{ "booking_id" }` | Contract status, PDF URL | Generates draft contract and sends for signature. |
| `POST /payments/confirm` | JWT | `{ "booking_id", "payment_intent_id" }` | Booking status | Captures payment and finalizes booking.  Requires client confirmation and host acceptance. |
| `GET /bookings/{id}` | JWT | – | Booking details | Shows status, contract, payments and timeline. |
| `POST /search/saved` | JWT | `{ "query", "filters" }` | `saved_search_id` | Saves search; returns ID. |
| `GET /search/saved` | JWT | – | List of saved searches | |
| `DELETE /search/saved/{id}` | JWT | – | Success | Deletes saved search. |
| `GET /agent/logs` | JWT | Query params | Logs for current user | Fetches agent run and tool call logs. |

## 11.2 Voice & WebSocket Endpoints

| Method & Path | Description |
| --- | --- |
| `GET /voice/ws` | WebSocket endpoint.  Clients (web or telephony gateway) connect to stream audio and receive streaming responses.  Upgrades to voice session; returns session ID. |
| `POST /webhooks/vapi/call_started` | Receives Vapi call started event (see section 5.4). |
| `POST /webhooks/vapi/message` | Receives streaming transcripts for host bot calls. |
| `POST /webhooks/vapi/call_ended` | Receives call end event. |

## 11.3 Host & Partner Endpoints

| Method & Path | Auth | Description |
| --- | --- | --- |
| `POST /companies/{id}/properties` | JWT (host) | Create property.  Body contains property fields and images. |
| `PUT /properties/{id}` | JWT (host) | Update property. |
| `DELETE /properties/{id}` | JWT (host) | Archive property. |
| `POST /properties/{id}/availability` | JWT (host) | Create or update availability slot. |
| `GET /properties/{id}/bookings` | JWT (host/sub-agent) | List bookings for a property. |
| `GET /analytics/properties` | JWT (host) | Returns occupancy rates, revenue and other metrics. |
| `GET /calls` | JWT (host) | List voice call sessions (Vapi). |
| `POST /partner/ingest` | API key | Ingest feed data (batch or delta).  Accepts JSON/CSV; returns ingestion status. |
| `GET /partner/properties` | API key | View listing status and errors. |

## 11.4 Webhooks

The API layer exposes webhooks for external services (Stripe, Vapi, partner systems).  Webhooks include signature verification and idempotency handling.

* **`POST /webhooks/payments/stripe`** – Payment events (payment_intent.succeeded, payment_intent.payment_failed, charge.refunded).  Updates `payments` and triggers contract or refund flows.
* **`POST /webhooks/partner/inventory`** – Partner pushes updates; may be used for real-time listing changes.
* **`POST /webhooks/vapi/...`** – As above.

# 12. Payment & Escrow Flow

## 12.1 Payment Providers & PCI Strategy

We support multiple providers: **Stripe** (default), **PayPal**, and **regional gateways**.  Payment details never touch our servers; we use provider-hosted fields/SDKs.  PCI DSS compliance is ensured via provider integration.  Tokens returned by providers (payment method IDs) are stored in `payments`.  For offline/wire transfers, manual reconciliation is supported but flagged for review.

## 12.2 Reservation Holds & Escrow

* When a booking is created, the payments service creates a **payment intent** with a hold (authorization) equal to the total amount plus security deposit.  Funds are captured only after the contract is signed.
* The hold has an expiration (e.g., 7 days).  If negotiation fails or contract is not signed, the hold is automatically released.
* Upon contract completion, the payments service **captures** the payment and transfers funds into an escrow account managed by the platform.  Hosts receive payouts based on payout schedules (e.g., weekly), minus service fees.  Escrow ensures renters cannot release funds until check-in or after a defined dispute period.

