ShareYourSpace 2.0 MVP - Cursor Development Guide

Version: 1.0 (Based on mvp-plan-v3.md v1.2) **Date:** 2025-04-14 **Goal:** Build the fully functional ShareYourSpace 2.0 MVP using Cursor AI, following the defined feature list.

Introduction: This guide provides step-by-step instructions to develop the ShareYourSpace 2.0 MVP. It heavily relies on using the Cursor IDE and its AI capabilities for accelerated development. Follow each step carefully. Ensure you have the necessary prerequisite software installed and accounts created before starting.

Prerequisites:

- Cursor IDE installed and configured.
- Node.js (latest LTS) and npm/yarn.
- Python (3.10+).
- Docker and Docker Compose.
- PostgreSQL (accessible locally via Docker or a cloud service).
- Git and a GitHub/GitLab account.
- Accounts for: Google Cloud (for Cloud SQL, Cloud Run, Vertex AI, APIs), Vercel, Stripe, SendGrid/Mailgun (or similar email provider).

Phase 0: Preparation & Foundational Setup

Goal: Set up the development environment, initialize projects, and configure basic structures.

Step 0.1: Environment Setup

- **Manual Action:** Verify all prerequisite software (Node, Python, Docker, Git, PostgreSQL client) is installed and working.
- **Manual Action:** Create accounts for GitHub/GitLab, Google Cloud, Vercel, Stripe, SendGrid/Mailgun. Obtain necessary API keys/credentials where applicable (keep them secure).

Step 0.2: Project Initialization

- **Manual Action:** Create a main project directory, e.g., `shareyourspace-mvp`.
- **Manual Action:** Open this directory in Cursor.
- **Manual Action:** Open Cursor's integrated terminal.
- **Manual Action (Backend):**
  - `mkdir sys2-backend && cd sys2-backend`
  - `python -m venv venv`
  - Activate virtual environment:
    - macOS/Linux: `source venv/bin/activate`
    - Windows: `.\venv\Scripts\activate`
  - `pip install fastapi "uvicorn[standard]" python-dotenv sqlalchemy psycopg2-binary alembic pydantic[email]`
  - `pip freeze > requirements.txt`
  - Create `.gitignore` file.

> **Cursor Prompt (@Chat in backend root):** "Generate a standard Python .gitignore file, including entries for virtual environments, pycache, IDE files, and .env files."

- Create `main.py`.

> **Cursor Prompt (@Chat in backend root):** "Generate a minimal FastAPI application structure in `main.py` with a simple root endpoint ('/') that returns {'message': 'ShareYourSpace 2.0 Backend'}."

- **Manual Action (Frontend):**
  - Navigate back to the root project directory in the terminal (`cd ..`).
  - `npx create-next-app@latest sys2-frontend --typescript --tailwind --eslint --app` (Follow prompts, select defaults).
  - `cd sys2-frontend`
  - `npm install zustand framer-motion @radix-ui/react-icons`
  - `npm install --save-dev @types/node @types/react @types/react-dom`

> **Cursor Prompt (@Chat in frontend root):** "Suggest a scalable folder structure for a Next.js 14 App Router project. Include directories for components, contexts/stores, hooks, lib/utils, styles, and API interaction logic."

  - *(Action: Review Cursor's suggested structure and create the basic folders like `components`, `stores`, `hooks`, `lib` inside the `app` or root directory as preferred).*
- **Manual Action (Git):**
  - `cd ..` (back to root `shareyourspace-mvp`)
  - `git init`
  - `git add .`
  - `git commit -m "Initial project setup for backend and frontend"`
  - Create a repository on GitHub/GitLab and push the initial commit.

Step 0.3: Basic Platform Structure & Configuration ([CP-01])

- **Manual Action (Backend):**
  - In `sys2-backend`, create a `.env` file. Add basic variables (replace placeholders):

```
DATABASE_URL=postgresql://user:password@host:port/dbname
SECRET_KEY=your_very_strong_random_secret_key # Generate a secure random key
# Add other secrets later (Email API key, Stripe keys, Google keys etc.)
```

  - **Manual Action:** Add `.env` to `sys2-backend/.gitignore` if not already present.
  - Create `config.py`.

