ChatGPT

Pet Hosting Platform MVP – System Architecture and Design

Overview

This platform is essentially an "Airbnb for pets," matching pet owners with vetted hosts who can care for pets in their homes. The Minimum Viable Product (MVP) will focus on core features needed to validate the concept and provide a smooth user experience for early adopters [1]. Key elements include robust user profiles, search and onboarding for hosts, a booking request workflow, secure in-platform communication, and payment handling. By starting with a lean feature set, the product can quickly enter the market to gather feedback, while the architecture is designed to scale for future mobile apps and international markets.

Core Features and Requirements: The MVP implements the following core features (with both pet owners and hosts as user roles):

- **User Registration & Authentication:** Users (owners and hosts) can sign up with email/password (or OAuth) and log in securely. Basic profile info (name, contact, etc.) is collected. Use JWT-based auth or session cookies for API security. Passwords are stored hashed for security. Email verification is recommended for trust.
- **Host Onboarding & Profiles:** Users can apply to become hosts by creating a host profile. During onboarding, they provide location (e.g. city or ZIP, for proximity search), a bio of their pet-care experience, availability (dates or schedule they can host), and care details (what types/sizes of pets they can host, home environment, etc.). This creates a public host profile listing that owners can browse. Hosts may also upload profile photos and optionally verification documents.
- **Search & Booking System:** Pet owners can search for hosts – filter by location, dates needed, pet size/type, etc. – to find a suitable match [1]. Owners request a booking with a host for given dates (and provide pet details). The host can accept or decline the request. Once accepted, the system captures payment from the owner in escrow. Bookings have statuses (Pending → Confirmed → Completed or Canceled). The booking workflow handles conflicts (e.g. host unavailable for overlapping booking).
- **Stripe-Powered Payments (Escrow):** The platform integrates Stripe for payments. When a booking is confirmed, the owner's payment is authorized and held in escrow (using Stripe's PaymentIntent with delayed capture or Stripe Connect's manual payouts) [2]. Funds are only released to the host after the stay is completed (minus platform fees). This escrow-like process ensures trust: owners know their payment is safe and hosts are paid after fulfilling the service. (Stripe doesn't offer true escrow accounts, but it supports holding funds and delaying payout up to 90 days for marketplace scenarios [2].) Refunds can be handled via Stripe in case of cancellations.
- **Progressive Trust Levels:** To build community trust, hosts earn badges/levels: **Trial Host** (new host), **Verified Host** (identity/background verified), **Trusted Host** (established with good reviews), and **Superhost** (top-rated, highly experienced). This is similar to how Airbnb awards a "Superhost" badge to its top-performing hosts to recognize exceptional hospitality [3] [4]. New hosts start as Trial, and as they meet criteria (e.g. complete X bookings with high ratings, verify ID, maintain low cancellation rate) they are automatically upgraded. Host profiles display these trust badges to help owners feel confident [5].

- **Photo Check-In Every 3 Hours:** During an active booking, hosts are **required** to upload photo updates of the pet at least every 3 hours. This unique feature ensures owners regularly see their pet is safe and happy. The system will send the host reminders and track compliance – if a host fails to check in, the platform can notify the owner or intervene. Photo check-ins (with timestamp) are saved to the booking record and visible to the owner for peace of mind. (For overnight stays, this could be adjusted to a reasonable schedule, but for MVP a simple 3-hour interval rule is enforced uniformly.)
- **Reviews & Ratings:** After each completed booking, both the pet owner and the host can leave a review for each other. This two-way review system (similar to Uber/Airbnb) helps maintain quality. Ratings (e.g. 1–5 stars) and comments are stored and shown on profiles. To encourage honesty, the platform may use a double-blind approach (reviews visible only after both sides submit, or after a time window). Over time, a host's average rating and review count will inform their trust level. Owners also accumulate ratings so hosts can avoid problematic clients.

Technology Stack and Architecture

Frontend: A responsive web application built with **React** (bootstrapped with Create React App or Next.js for convenience). React offers a rich component ecosystem and fast development. The UI will be localized in English and Romanian, using an i18n library (like **react-i18next** or FormatJS) so all text strings can toggle between languages. This prepares the product for international rollout. The frontend is a Single Page Application that interacts with the backend via REST API calls. It provides features like dynamic search filters, profile management forms, messaging interfaces, etc., and can be easily wrapped later as a React Native or Ionic app for mobile.