## 12.3 Cancellations & Refunds

* **Before capture:** user can cancel; hold is voided.  Booking status set to `cancelled`.  Host is notified.
* **After capture but before check-in:** refund amount depends on cancellation policy (flexible, moderate, strict).  The payments service issues a partial refund via provider.  Status set to `refunded` and recorded in `payments`.
* **After check-in:** funds may be released to host.  Disputes are handled via a dispute management process; hold a portion of funds until resolution.

## 12.4 Dispute Workflow

1. Client opens dispute via API; provides evidence.
2. Payments service marks booking as `dispute_pending` and withholds payout.
3. Superadmin reviews evidence; may request host response.
4. Outcome triggers partial or full refund; logs recorded in `audit_log`.

## 12.5 Idempotency & Double-Spend Protection

Each payment-related endpoint accepts an `idempotency_key`.  The payments service stores past keys for 24 hours.  If the same key is submitted again, the original response is returned; no duplicate charges are created.  All booking & payment updates are idempotent by design.

# 13. Search & Ranking / “Is This The Right Property?” Study

## 13.1 Normalized Scoring Rubric

Properties are scored across multiple dimensions.  The default weighting can be adjusted per user preferences (explicitly specified or inferred).  Example rubric:

| Dimension | Metrics | Data sources | Weight (default) |
| --- | --- | --- | --- |
| **Price Score** | Price per square foot vs local median; total cost relative to user budget. | Market data, partner feeds | 20% |

| | | | |
|---|---|---|---|
| **Location/Commute Score** | Distance to work/school; transit options; walkability; neighbourhood quality of life (education, health care, environment)【728461639475720†L110-L140】. | GIS, traffic APIs, US News quality-of-life index | 20% |
| **Property Features** | Bedrooms, bathrooms, area, amenities (AC, parking, pool), property condition. | Partner feed, host input | 15% |
| **Lifestyle Match** | Proximity to lifestyle needs (parks, nightlife, family-friendly facilities), community style. | Knowledge graph and local data【253406740739376†L195-L232】 | 15% |
| **Terms & Flexibility** | Lease terms (minimum stay, cancellation policy), pet policy, utilities included. | Host input | 10% |
| **Reviews & Reputation** | Host rating, property reviews, response times. | User reviews | 10% |
| **Seasonality & Market Trends** | Peak vs off-peak pricing, occupancy trends. | Historical booking data | 5% |
| **Risk/Compliance** | Safety, crime rates, regulatory compliance. | Government data | 5% |

The scoring algorithm normalizes each metric to a [0,1] range, applies user-specified weights, then sums to a total score.  The algorithm also generates an explanation explaining trade-offs (e.g., “higher price but excellent location”).

## 13.2 Enrichment & Tooling

To compute the rubric, the property study pipeline (section 4.4) fetches additional data:

* **School quality & health care:** US News indexes for education and health care【728461639475720†L110-L140】.
* **Crime & safety:** local crime statistics, safety indices.
* **Walkability & transport:** WalkScore API, transit networks.
* **Utilities & legal:** Integration with regional systems (e.g., DEWA/Ejari in Dubai) to check utility registration, license status.
* **Review Analysis:** Sentiment analysis on reviews to quantify satisfaction.
* **Seasonality:** Time-series analysis of booking rates and pricing; compute seasonality factor.

## 13.3 Caching & Re-ranking

* **Caching:** Search results and scores are cached per user query in Redis/pgmem to improve latency.  Cached results are invalidated when property data or user preferences change.
* **Re-ranking:** Users can adjust weight sliders; the agent re-calls `score_property` tool to compute new scores.  Real-time re-ranking occurs client-side without re-fetching raw property data.