> **Cursor Prompt (@Edit `config.py`):** "Create a Pydantic `BaseSettings` class to load environment variables from the `.env` file. Include `DATABASE_URL` and `SECRET_KEY` initially."

  - Modify `main.py` to use the config.

> **Cursor Prompt (@Edit `main.py`):** "Import the settings from `config.py` and make them accessible, potentially via dependency injection or a global instance."

- **Manual Action (Frontend):**
  - In `sys2-frontend`, create a `.env.local` file. Add:

```
NEXT_PUBLIC_API_BASE_URL=http://127.0.0.1:8000 # Default backend URL
for local dev
# Add other public keys later (Stripe public key etc.)
```

  - **Manual Action:** Add `.env.local` to `sys2-frontend/.gitignore`.
  - **Manual Action:** Review `tailwind.config.ts` and `app/globals.css`. Adjust base styles if needed.

Step 0.4: UI Component Library Setup ([CP-02] - Part 1)

- **Manual Action (Frontend):**
  - In the `sys2-frontend` directory terminal:
  - `npx shadcn-ui@latest init` (Follow prompts: Choose style, base color, CSS variables, `tailwind.config.js`, components alias like `@/components`, utils alias like `@/lib/utils`, React Server Components: Yes).
  - `npx shadcn-ui@latest add button input card label dialog dropdown-menu popover sheet toast`
  - Create `app/components/layout/MainLayout.tsx`.

> **Cursor Prompt (@Edit `app/components/layout/MainLayout.tsx`):** "Generate a basic reusable layout component using Shadcn UI components. It should accept `children` as props and include a simple header or placeholder area."

  - Modify `app/layout.tsx` to use `MainLayout` if desired, or use `MainLayout` within specific page layouts later. Add `Toaster` component here for global toasts.

> **Cursor Prompt (@Edit `app/layout.tsx`):** "Import and add the Shadcn `Toaster` component within the main layout body."

Step 0.5: Database Setup & ORM ([CP-04] - Part 1)

- **Manual Action (Database):** Start your PostgreSQL instance (e.g., via Docker: `docker run --name sys-db -e POSTGRES_PASSWORD=yourpassword -e POSTGRES_USER=youruser -e POSTGRES_DB=sysdb -p 5432:5432 -d postgres`). Update `.env` with correct credentials.
- **Manual Action (Backend):**
  - Create `database.py`.

> **Cursor Prompt (@Edit `database.py`):** "Generate SQLAlchemy setup code. Include `create_engine` using `DATABASE_URL` from config, `sessionmaker` for creating sessions, and a `declarative_base`. Also include a function `get_db` for FastAPI dependency injection."

  - `alembic init alembic`
  - Configure `alembic.ini`: Set `sqlalchemy.url = ${DATABASE_URL}` (or load from config).

- Configure `alembic/env.py`:
  - Import your `Base` from `database.py`.
  - Set `target_metadata = Base.metadata`.
  - Ensure it can load the database URL correctly (might need to import your config).

> **Cursor Prompt (@Chat):** "How do I configure `alembic/env.py` to correctly import my SQLAlchemy Base metadata and database URL from my `config.py` settings?" *(Follow Cursor's guidance to adjust `env.py`)*.

- Create `models/user.py`.

> **Cursor Prompt (@Edit `models/user.py`):** "Generate a SQLAlchemy User model class inheriting from the Base in `database.py`. Include columns: `id` (Integer, primary key), `email` (String, unique, index), `hashed_password` (String), `full_name` (String, nullable), `role` (String - e.g., 'CorporateAdmin', 'StartupAdmin', 'StartupMember', 'Freelancer', 'SysAdmin'), `status` (String - e.g., 'Waitlisted', 'PendingOnboarding', 'Active', 'Inactive'), `created_at` (DateTime, default now), `updated_at` (DateTime, default now, onupdate now). Use appropriate SQLAlchemy types and constraints."

- Ensure your `models/user.py` (and future models) are imported somewhere Alembic can see them (e.g., create `models/__init__.py` and import models there, then import `models` in `alembic/env.py`).
- **Manual Action:** Run `alembic revision --autogenerate -m "Initial user model"` in the `sys2-backend` terminal. Check the generated migration script in `alembic/versions/`.
- **Manual Action:** Run `alembic upgrade head` to apply the migration to your database. Verify the table creation in your DB.