Backend: A RESTful API server built with a lightweight framework (for example, **Node.js** with Express or **Python FastAPI**). The backend is responsible for the core business logic: user authentication, handling booking requests, updating statuses, and interfacing with Stripe. The API is designed statelessly (using tokens for auth) so that future mobile apps or other clients can reuse the same endpoints. We will enforce role-based access (ensuring, for example, only a host can accept a booking request for themselves, only participants in a booking can post a review, etc.). Data validation and security (OWASP best practices) are applied on all endpoints. Integration with third-party services (Stripe, AWS S3, email/SMS services) is done through server-side SDKs.

Database: PostgreSQL is chosen as the primary data store (relational DB). Postgres is reliable, SQL-friendly, and easily hosted (e.g. via AWS RDS). It will hold structured data for users, profiles, bookings, etc. The schema is relatively small and straightforward for MVP, but designed with future growth in mind. We use foreign keys and indexes for relationships (e.g. bookings tied to users, etc.). Given the need for geo-search by location, we could use PostGIS extension or simply store city/region and use basic queries for MVP, upgrading to geo-indexing later. Data access is via an ORM (like Sequelize for Node or SQLAlchemy for Python) for productivity.

File Storage: User-uploaded images (pet photos, profile avatars, and the required check-in photos) will **not** be stored in the database (to keep it lean). Instead, they are stored on cloud object storage **AWS S3** (Simple Storage Service) and delivered via CDN. Each file's URL or key is saved in the DB for reference. This is cost-effective and scalable: S3 can handle infinite images and bandwidth, offloading that from our server [6]. For example, when a host uploads a check-in photo, the front-end will send it to an API endpoint that generates a pre-signed S3 URL or directly streams to S3, then the backend records the URL in a `checkin_photos` table. S3 ensures high durability for stored media, and we can put CloudFront in front of S3 for faster international delivery [7] [8].

Overall System Architecture: The system follows a standard three-tier web architecture with clear separation of concerns. **Frontend** UI code (React SPA) occupies the presentation layer, **backend API** with application logic serves as the middle layer, and the **database + storage** form the data layer [9]. The diagram below illustrates the high-level architecture:

High-level architecture of the pet hosting platform MVP. The React front-end (browser) communicates with a REST API backend. Static assets (and potential front-end build) can be served via AWS S3/CloudFront, while the API runs on a server (or serverless) connecting to a PostgreSQL database. User-uploaded images are stored on S3, and payments are processed through the Stripe API.

In this architecture, the React app (which can be deployed as static files on an AWS S3 bucket or a simple web server) interacts with the backend via HTTPS API calls. We can host the backend on an **AWS EC2** instance or container, fronted by an **Elastic Load Balancer** if scaling to multiple instances [10]. The backend server contains the REST API endpoints and uses drivers/ORM to communicate with the **PostgreSQL** database (e.g. an AWS RDS instance) for persistent data. For file storage, an **AWS S3** bucket holds images, and the backend generates signed URLs for the front-end to upload/download images. We integrate **Stripe API** for payments – the backend securely talks to Stripe's endpoints (using Stripe's SDK and secret keys) to create payment intents, handle webhooks, and manage payouts to hosts. External integrations (like an email service for notifications, or SMS for alerts) can be added as needed.

Security is enforced at multiple levels: front-end routes are protected for logged-in users; API endpoints require valid auth tokens; sensitive data in transit is encrypted via SSL. On the server side, we implement input validation and use parameterized queries to prevent SQL injection. We'll also use cloud security features (security groups, IAM roles for AWS resources, etc.) to lock down access.

This modular approach (separate front-end and REST API) ensures that a future mobile app can simply use the same REST endpoints for all functionality without requiring a rewrite [1]. It also allows each component to scale independently – e.g. we can put the database on a larger instance or use read replicas if needed, scale out more app servers behind the load balancer, or serve static content via a CDN for global users.