# 14. Agent Safety, Compliance & Guardrails

* **Forbidden Domains:** The agent must refuse tasks unrelated to property rentals (e.g., buying stocks, recommending medicines).  It must also decline to provide legal advice beyond surface definitions and refer to human experts for legal/financial matters.
* **Prompt Structure:** System prompt defines domain, allowed tools, prohibited actions, and behaviour guidelines.  Example: “You are a real-estate assistant. Only provide information about properties and rentals. When uncertain, ask clarifying questions or refer to a human.”  Developer prompt enumerates tool schemas and UI contract.
* **Fallback & Human-in-the-Loop:** If the agent encounters unknown tasks or ambiguous instructions, it calls `human_escalation` tool which alerts a human operator.  High-risk actions (payment capture, contract signing) require explicit user confirmation.
* **RLS Enforcement:** The agent cannot bypass RLS.  All tool calls go through the API which enforces per-role policies.  Service roles used by the orchestrator are restricted to necessary tables.
* **Logging & Monitoring:** All agent interactions, tool inputs/outputs and prompts are logged.  A

moderation model periodically scans conversations for sensitive content.  Users can request deletion of their logs under GDPR.
* **Rate-Limiting & Concurrency:** Per-user concurrency limits to avoid spamming tool calls. Circuit breakers prevent cascading failures; retries use exponential backoff.  Dead-letter queues capture failed tasks for later inspection.
* **Model Versioning & Rollback:** Each agent run records model version and prompt hash.  If a new model causes regressions or policy violations, superadmin can revert to the previous configuration.

# 15. Observability & Operations

* **Tracing & Metrics:** Use **OpenTelemetry** across services; instrument tool calls, DB queries and external API calls.  Export spans to **Jaeger** or **Tempo**.  Use **Langfuse** to trace LLM prompt/response pairs, tool calls and decisions.  Tag spans with session ID, user ID, tool names and latencies.
* **Structured Logs:** All services emit JSON logs to **Vector** or **Fluentbit**, enriched with request IDs and user context.  Logs are persisted in an ELK/ClickHouse stack for querying.
* **Analytics:** Collect metrics on search queries, conversion rates, negotiation lengths, payment success rates, voice call durations.  Build dashboards in **Superset** or **Metabase**.
* **Rate Limits & Circuit Breakers:** API Gateway implements per-IP and per-user rate limits.  Use token buckets; exceeders receive 429 responses.  Circuit breakers open when downstream services exceed error thresholds.  Dead-letter queue (e.g., Kafka topic) stores failed events for reprocessing.
* **Alerting:** Prometheus monitors service health; alerts (via Grafana or PagerDuty) trigger on high error rates, high latency, or unusual spikes (possible fraud).  Oncall rotation defined in runbooks.
* **Disaster Recovery:** PostgreSQL streaming replication; daily backups; point-in-time recovery. For object storage, use versioning and cross-region replication.

# 16. Testing & QA Plan

* **Unit Tests:** Each microservice has unit tests for business logic, tools, and DB functions.  Use pytest and FastAPI test client; run in CI.
* **Integration Tests:** Simulate full flows (search → negotiation → booking → payment).  Use seeded Supabase and run orchestrator against mock partner APIs.  Validate RLS policies by testing unauthorized access attempts.
* **Contract Tests:** OpenAPI schema is the contract; consumer-driven contract tests ensure API changes are backward compatible.  Use tools like `schemathesis` to fuzz endpoints.
* **End-to-End Tests:** Use Playwright to automate client UI: open the app, perform search, view property cards, start negotiation, complete payment in sandbox mode.  For voice flows, use Vapi CLI and recorded audio to simulate calls and verify transcripts.
* **Load Tests:** Use Locust or k6 to stress test search and booking endpoints; evaluate performance under 100s of concurrent users.  Voice service load tested with simulated concurrent calls.
* **Red-Team & Safety Testing:** Craft adversarial prompts to test guardrails (e.g., ask the agent to perform a bank transfer); ensure the agent refuses and escalates.  Validate that RLS prevents data leakage.
* **Synthetic Conversations:** Maintain a library of conversation transcripts and expected tool call sequences.  Run nightly regression tests on new model versions to ensure behaviour stability.