Step 0.6: Commit Progress

- **Manual Action:**
  - `git add .`
  - `git commit -m "Feat: Phase 0 - Foundational setup, DB init, UI Lib"`
  - `git push`

Phase 1: Core Authentication & Onboarding Flow

Goal: Implement user registration, email verification, login, and initial onboarding screens.

Step 1.1: User Registration ([ON-01])

- **Manual Action (Backend):**
  - `pip install passlib[bcrypt]` (Add to `requirements.txt`).
  - Create `core/security.py`.

> **Cursor Prompt (@Edit `core/security.py`):** "Generate functions for password hashing using passlib bcrypt: `verify_password(plain_password, hashed_password)` and `get_password_hash(password)`."

- Create `schemas/user.py`.

> **Cursor Prompt (@Edit `schemas/user.py`):** "Generate Pydantic schemas for user operations: `UserBase`, `UserCreate` (includes password, role, potentially company name), `User` (response model, excludes password)."

- Create `routers/auth.py`.

> **Cursor Prompt (@Edit `routers/auth.py`):** "Generate a FastAPI router endpoint `POST /auth/register`. It should: * Accept a `UserCreate` schema. * Check if a user with the given email already exists (query the DB). Raise HTTPException 400 if exists. * Hash the password using `get_password_hash`. * Determine the initial `status` ('Waitlisted' for Startup/Freelancer, 'PendingOnboarding' for Corporate). * Create the new User record in the database. * Return the created user using the `User` schema (or a simple success message)."

- Modify `main.py` to include the auth router.

- **Manual Action (Frontend):**
  - Install form handling libraries: `npm install react-hook-form @hookform/resolvers zod`
  - Create registration forms:
    - `app/(auth)/signup/corporate/page.tsx`
    - `app/(auth)/signup/startup/page.tsx`
    - `app/(auth)/signup/freelancer/page.tsx`

> **Cursor Prompt (@Edit `app/(auth)/signup/startup/page.tsx`):** "Generate a registration form using Shadcn UI components (`Card`, `Input`, `Button`, `Label`), `react-hook-form`, and `zod` for validation. Include fields for Full Name, Email, Password, Company Name, Role (pre-filled/hidden 'StartupAdmin'). Handle form submission to call a (placeholder) API function `registerUser`." *(Adapt this prompt for Corporate and Freelancer forms)*.

- Create `lib/api/auth.ts`.

> **Cursor Prompt (@Edit `lib/api/auth.ts`):** "Generate an async function `registerUser(data)` that takes registration data, makes a POST request to `${process.env.NEXT_PUBLIC_API_BASE_URL}/auth/register`, and handles success/error responses (e.g., using try/catch and showing toasts)."

- Connect the form submission logic to call `registerUser`.

Step 1.2: Email Verification ([ON-03]) & Basic Email Service ([CP-05])

- **Manual Action (Backend):**
  - Choose and install email provider SDK (e.g., `pip install sendgrid`). Add to `requirements.txt`.
  - Add email API key to `.env`. Update `config.py`.
  - Create `utils/email.py`.

> **Cursor Prompt (@Edit `utils/email.py`):** "Generate a function `send_email(to_email: str, subject: str, html_content: str)` using the SendGrid SDK (or chosen provider) to send emails. Load API key from config."

- Modify `core/security.py` (or create `core/tokens.py`).

> **Cursor Prompt (@Edit `core/security.py`):** "Generate functions `create_verification_token(email: str)` and `verify_verification_token(token: str)`. Use JWT with a short expiry and a specific subject/audience for verification purposes. Use `SECRET_KEY` from config."

    - Modify `routers/auth.py`:
        - After successful user creation in `/register`:
            - Generate a verification token.
            - Create verification URL (e.g., `http://yourfrontend.com/auth/verify/{token}`).
            - Generate email HTML content (simple template).
            - Call `send_email` to send the verification link.
    - Add a new endpoint to `routers/auth.py`.

> **Cursor Prompt (@Edit `routers/auth.py`):** "Generate a FastAPI endpoint `GET /auth/verify/{token}`. It should: * Call `verify_verification_token` with the token from the path. * If valid, find the user by email from the token payload. * If user found and status is 'Waitlisted' or 'PendingOnboarding', update their status to 'ActiveWaitlist' or 'ActivePending' (or similar distinct status indicating verified but not fully onboarded). * Return success or error message."