API Specification

All API endpoints will be prefixed with `/api/v1/` (for versioning). The backend returns JSON responses and uses standard HTTP status codes. Below is a list of the primary API endpoints, their methods, and their purpose:

Auth & User Accounts

- **POST** `/api/v1/auth/register` – Register a new user. Accepts user details (name, email, password, role type). Creates a new account as pet owner by default; user can later become a host through onboarding.
- **POST** `/api/v1/auth/login` – Authenticate a user and return a JWT token (or session cookie). The token will be used for subsequent requests in Authorization header.
- **POST** `/api/v1/auth/logout` – Invalidate the current session (optional for JWT, mainly for client to discard token).
- **GET** `/api/v1/users/me` – Get the authenticated user's own profile details (including whether they are a host or owner, their trust level, etc.).
- **PUT** `/api/v1/users/me` – Update profile information (name, contact info, password, etc.). Users can update their details.

(Note: For simplicity, owners and hosts share one `User` model. A user has flags/fields indicating if they are an active host. This allows one person to be both an owner and a host.)

Host Profiles & Onboarding

- **POST** `/api/v1/hosts` – Create a host profile (for the current authenticated user). This is used when a user completes the host onboarding form. It includes data like location (e.g. city or address coordinates), description/bio, pet preferences (sizes, species), availability info, and initial trust level (defaults to "Trial"). After creation, the user is marked as a host.
- **GET** `/api/v1/hosts` – Search/browse hosts. Supports query params for filtering, e.g. `?location=Bucharest&from=2025-07-01&to=2025-07-05&petType=dog`. Returns a list of host profiles available in that area and time (for MVP, availability filter might be basic). Owners use this to find potential hosts. Pagination can be implemented for long lists.
- **GET** `/api/v1/hosts/{hostId}` – Get public profile of a specific host. Includes host details, aggregate rating, trust level badge, and possibly reviews. Does not require auth to view, so owners can read about hosts before booking.
- **PUT** `/api/v1/hosts/{hostId}` – Update host profile (only allowed for the profile owner or an admin). Hosts use this to edit their bio, availability calendar, etc. They can also upload new photos of their home or such (photo uploads would use a separate endpoint or the same with multipart form data).
- **GET** `/api/v1/hosts/{hostId}/availability` – (Optional for MVP) Retrieve a host's available dates or slots. MVP might not implement a full calendar; hosts might instead mark themselves "available" or not for given dates. This endpoint would support future calendar features.

Pet Profiles (Owner's pets)

- **POST** `/api/v1/pets` – Create a pet profile for an owner's pet. Owners can save profiles of their pets (name, species, breed, age, notes like medical or behavior info, etc.). This makes it easy to include pet details in bookings.
- **GET** `/api/v1/pets` – List all pets for the authenticated owner.
- **GET** `/api/v1/pets/{petId}` – Get details of a specific pet (ensuring the requesting user is the owner).
- **PUT** `/api/v1/pets/{petId}` – Update pet info (if the current user owns the pet).
- **DELETE** `/api/v1/pets/{petId}` – Remove a pet profile (if not tied to active bookings).

(Pet profiles are not strictly required to make a booking – an owner could input pet info each time – but having them improves convenience and allows hosts to see a pet's profile during booking.)

Bookings & Payments

- **POST** `/api/v1/bookings` – Create a new booking request. Called by a pet owner when requesting a host. The body includes: hostId, petId(s) (or pet details if no profile), start datetime, end datetime, and maybe a message/introduction. The backend will calculate pricing (e.g. based on duration and host's rate) and create a Booking record in "Pending" status. It also initiates a Stripe PaymentIntent for the proposed amount. The PaymentIntent client secret can be returned if using client confirmation, but here we'll likely confirm server-side on acceptance.
- **GET** `/api/v1/bookings?role=owner/host&status=...` – List bookings relevant to the user. Owners see their requests (upcoming and past), hosts see requests made to them. Support filtering by status (pending, confirmed, etc.).