# 17. Developer Experience & Environments

* **Local Development:** `docker-compose` config spins up FastAPI services, Supabase (local), vector service, payments mock (stripe-mock), and UIs with hot reload.  Developer environment uses

`.env.development` for secrets.  Use Vite/Next.js dev server for client apps.
* **CI/CD:** Use GitHub Actions or GitLab CI.  Steps: lint (ruff, eslint), unit tests, integration tests with dockerized Supabase, build images, run OpenAPI spec validation, push to registry.  Deploy to staging environment (Kubernetes cluster) via ArgoCD.  Feature flags control release of new features.
* **Migrations:** Database schema managed via **alembic** or **supabase migration tool**. Migrations are version-controlled in `db-schema`.  Pull requests must include migration scripts and updates to RLS policies.
* **Secrets Management:** Use **Vault** or environment variables in Kubernetes secrets.  Developers use `.env.local` with dummy keys.  Service accounts with minimal privileges.
* **Developer Tools:** Provide CLI to run agent tools locally and inspect outputs; integration with Langfuse for debugging prompts; ability to replay sessions in local environment.

# 18. Security, Privacy & Data Governance

* **PII Handling:** Personal data (name, email, phone, ID documents) stored encrypted at rest using pgcrypto/pgsodium.  Access via RLS; hashed user IDs used in logs.  No raw payment data stored; tokens only.
* **KYC & AML:** For hosts, require identity verification; store KYC documents in encrypted storage; share only with payment providers as needed.
* **Encryption:** TLS enforced for all network connections.  Database encryption at rest (AES-256). Secrets stored in secure secret manager.  Passwords hashed using argon2.
* **RLS & RBAC:** RLS prevents cross-tenant data access; roles enforced at API and DB.  Service roles have minimal privileges and rotate credentials.  Use Postgres `security_invoker` views when exposing aggregated analytics□178230355922264†L372-L374□.
* **Auditing & Compliance:** All sensitive actions logged in `audit_log` with cryptographic hash and timestamp.  SOC 2 controls implemented (access logs, change management).  GDPR/CCPA compliance: data deletion requests honoured; clear privacy policy; data residency options for EU customers.
* **Vulnerability Management:** Regular security scans (Snyk, Trivy).  Penetration testing before major releases.  Incident response plan with defined escalation path.
* **Third-Party Risk:** Evaluate partner and model providers (OpenAI, Vapi, Stripe) for compliance; sign DPAs; monitor for breaches.

# 19. Incremental Roadmap & Milestones

| Phase | Duration | Owner | Key Outputs | Acceptance Tests |
| --- | --- | --- | --- | --- |
| **Phase 0:** SUNA Codebase Audit & Influence Extraction | 2 weeks | CTO & Architect | Documented learnings from SUNA architecture; list of patterns to adopt/avoid; clean-room design checklist. | Audit report approved; no code copied; patterns enumerated. |
| **Phase 1:** Core Infrastructure & RBAC | 4 weeks | Backend Lead | Monorepo scaffold; Supabase setup with users, roles, companies; RLS policies for base tables; API gateway skeleton; CI/CD pipeline. | All base tables created; RLS tests pass; API returns 401/403 correctly; local dev environment documented. |
| **Phase 2:** Client Agent & Property Ingestion | 6 weeks | AI Lead & ETL Lead | Search & ranking pipeline; agent orchestrator with search & scoring tools; property ingestion from at least one data partner; client web UI skeleton with 3-column layout. | A user can ask for “2BR in Dubai” and receive ranked results; property cards display scores; agent logs recorded. |
| **Phase 3:** Host Dashboard & VAPI Integration | 4 weeks | Product Lead & Voice Lead | Host dashboard CRUD flows; negotiation tools; Vapi integration for host CS bot; company/sub-agent roles. | Host can list property, receive leads, respond via negotiation tool; Vapi host bot answers FAQ. |
| **Phase 4:** Payments, Contracts & Escrow | 5 weeks | Payments Lead | Payment service with Stripe integration; booking & contract flows; escrow ledger; refund/dispute handling. | User can book