- **Manual Action (Frontend):**
    - Create `app/(auth)/verify-email/page.tsx`.

> **Cursor Prompt (@Edit `app/(auth)/verify-email/page.tsx`):** "Create a simple page informing the user to check their email for a verification link."

    - Create `app/(auth)/verify/[token]/page.tsx`.

> **Cursor Prompt (@Edit `app/(auth)/verify/[token]/page.tsx`):** "Create a page component that: * Extracts the `token` from the URL parameters. * On component mount (using `useEffect`), calls a new API function `verifyEmail(token)` (define this in `lib/api/auth.ts` to call the backend `GET /auth/verify/{token}`). * Displays loading state, success message (e.g., 'Email verified! Redirecting...'), or error message based on the API response. * Redirects to the appropriate next step (e.g., login or waitlist page) on success."

Step 1.3: Login System & Auth ([ON-04], [ON-02] - Basic)

- **Manual Action (Backend):**
    - `pip install python-jose[cryptography]` (Add to `requirements.txt`).
    - Modify `core/security.py`.

> **Cursor Prompt (@Edit `core/security.py`):** "Generate functions `create_access_token(data: dict, expires_delta: timedelta = None)` and potentially `create_refresh_token`. Use JWT with appropriate expiry times (e.g., 15 mins for access, 7 days for refresh). Include logic to handle token decoding and validation, raising `CredentialsException` if invalid. Define `ALGORITHM` and load `SECRET_KEY`."

    - Modify `schemas/auth.py` (or create it).

> **Cursor Prompt (@Edit `schemas/auth.py`):** "Generate Pydantic schemas: `Token` (with `access_token`, `token_type`, potentially `refresh_token`), `TokenData` (payload structure, e.g., `email` or `user_id`)."

- Modify `routers/auth.py`.

> **Cursor Prompt (@Edit `routers/auth.py`):** "Generate a FastAPI endpoint `POST /auth/login` using `OAuth2PasswordRequestForm`. It should: * Authenticate the user (find user by email, verify password using `verify_password`). Raise HTTPException 401 if invalid. * Create access (and refresh) tokens using `create_access_token`. * Return the tokens in the `Token` schema."

- Implement "Forgot Password" Flow:

> **Cursor Prompt (@Edit `routers/auth.py`):** "Generate endpoint `POST /auth/forgot-password`. Accepts email. Generates a password reset token (similar to verification token, maybe different secret or subject). Sends email with reset link."
> **Cursor Prompt (@Edit `routers/auth.py`):** "Generate endpoint `POST /auth/reset-password`. Accepts token and new password. Verifies token, finds user, updates hashed password."

- **Manual Action (Frontend):**
  - Create `app/(auth)/login/page.tsx`.

> **Cursor Prompt (@Edit `app/(auth)/login/page.tsx`):** "Generate a login form using Shadcn UI, `react-hook-form`, `zod`. Fields: Email, Password. Handle submission to call `loginUser` API function."

- Modify `lib/api/auth.ts`.

> **Cursor Prompt (@Edit `lib/api/auth.ts`):** "Generate `loginUser(data)` function. Makes POST request to `/auth/login` with form data. On success, securely store tokens (e.g., in httpOnly cookies managed by the backend response header `Set-Cookie`, or if needed client-side, use state/localStorage carefully - server-set cookies preferred)."

- Set up global auth state: Create `stores/authStore.ts`.

> **Cursor Prompt (@Edit `stores/authStore.ts`):** "Generate a Zustand store for authentication. Include state for `user` (null or user object), `isAuthenticated`. Include actions `login(userData)` and `logout()`."

- Implement protected routes: Create `components/auth/ProtectedRoute.tsx` or use middleware (`middleware.ts` in root).

> **Cursor Prompt (@Chat):** "How can I implement protected routes in Next.js 14 App Router using Zustand for auth state? Show an example using either a wrapper component or middleware." *(Choose an approach and implement)*.