- **GET** `/api/v1/bookings/{bookingId}` – Get details of a specific booking. Includes information about the owner, host, pet, timeframe, price, and current status. Only the owner, the host, or an admin can fetch a booking.
- **PUT** `/api/v1/bookings/{bookingId}` – Update a booking's status or details. This serves multiple actions:
- A host can **accept** a pending booking (status -> "Confirmed") or **decline** it (status -> "Declined"). If accepted, the backend will confirm the payment capture with Stripe (charging the owner's card) and hold the funds. If declined, the PaymentIntent is canceled.
- An owner can cancel a booking request they made (if it's still pending or even if confirmed, subject to cancellation policy). This might trigger a refund via Stripe if already charged.
- When the end date of a booking passes, the system (or host) can mark the booking as **completed**. Completion would trigger the Stripe payout to the host's account (minus fees) from the held funds. This completion could be automated by a scheduled job as well.
- **POST** `/api/v1/bookings/{bookingId}/checkins` – (File upload) Host uploads a photo check-in for this booking. The image file is uploaded (perhaps as multipart form data or obtained via a pre-signed URL). The backend saves the file to S3 and creates a record (with timestamp and file URL). It also could notify the owner (e.g. via a WebSocket or push notification) that a new update is available.
- **GET** `/api/v1/bookings/{bookingId}/checkins` – List all check-in photo entries for the booking (for the owner to view the history of updates). Could include timestamp and photo URLs. (Alternatively, these could be included as part of GET booking details).

Stripe Webhooks: Additionally, we will set up an endpoint like **POST** `/api/v1/webhook/stripe` to receive events from Stripe (e.g. payment succeeded, failed, payout succeeded, etc.). This ensures the system can react to asynchronous payment events – for example, if a payment fails or a chargeback occurs, we could update the booking status and notify parties. The webhook endpoint will be secured with Stripe's signature secret.

Reviews & Ratings

- **POST** `/api/v1/bookings/{bookingId}/review` – Submit a review for a completed booking. The reviewer can rate 1-5 stars and leave comments. The request is attributed to the authenticated user; the backend determines the target (if a pet owner is calling this, then they are reviewing the host, and vice-versa). It creates a Review record linking to the booking, reviewer, reviewee, rating, etc. Only allowed once per party per booking. If we implement double-blind, we might hold the review until both sides have submitted, but MVP can simply allow immediate posting.
- **GET** `/api/v1/hosts/{hostId}/reviews` – Retrieve all reviews for a given host, with ratings and text (this is for displaying on a host's profile page). Similarly, **GET** `/api/v1/users/{userId}/reviews` could fetch reviews about an owner (mostly for internal use since owners' reviews might not be public to all, but hosts might see them).
- **DELETE** `/api/v1/reviews/{reviewId}` – (Admin or maybe allow edit) Remove or edit a review. In MVP, editing reviews might not be allowed through API (maybe contact support), but an admin interface could remove inappropriate content.

Messaging & Notifications

(Optional MVP feature – if time allows)
To facilitate communication, we could include a basic messaging system:
- **POST** `/api/v1/messages` – Send a message to another user regarding a booking (or general inquiry). Body includes the recipient userId (or bookingId context), and message text. Typically,

messages would be allowed only between an owner and host who have an ongoing negotiation or booking.
- **GET** `/api/v1/messages?bookingId=X` – Get the message thread for a specific booking (for contextual chat). Or `/messages?withUser=Y` for direct user-to-user messages if implemented.

For notifications: the system will send email notifications for key events (booking request, booking accepted, upcoming booking start, missed photo check-in, etc.). These may be handled by an async job that calls an email service (like Amazon SES or SendGrid). Real-time notifications can be added via WebSockets or polling in the future, but for MVP, email (and maybe SMS for urgent things like no check-in) is sufficient.

Each of these API endpoints corresponds to a well-defined function in the backend. The API will have proper error handling (returning 400 for bad requests, 401 for unauthorized, 404 for not found, etc.). Documentation will be provided (possibly using OpenAPI/Swagger) so front-end and future integrations know how to use the endpoints.

Database Schema (PostgreSQL)

The database uses a normalized schema capturing users, hosts, pets, bookings, payments, etc. Below is a proposed **lightweight but extensible schema**:

- **Users** – Stores all user accounts (both owners and hosts).
- `id` (PK), `name`, `email` (unique), `password_hash`, `phone`, `role` (enum: owner, host, both), `is_host` (boolean flag or separate role table), `created_at`, `email_verified` (boolean).

- **Note:** Rather than separate tables for owners and hosts, one Users table can cover both, with additional host details in a HostProfile table. This avoids duplicate accounts if someone is both.

- **HostProfile** – Details specific to hosts (1-to-1 with Users for those who are hosts).

- `user_id` (PK, FK to Users), `location` (could be a text address or lat/long), `bio` (text), `experience` (text or years), `average_rating` (computed, can also calculate on the fly), `review_count`, `trust_level` (enum: Trial, Verified, Trusted, Superhost), `verification_status` (maybe stores if ID/background check done), `is_active` (bool if host is currently taking bookings), `created_at`.