property, complete payment hold, sign contract, payment captured; refund scenario passes tests. |
| **Phase 5:** Voice Realtime Agent | 4 weeks | Voice Lead | Voice service using ChatGPT Realtime API; chat-supervisor pattern; low-latency streaming; barge-in handling; session handoff. | Voice call scenario: user describes property requirements, receives options, books via voice; latency within target; transcripts saved. |
| **Phase 6:** Partner Dashboards & Analytics | 3 weeks | Partner Lead & Data Lead | Data-partner dashboard; ingestion status; aggregated analytics; additional partner feeds integrated. | Partner can view feed status and booking metrics; new listings appear in search within SLA. |
| **Phase 7:** Observability & Compliance Hardening | 3 weeks | DevOps Lead & Compliance Officer | OpenTelemetry integration; Langfuse tracing; incident tooling; SOC 2 documentation; load tests; red-team tests. | All services emit traces; logs searchable; red-team tests show guardrails working; SOC 2 readiness checklist complete. |

# 20. Risk Register & Design Tradeoffs

| Risk | Mitigation | Tradeoff |
| --- | --- | --- |
| **Dependency on external models (OpenAI, Anthropic, etc.)** | Abstract the agent orchestrator to support multiple providers; implement fallback models; cache responses; monitor cost and performance. | Complexity of provider integration; may require model-agnostic prompt design. |
| **Licensing & Breaking Changes from SUNA** | Do not copy code; only reuse architectural patterns. Audit dependencies and ensure licences are compatible (MIT/Apache). | Slower initial development to reimplement features; must validate patterns independently. |
| **LLM costs & Rate Limits** | Implement caching and rank-aware search to reduce calls; use streaming to send partial responses; set budgets per session; degrade gracefully under high load. | Lower fidelity responses when cost constraints are hit; need to tune budgets. |
| **Payment Failure & Fraud** | Use idempotency keys; apply fraud detection (velocity checks, IP reputation); hold funds in escrow; require KYC for hosts; enable 3DS where available. | Additional friction for users; may delay bookings; requires dispute management. |
| **Data-Partner SLAs & Stale Inventory** | Periodic feed refresh; partner webhook notifications; display freshness date to users; fallback to real-time availability checks before booking. | Additional infrastructure for ingestion; stale data may still occur; need to handle booking failures gracefully. |
| **Voice Agent Accuracy & Latency** | Use chat-supervisor pattern and high-quality models; add pre-defined fallback phrases; monitor latency metrics; tune STT/TTS providers. | Higher compute cost; may require multiple providers. |
| **RLS Misconfiguration** | Write comprehensive tests; use Supabase RLS helpers; review policies regularly; restrict service roles; follow least-privilege principle. | Additional development overhead; potential performance impact【178230355922264†L294-L296】. |
| **Privacy & Legal Compliance** | Adhere to GDPR/CCPA; implement user data deletion; store consent; run data protection impact assessments; maintain audit logs. | Additional processes; may limit data collection/analytics. |