- Implement Logout button/functionality (clears state/tokens).
- Create `app/(auth)/forgot-password/page.tsx` and `app/(auth)/reset-password/[token]/page.tsx` forms and connect them to the backend endpoints via

functions in `lib/api/auth.ts`.
    - **(Optional MVP - Social Login [ON-02]):** This adds complexity.
        - **Backend:** Install `authlib` or `fastapi-sso`. Configure providers (Google, etc.) with client IDs/secrets in `.env`. Add endpoints for `/login/{provider}` and `/auth/{provider}/callback`. Ask Cursor for help integrating a specific provider (e.g., `"@Chat Show FastAPI integration example for Google OAuth using Authlib"`).
        - **Frontend:** Add social login buttons triggering the `/login/{provider}` backend URL. Handle the callback redirection.

Step 1.4: Waitlist/Corporate Onboarding Info Pages ([ON-05], [ON-06])

- **Manual Action (Frontend):**
    - Create `app/waitlist/page.tsx`.

> **Cursor Prompt (@Edit `app/waitlist/page.tsx`):** "Create a page component displaying the waitlist message for Startups/Freelancers as described in [ON-05]. Should be shown after successful email verification if their status indicates waitlisted." *(Routing logic needed after verification/login to direct users here).*

    - Create `app/onboarding-pending/page.tsx`.

> **Cursor Prompt (@Edit `app/onboarding-pending/page.tsx`):** "Create a page component displaying the 'Contact You Soon' message for Corporate signups as described in [ON-06]." *(Routing logic needed).*

Step 1.5: Simple In-App Onboarding Guidance ([ON-07])

- **Manual Action (Frontend):**
    - *(Implement this after users can be fully activated and access the main dashboard - Phase 5).*
    - Choose a library for tooltips/tours (e.g., `react-joyride`) or use simple conditional rendering with Shadcn Dialog/Popover.
    - Trigger this guidance component only on the *first* login after a user's status becomes fully 'Active' (requires backend to track this, maybe an `is_onboarded` flag on the User model).

> **Cursor Prompt (@Chat):** "Show how to implement a simple first-login checklist or tooltip guide in React using Shadcn components, triggered based on a user's onboarding status from the auth store."

Step 1.6: Commit Progress

- **Manual Action:**
    - `git add .`
    - `git commit -m "Feat: Phase 1 - Authentication, Email Verification, Onboarding Flows"`
    - `git push`

Phase 2: User Profiles & Space Association

Goal: Allow users to create detailed profiles and associate them with spaces/companies.

Step 2.1: Detailed User Profiles ([UP-01])

- **Manual Action (Backend):**
  - Extend `models/user.py`.

> **Cursor Prompt (@Edit `models/user.py`):** "Add the following nullable fields to the User model: `profile_picture_url` (String), `short_bio` (Text), `contact_info_visibility` (String, e.g., 'Private', 'Connections', 'Public'), `linkedin_profile_url` (String), `skills_tags` (JSON or Text[]), `industry_focus_tags` (JSON or Text[]), `project_interests_goals` (Text), `collaboration_preferences_tags` (JSON or Text[]), `tools_technologies_tags` (JSON or Text[]). Adjust types based on DB (JSONB for Postgres is good for tags)." *(Alternatively, create separate related tables for tags/skills if structured querying is highly needed).*

  - **Manual Action:** Run `alembic revision --autogenerate -m "Add detailed user profile fields"` and `alembic upgrade head`.
  - Modify `schemas/user.py`.

> **Cursor Prompt (@Edit `schemas/user.py`):** "Update `User` schema (response) and create `ProfileUpdate` schema (input) reflecting the new fields. Ensure sensitive fields like `hashed_password` are excluded from responses."

  - Create `routers/profiles.py`.

> **Cursor Prompt (@Edit `routers/profiles.py`):** "Generate FastAPI router endpoints: * `GET /users/me/profile`: Requires authentication, fetches the logged-in user's profile data from DB. * `PUT /users/me/profile`: Requires authentication, accepts `ProfileUpdate` schema, updates the user's profile in DB, returns updated profile."

  - Modify `main.py` to include the profiles router.
- **Manual Action (Frontend):**
  - Create `app/dashboard/profile/page.tsx`.

> **Cursor Prompt (@Edit `app/dashboard/profile/page.tsx`):** "Generate a profile view/edit page. * Fetch user profile data using an API call to `/users/me/profile` (create function in `lib/api/profile.ts`). * Display profile data using Shadcn components (`Input`, `Textarea`, `Avatar` for picture). * Use `react-hook-form` to manage editable fields. * Implement tag input component (or use a library like `react-tag-input`) for skills, industry etc. * Handle form submission to call a `updateProfile` function (in `lib/api/profile.ts`) making a PUT request to `/users/me/profile`."