- *This table is created when a user becomes a host. Trust level updates over time based on criteria (could be updated via triggers or logic after each booking/review).*

- **Pets** – Pet profiles (owned by users).

- `id` (PK), `owner_id` (FK to Users), `name`, `type` (e.g. Dog/Cat – could use an enum or separate Species table), `breed`, `age` or `birth_date`, `notes` (text for special needs, etc.), maybe `photo_url` for a profile picture.

- A user can have multiple pets. These are referenced when making bookings.

- **Bookings** – Represents a pet stay request and its lifecycle.

- `id` (PK), `owner_id` (FK Users), `host_id` (FK Users, the host), `start_time`, `end_time`, `status` (enum: Pending, Confirmed, Completed, Cancelled, Declined), `total_price` (decimal), `payment_intent_id` (store Stripe PaymentIntent or charge ID for reference), `payout_id` (Stripe payout reference if needed), `created_at`, `updated_at`.

- **Relationships:** One booking can be associated with one or multiple pets. We handle this via a join table **BookingPets**: with `booking_id` and `pet_id` pairs (if an owner includes multiple pets in a single booking). Alternatively, if we restrict one pet per booking initially (to simplify MVP), we can include `pet_id` directly in Bookings (and support multiple later). The booking contains the agreed price (which could depend on number of pets and duration).

- **BookingPets** – (If needed) Associates bookings with pets (many-to-many).

- Composite PK of `booking_id` + `pet_id`. This allows multiple pets in one booking record. Also could include `pet_count` or details if needed (e.g. 2 dogs).

- **Payments** – (Optional separate table) Track payment transactions.

- `id` (PK), `booking_id` (FK), `amount`, `currency`, `status` (Authorized, Captured, Refunded, PaidOut, etc.), `stripe_payment_intent` (or charge ID), `stripe_transfer_id` (for payout to host), `created_at`.

- We might not need a separate table if the Booking holds most payment info, but it can be useful for audit or if multiple transactions (e.g. partial payments or refunds). For MVP, a simple approach is: booking status and fields cover payment state (e.g. if `status=Confirmed` implies payment authorized, `status=Completed` implies payment captured and released).

- **CheckInPhotos** – Stores the periodic photo uploads for active bookings.

- `id` (PK), `booking_id` (FK), `timestamp` (when uploaded), `photo_url` (text, the S3 link or key), `comment` (optional text note).

- There will typically be multiple entries per booking (one per check-in occurrence). This table will be queried to ensure at least one entry per required interval. We could also store `posted_by` (should always be host in this case).

- **Reviews** – Reviews given after bookings.

- `id` (PK), `booking_id` (FK, to ensure one review per party per booking), `reviewer_id` (FK Users), `reviewee_id` (FK Users – the person being reviewed), `role_of_reviewer` (or we infer by comparing reviewer vs host/owner in booking), `rating` (int 1-5), `comment` (text), `created_at`.

- We expect at most two reviews per booking (one each side). We might enforce `UNIQUE(booking_id, reviewer_id)` to prevent duplicates.

- **TrustLevelHistory** – (Optional) If we want to log when a host levels up.

- `id`, `user_id`, `new_level`, `changed_at`, `notes` (e.g. "automatically promoted to Trusted after 5 five-star reviews"). This is not required, but could be nice for auditing. Alternatively, just keep the current level in HostProfile and compute criteria on the fly.

- **Messages** – (If messaging implemented) Stores messages between users.

- `id`, `sender_id` (FK Users), `receiver_id` (FK Users), `booking_id` (FK, nullable if conversation tied to a booking), `message_text`, `sent_at`.

- For simplicity, messages could be tied to bookings (one conversation per booking request). A more complex approach is a separate Conversation thread table, but MVP can just use booking ID as context.

- **Verification** – (Optional for verifying hosts) Store verification info.

- E.g. `id`, `user_id`, `type` (e.g. ID_check, background_check), `status` (pending, verified, failed), `document_url` (if storing IDs on S3), `requested_at`, `verified_at`.
- This could support the "Verified Host" badge. For MVP, it might be manual (admin verifies offline), but the schema can accommodate it for future automation (via a service like Onfido integration).