# 21. Appendices

## 21.1 Complete Postgres/Supabase SQL DDL (excerpt)

Due to space, below is a representative portion of the DDL.  The real schema includes all tables described in section 10.

```sql
-- Companies and company_users
create type org_type_enum as enum ('host','data_partner');
create table companies (
  id uuid primary key default gen_random_uuid(),
  org_type org_type_enum not null,
  name text not null,
  created_at timestamptz default now()
);
create type company_role_enum as enum ('owner','manager','agent','partner_admin');
create table company_users (
  company_id uuid references companies(id) on delete cascade,
  user_id uuid references auth.users(id) on delete cascade,
  role company_role_enum not null,
  primary key (company_id, user_id)
);

-- Properties
create type property_status_enum as enum ('active','inactive','blocked');
create table properties (
  id uuid primary key default gen_random_uuid(),
  org_id uuid references companies(id) on delete cascade,
  partner_property_id text,
  company_id uuid references companies(id) on delete cascade,
  title text not null,
  description text,
  location geography(point) not null,
  address text,
  bedrooms int,
  bathrooms int,
  area_sqft numeric,
  price_monthly numeric,
  amenities jsonb,
  images jsonb,
  status property_status_enum not null default 'active',
  created_at timestamptz default now(),
  updated_at timestamptz default now()
);
create index idx_properties_location on properties using gist (location);
create index idx_properties_org on properties(org_id);

-- Bookings
create type booking_status_enum as enum
('pending_payment','pending_contract','confirmed','cancelled','refunded');
create table bookings (
  id uuid primary key default gen_random_uuid(),
  property_id uuid references properties(id) on delete restrict,
  client_id uuid references auth.users(id),
  host_id uuid references auth.users(id),
  start_date date not null,
  end_date date not null,
  total_amount numeric not null,
  currency text not null,
  status booking_status_enum not null,
  payment_intent_id text,
  contract_id uuid references contracts(id),
  created_at timestamptz default now(),
  check (end_date > start_date)
);
create index idx_bookings_property on bookings(property_id);

-- Payments
create type payment_status_enum as enum ('created','authorized','captured','refunded','failed');
create table payments (
  id uuid primary key default gen_random_uuid(),
  booking_id uuid references bookings(id) on delete cascade,
  provider text not null,
  intent_id text not null,
  status payment_status_enum not null,
  amount numeric not null,
  currency text not null,
  captured_at timestamptz,
  failure_reason text,
  created_at timestamptz default now()
);
create index idx_payments_booking on payments(booking_id);

-- Agent runs & tool calls
create type agent_type_enum as enum
('search','scoring','negotiation','payment','contract','supervisor');
create table agent_runs (
  id uuid primary key default gen_random_uuid(),
  session_id uuid,
  agent_type agent_type_enum not null,
  input jsonb,
  output jsonb,
  status text,
  error text,
  created_at timestamptz default now()
);
create table tool_calls (
  id uuid primary key default gen_random_uuid(),
  agent_run_id uuid references agent_runs(id) on delete cascade,
  tool_name text not null,
  input jsonb,
  output jsonb,
  latency_ms int,
  created_at timestamptz default now()
);
```

## 21.2 Representative API Spec (OpenAPI Snippet)

```yaml
paths:
  /search:
    post:
      summary: Search properties
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                query: { type: string }
                filters: { type: object }
                limit: { type: integer, default: 20, maximum: 50 }
                offset: { type: integer, default: 0 }
      responses:
        '200':
          description: Search results
          content:
            application/json:
              schema:
                type: object
                properties:
                  results:
                    type: array
                    items: $ref: '#/components/schemas/PropertySummary'
                  next_offset: { type: integer }
  /bookings:
    post:
      summary: Create booking
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                property_id: { type: string, format: uuid }
                dates:
                  type: object
                  properties:
                    start_date: { type: string, format: date }
                    end_date: { type: string, format: date }
                negotiation_id: { type: string, format: uuid, nullable: true }
      responses:
        '201':
          description: Booking created
          content:
            application/json:
              schema:
                type: object
                properties:
                  booking_id: { type: string, format: uuid }
                  total_amount: { type: number }
                  status: { type: string }
```

## 21.3 Prompt Templates (Simplified)

**System Prompt (search agent):**

> *You are a property rental assistant. You may call tools strictly listed in the provided tool schema. You must not provide financial or legal advice. Always ask clarifying questions if the user’s request is vague. When returning property information, use concise language and reference the

UI contract.*

**Developer Prompt (tool schema excerpt):**

```yaml
tools:
  - name: search_properties
    description: Search for properties using filters and preferences
    parameters:
      type: object
      properties:
        filters:
          type: object
        preferences:
          type: object
        limit:
          type: integer
          maximum: 50
      required: [filters]
  - name: score_property
    description: Compute a detailed scoring breakdown
    parameters:
      type: object
      properties:
        property_id: { type: string }
        user_preferences: { type: object }
      required: [property_id]
```