  - **(Profile Picture Upload):** This requires more setup.
    - **Backend:** Setup cloud storage (e.g., GCS bucket). Add endpoint (e.g., `POST /users/me/profile/picture`) to handle `UploadFile`, upload to GCS, update `profile_picture_url` on User model. Install `google-cloud-storage`.

> **Cursor Prompt (@Chat):** "Show a FastAPI example endpoint to upload a file, save it to Google Cloud Storage, and return the public URL."

- **Frontend:** Add file input component. Handle file selection and POST to the upload endpoint. Update UI optimistically or after successful upload.

Step 2.2: Space Association ([UP-02]) & Basic Company/Startup Profiles ([PROF-01])

- **Manual Action (Backend):**
  - Create `models/space.py`.

> **Cursor Prompt (@Edit `models/space.py`):** "Generate SQLAlchemy model `Space` with fields: `id`, `name` (String, unique), `location` (String, nullable), `description` (Text, nullable)."

  - Create `models/company.py`.

> **Cursor Prompt (@Edit `models/company.py`):** "Generate SQLAlchemy model `Company` with fields: `id`, `name` (String, unique), `logo_url` (String, nullable), `industry_focus` (String, nullable), `description` (Text, nullable), `website_link` (String, nullable)."

  - Modify `models/user.py`.

> **Cursor Prompt (@Edit `models/user.py`):** "Add nullable ForeignKey fields `space_id` referencing `spaces.id` and `company_id` referencing `companies.id` to the User model. Define the SQLAlchemy relationships (`relationship(...)`) backref if needed."

  - **Manual Action:** Run `alembic revision --autogenerate -m "Add Space, Company models and user associations"` and `alembic upgrade head`.
  - Create `routers/admin.py` (or similar for SYS Admin tasks).

> **Cursor Prompt (@Edit `routers/admin.py`):** "Generate basic FastAPI endpoints (protected for SYS Admin role) to: * `POST /spaces`: Create a new space. * `POST /companies`: Create a new company. * `PUT /users/{user_id}/assign`: Assign a `space_id` and `company_id` to a user."

  - Create `routers/companies.py`.

> **Cursor Prompt (@Edit `routers/companies.py`):** "Generate endpoint `GET /companies/{company_id}`. Fetches Company details and potentially a list of *active* associated users (respecting privacy - maybe just names/roles, requires join query)."

  - Modify `main.py` to include new routers.
- **Manual Action (Frontend):**
  - Modify profile page/dashboard to display associated Space/Company name (if available on fetched user data).
  - Create `app/company/[id]/page.tsx`.

> **Cursor Prompt (@Edit `app/company/[id]/page.tsx`):** "Generate a view-only Company/Startup profile page. * Fetch data using the company ID from the URL via an API call to `/companies/{id}` (create function in `lib/api/company.ts`). * Display Company Name, Logo, Industry, Description, Website, and list of associated members."

Step 2.3: Commit Progress

- **Manual Action:**

- `git add .`
- `git commit -m "Feat: Phase 2 - User Profiles, Space/Company Models & Association"`
- `git push`

Phase 3: Core Matching Engine (High-Quality MVP Focus)

Goal: Implement the intra-space matching logic using vector embeddings and `pgvector`.

Step 3.1: Vector Embeddings Setup ([MA-01] - Backend)

- **Manual Action (Database):** Connect to your PostgreSQL instance and run: `CREATE EXTENSION IF NOT EXISTS vector;`
- **Manual Action (Backend):**
  - Install required libraries: `pip install pgvector google-generativeai` (or `sentence-transformers`). Add to `requirements.txt`. Update `.env` and `config.py` with Google API Key if using their model.
  - Install `sqlalchemy-pgvector`.

> **Cursor Prompt (@Edit `models/user.py`):** "Import `Vector` from `sqlalchemy_pgvector` and add a `profile_vector` column of type `Vector(embedding_dim)` to the User model (replace `embedding_dim` with the dimension of your chosen model, e.g., 768 for `text-embedding-004`). Make it nullable."