All primary keys are UUID or auto-increment integers. Foreign keys ensure referential integrity (e.g., a Booking can't reference a non-existent User or Pet). We will also add indexes such as: an index on `Bookings(host_id)` for hosts to quickly find their bookings, on `Bookings(owner_id)` for owners, on `HostProfile(location)` to speed up location-based searches (perhaps using a trigram or PostGIS index if doing radius queries). The schema is designed to be extensible: for example, adding a table for host payout transactions or adding fields for more host attributes (like pricing, which in MVP might be a flat rate per night stored in HostProfile or a separate Rate table).

The **lightweight nature** of this schema keeps the initial development simple, while the clear separation of concerns (users, pets, bookings, payments, reviews) will allow easy modifications. For instance, supporting coupon codes or promotions would mean adding a new table and linking it to bookings or payments without altering existing tables. PostgreSQL's reliability and features (such as JSON columns if we need to store flexible data like pet characteristics) make it a solid choice for this structured data [11].

Cloud Infrastructure and Deployment

For a lean MVP, we aim for cost-effective hosting that can scale with demand. Here are the recommendations:

- **AWS Hosting:** Leverage Amazon Web Services for an integrated solution. We can deploy the React front-end as static files on **Amazon S3** and serve it via **Amazon CloudFront** (CDN) for low-latency access worldwide [7]. The REST API can run on a small **Amazon EC2** instance (e.g. a t3.small) or container (using AWS Fargate or Elastic Beanstalk for simplicity). For scalability and high availability, we'd put the API servers behind an **Elastic Load Balancer**, allowing us to add more instances as traffic grows [10]. The database will run on **Amazon RDS (PostgreSQL)**, which manages backups, replication, and upgrades, ensuring our data is safe and easily scalable (we can start with a single-db instance and later move to a Multi-AZ deployment for high availability) [11]. User-uploaded images and other media are stored on **Amazon S3**, which is optimized for serving binary content and scales virtually infinitely [6]. We will use S3's durability and Lifecycle policies (e.g. moving older images to Glacier for cost savings if needed long-term).

- **Stripe Integration:** We utilize Stripe's cloud service – no self-hosted payment needed. Stripe handles the heavy lifting of payments security (PCI compliance). We just need to securely store our Stripe API keys and implement the server-side Stripe flows. We will use **Stripe Connect** if we

plan to send payouts to hosts' bank accounts. In that case, each host is onboarded as a Connected Account under our Stripe platform [2]. For MVP (to avoid complex KYC requirements), we might choose a simpler route: the platform collects payment from owners and temporarily holds it (in Stripe balance), then manually transfers to hosts. This is configurable via Stripe and keeps us lean – as Stripe's docs note, using "manual" payouts can emulate an escrow-like delay [2].

- **Deployment Pipeline:** Use an Infrastructure-as-Code approach (like CloudFormation or Terraform) to provision resources. Continuous Integration (CI) can automate tests, and Continuous Deployment (CD) can push updates. For instance, when we push front-end code to main branch, a GitHub Action could build the React app and deploy to S3/CloudFront. Similarly, backend code changes can be containerized (Docker) and deployed to an AWS ECS service or directly to an EC2 instance via SSH in early stages. Keeping things automated ensures rapid iteration.

- **Storage and Backups:** Besides the primary Postgres database (whose automated snapshots will be enabled via RDS), all critical user data like images on S3 should be backed up (S3 is inherently redundant across availability zones, and we can enable versioning to recover deleted files). Logs from the application can be sent to **CloudWatch** for monitoring. For the photo check-in requirement, since images are on S3, we just log references in the DB – S3's durability protects the actual files. Any critical configuration (API keys, etc.) will be stored in secure storage (AWS Secrets Manager or at least environment variables not checked into code).

- **Cost considerations:** For MVP on AWS, we can use minimal resources: e.g., one small EC2 instance or utilize AWS's free tier (a t2.micro, which might suffice for initial user counts), a small RDS instance (db.t2.micro which is often free-tier eligible for 12 months), and S3/CloudFront where costs will be negligible at low usage (and easily scaled as needed). CloudFront can be toggled on once international traffic grows; initially, serving directly from S3 or the EC2 static server might be fine to save cost. We will also consider using AWS **S3 for static front-end** hosting vs. a service like Netlify or Vercel which could be even simpler for deploying the React app. However, since we already use S3 for user uploads and want everything in one cloud, using S3 for the website is straightforward.