## 21.4 Voice Agent Session Protocol

1. **Session Start:** The client (browser or phone) opens a WebSocket connection and sends an initialization message containing `user_id`, `language`, and optional `session_resume_token`.
2. **Audio Streaming:** The client streams audio data frames; the server returns interim transcriptions and final partial phrases.
3. **Agent Turn:** When the model has enough information to respond, it sends a control message (`{type: 'agent_thinking'}`) followed by audio frames of the agent’s response.
4. **Barge-In:** If the user interrupts, the client sends a `barge_in` message; the server pauses TTS and sends back the partial transcript to the agent.  After processing, the agent continues responding.
5. **End Session:** When the call ends, the server sends a `session_end` message with summary statistics (duration, sentiment, actions).  The transcript is stored and a session resume token is generated.

## 21.5 UI State Machine Contract (Simplified)

*States:* `idle` → `searching` → `results_displayed` → `property_selected` → `negotiation` → `booking_pending` → `payment_pending` → `contract_pending` → `completed` → `cancelled`.

Transitions are triggered by agent events (`results_ready`, `offer_received`, `booking_created`, `payment_confirmed`, `contract_signed`) and user events (`select_property`, `counter_offer`, `confirm_payment`, `sign_contract`, `cancel`).  Each transition updates UI components accordingly.

## 21.6 RLS Examples per Table

*See section 10 for detailed policies.*  For example, the `property_availability` table has a policy allowing hosts and assigned agents to manage availability, while clients can only read availability for active properties in their search results.

## 21.7 Event Sourcing Schema (Optional)

If adopting event sourcing for bookings/payments, each action (create booking, update status, capture payment, refund) is appended to a `booking_events` table.  The current state is derived by folding events.  This supports auditing and time travel queries.

## 21.8 Voice & Telephony Integration Implementation Plan

This section provides a **step-by-step implementation plan** for the voice services described in section 5 (client-facing voice agent) and section 5.4 (Vapi integration).  The goal is to make the plan actionable for engineers working in our monorepo.

### 21.8.1 Client-Facing Voice Agent (LiveKit + Realtime API)

1. **Provision LiveKit Cloud:**
   - Sign up for LiveKit Cloud and create a project.  Note the API key and secret used to generate access tokens.
   - Configure **rooms** for voice calls.  Each voice session will use a unique LiveKit room ID (e.g., `voice_{session_uuid}`).  Enable E2E encryption if required.

2. **Add `voice-service` microservice:**
   - In the monorepo, under `apps/voice-service/`, scaffold a FastAPI or Node service responsible for voice streaming.
   - Add dependencies: `livekit-agents` (Python or TypeScript), `openai` for the Realtime API, and our `agent-tools` library for orchestrating tool calls.

3. **Generate tokens:**
   - Implement an endpoint `POST /voice/token` that accepts `user_id` and `session_id` and returns a LiveKit JWT.  Use the LiveKit API key/secret to sign tokens with permissions `roomJoin` and `roomPublish`.
   - For each session, also generate a token for the server-side agent.

4. **Client SDK integration:**
   - In the `client-web` app, integrate the LiveKit React hooks (`@livekit/components-react`).  When a user starts a call, fetch a token from `/voice/token` and join the corresponding room.
   - Use `RoomAudioRenderer` and `useVoiceAssistant()` to capture the microphone and play back audio.
   - Display a visualizer and controls (mute, end call) in the UI.