- **Analytics and Monitoring:** Use AWS CloudWatch for server health metrics and logs. Set up alarms (e.g. if CPU is high or if memory/disk is running out on EC2, or if error rates spike). Utilize Stripe's dashboard for monitoring payment volume and any payment failures. In future, integrate an APM (Application Performance Monitoring) like New Relic or Datadog for deeper insight, but CloudWatch and simple logging suffice for MVP.

- **Scaling Strategy:** Though the MVP will run on minimal infrastructure, we design with scalability in mind. The stateless REST API can be cloned behind load balancers easily. The database can be vertically scaled (to a larger instance) or read-replicas added if read-heavy. We separate components (app server vs static content vs database) so that each can scale independently – a fundamental best practice for web apps [12]. Caching layers (like Redis/ElastiCache) can be introduced later to offload reads (for example, caching often-read data like host search results or session data if needed). For now, using in-memory caching on the server or simple CDN caching for static resources is enough.

- **Development & Testing Environment:** We'd likely use a staging environment that mirrors production on a smaller scale. This could mean a separate RDS instance or just a local Postgres for testing, and perhaps Stripe's test mode for payments in staging. This ensures we can safely

test new features (like the photo check-in upload flow or Stripe integration) without impacting real users.

- **Future Mobile & Internationalization:** The chosen architecture (React front-end + REST API) makes it straightforward to build native mobile apps later – they will use the same REST endpoints over HTTPS. We might introduce a GraphQL API in the future to optimize data fetching for mobile, but the REST design is a solid starting point. International rollout will involve deploying in additional AWS regions or using a CDN heavily. For example, if we get many European users (given Romanian support), hosting the backend in EU (e.g. AWS Frankfurt region) might reduce latency. CloudFront will ensure static assets and possibly API (if using Lambda@Edge or similar) are quick globally. We will also need to handle multi-currency in payments (Stripe can automatically handle charges in different currencies to some extent). Our database should store locale for users so we can send locale-specific communications and handle things like date/time formatting properly.

In summary, the system is designed to fulfill all listed requirements while keeping the implementation **lean and cost-effective**. The architecture uses proven technologies and cloud services that provide out-of-the-box scalability (e.g. AWS S3 for infinite storage, Stripe for global payments). By building a solid foundation with a clear separation of concerns, we ensure that as the platform grows (more users, mobile apps, new features like insurance or advanced search), the core doesn't need a rewrite – we can extend it step by step. This design emphasizes trust and safety (through verification levels, escrow payments, regular photo updates) which are critical in the pet hosting domain, all while maintaining a pragmatic scope for the MVP. The choices made balance immediate practicality with future-readiness, aligning with the goal of a scalable platform that can be iteratively improved.

Sources:

- ITRex Group – *Pet Sitting App MVP Features* [1]
- Airbnb Community – *Superhost Program Criteria* [3] [4]
- Hospitable Blog – *Trust Badges on Profiles* [5]
- Stripe Documentation – *Escrow-like Delayed Payouts* [2]
- AWS Prescriptive Guidance – *SPA Deployment Architecture* [7] [8]
- AWS Architecture Basics – *Three-Tier App Components* [9]
- AWS Reference – *Web App on AWS (EC2, ELB, S3, RDS)* [10] [11]

---

[1] How Much Does an MVP Cost in 2024? — ITRex
https://itrexgroup.com/blog/how-much-does-mvp-cost/

[2] Controlling bank and debit card transfers | Stripe Documentation
https://docs.stripe.com/connect/legacy-transfers

[3] [4] [5] Airbnb Badges You Can Earn as a Host | Hospitable
https://hospitable.com/airbnb-badges/

[6] [9] [12] Basics of AWS Architecture Diagram | by Farzana Afrin Tisha | Medium
https://farzanaafrintisha.medium.com/basics-of-aws-architecture-diagram-278563b9cfd1

[7] [8] Deploy a React-based single-page application to Amazon S3 and CloudFront - AWS Prescriptive Guidance
https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/deploy-a-react-based-single-page-application-to-amazon-s3-and-cloudfront.html

10 11 AWS Web Application Reference Architecture template - Cloudairy Template
https://cloudairy.com/template/web-application-architecture-diagram-template/