5. **Server-side agent loop:**
   - In `voice-service`, implement a **worker** that uses LiveKit’s `MultimodalAgent` or raw `Room` API.  Connect to the room using the server token.
   - For each audio track from the user, stream audio frames to the ChatGPT Realtime API via WebSocket.  Use the `RealtimeModel` wrapper from `livekit.agents` to simplify streaming【618157445090202†L40-L58】.
   - Collect transcription results and forward them to the agent orchestrator via an internal API call (e.g., `POST /agent/voice_input` with `session_id`, `user_id`, `transcript`).
   - When the orchestrator responds with a message or indicates a tool call is in progress, send an “agent thinking” control message to the client.  Once the final response text arrives, stream it back via LiveKit using the TTS provided by the Realtime API.
   - Handle barge-in events by listening for user audio while the agent is speaking.  On barge-in,

pause TTS, send the partial transcript back to the orchestrator and resume once the new response is ready.

6. **Session lifecycle management:**
   - Persist voice sessions in the `voice_sessions` table: `id`, `user_id`, `room_id`, `status`, `started_at`, `ended_at`, `duration`, `summary`.
   - When a session starts, create a record; update `ended_at` and `duration` when the call ends; store transcripts in a related `voice_transcripts` table.
   - Generate a `session_resume_token` for handoff to the chat UI (see section 5.3).

7. **Testing:**
   - Create automated integration tests that spin up a LiveKit test server (using Docker or LiveKit's test container), connect a mock client and agent, and verify end-to-end streaming, barge-in handling and tool call handoffs.
   - Use synthetic audio samples to validate latency budget and ensure transcripts are accurate.

### 21.8.2 Host Voice Agent (Vapi AI Integration)

1. **Create Vapi assistant:**
   - For each host company, create a Vapi assistant via the Vapi dashboard or API.  Define the system prompt and allowed tool list (e.g., property availability lookup, booking status).  Configure call routing to the company's phone number or web widget.
   - Store the `assistant_id` and `company_id` in our database.

2. **Implement Vapi webhook endpoints:**
   - In `api-gateway`, add routes under `/webhooks/vapi/...` to handle events:
     * `call_started` – triggered when a call begins; create a `voice_session` record with `host_id`, `caller`, and `assistant_id`.
     * `message` – streaming transcripts of caller and assistant messages.  Forward user utterances to the host's sub-agent (if complex logic is required) or allow Vapi to handle simple FAQ directly.
     * `call_ended` – call summary, duration, outcome (resolved/escalated), call recordings.
   - Validate incoming requests using Vapi's signature verification.

3. **Secure data access:**
   - Limit data exposed to Vapi.  Provide only metadata needed for the call (property names, booking dates) and never share PII like payment details.  Enforce this via Supabase RLS policies and by constructing a minimal payload in the webhook responses.

4. **Analytics & monitoring:**
   - Record call duration, outcome (resolved, escalated, abandoned) and satisfaction score (if available) in a `host_call_metrics` table.
   - Provide dashboards in the Host UI to visualize call volumes, response times and common intents.

5. **Testing:**
   - Use Vapi's sandbox environment to simulate incoming calls and ensure our webhook handling and orchestration logic work as expected.
   - Write unit tests for webhook signature validation and data filtering.

### 21.8.3 Deployment Considerations

1. **Self-host vs. Cloud:**  For initial development, use LiveKit Cloud and Vapi's hosted services to minimize infrastructure overhead.  For production, evaluate self-hosting LiveKit if data residency is required; ensure servers are geographically distributed to achieve <100 ms latency【510217091517502†L130-L145】.

2. **Secrets management:**  Store LiveKit and Vapi API keys in a secure vault (e.g., Supabase secrets or Hashicorp Vault).  Rotate keys regularly.

3. **Scalability:**  Use LiveKit’s built-in load balancing and auto-scaling features to handle spikes in call volume【618157445090202†L127-L133】.  For Vapi, monitor quotas and configure additional phone numbers if needed.

4. **Fallback modes:**  If LiveKit or Vapi services are unavailable, fall back to text-only chat and display a message to the user.  Log incidents to the superadmin console for investigation.

# End of Plan