  - **Manual Action:** Run `alembic revision --autogenerate -m "Add profile_vector column to user"` and `alembic upgrade head`.
  - Create `services/embedding_service.py`.

> **Cursor Prompt (@Edit `services/embedding_service.py`):** "Generate a service class `EmbeddingService`. Include: * Initialization (`__init__`) setting up the embedding client (Google GenAI or SentenceTransformer). * A method `generate_embedding(text: str)` that calls the embedding model API/library and returns the vector. * A method `update_user_embedding(user_id: int, db: Session)` that: * Fetches the user by ID. * Concatenates relevant text fields (bio, skills, interests, goals). Handle potential None values. * Calls `generate_embedding` on the concatenated text. * Updates the `profile_vector` field for the user in the DB."

  - Modify profile update endpoint (`routers/profiles.py`, `PUT /users/me/profile`). After successfully saving profile changes, call `embedding_service.update_user_embedding`. Inject `EmbeddingService` as a dependency.

Step 3.2: Intra-Space Similarity Search ([MA-01] - Backend)

- **Manual Action (Backend):**
  - Create `routers/matching.py`.

> **Cursor Prompt (@Edit `routers/matching.py`):** "Generate a FastAPI router endpoint `GET /matching/discover`. It should: * Require authentication. Get current user ID and their `space_id`. * Fetch the current user's `profile_vector`. If null, maybe return an error or

> empty list. * Perform a SQLAlchemy query on the User model: * Filter by `space_id` matching the current user's space. * Filter out the current user (`User.id != current_user_id`). * Filter out users from the same company (`User.company_id != current_user.company_id`, handle null company IDs). * Order by similarity using `pgvector` operator. Use `User.profile_vector.cosine_distance(current_user_vector)` or `l2_distance` / `max_inner_product`. * Limit the results (e.g., top 20). * **(Refinement - Placeholder):** Add comments indicating where structured data filtering/boosting (e.g., shared tags) would occur *after* retrieving the initial vector candidates. Keep it simple for MVP. * Return a list of matched users (include ID, name, role, profile picture, key skills/interests for the cards, and the calculated similarity score)."

- Modify `main.py` to include the matching router.

Step 3.3: Matching Interface & Explainable AI ([MA-02])

- **Manual Action (Frontend):**
  - Create `app/dashboard/discover/page.tsx`.

> **Cursor Prompt (@Edit `app/dashboard/discover/page.tsx`):** "Generate the 'Discover Connections' page. * Fetch potential matches using an API call to `/matching/discover` (create function in `lib/api/matching.ts`). * Display matches using Shadcn `Card` components in a grid or list. Each card should show photo (`Avatar`), name, role, key skills/interests, and the match score. * Include a placeholder area on each card for 'Match Reasons'."

  - **Explainable AI (Simple MVP Approach):**
    - **Backend Modification ([MA-01] refinement):** In `routers/matching.py`, after getting the top N vector matches, iterate through them. For each match, compare their structured `skills_tags`, `industry_focus_tags`, etc., with the current user's profile. Add simple reason strings (e.g., "Shared skill: Python", "Industry: Tech") to the returned match data based on overlaps.

> **Cursor Prompt (@Edit `routers/matching.py` endpoint):** "After retrieving the top N vector matches, iterate through them. For each match, compare their `skills_tags` and `industry_focus_tags` with the current user's profile data. Add a list of simple string 'reasons' (e.g., 'Shared skill: X', 'Common industry: Y') to the data returned for each match based on detected overlaps."

    - **Frontend Implementation:** Display the returned `reasons` list in the designated placeholder on the match cards.

Step 3.4: Connection Requests & Management ([MA-03], [MA-04], [NT-01] - Part 1)

- **Manual Action (Backend):**
  - Create `models/connection.py`.

> **Cursor Prompt (@Edit `models/connection.py`):** "Generate SQLAlchemy model `Connection`. Fields: `id`, `requester_id` (ForeignKey User.id), `recipient_id` (ForeignKey User.id), `status` (String: 'pending', 'accepted', 'declined', default 'pending'), `created_at`. Define relationships `requester` and `recipient` back to User."

    - **Manual Action:** Run `alembic revision --autogenerate -m "Add connection model"` and `alembic upgrade head`.
    - Create `routers/connections.py`.

> **Cursor Prompt (@Edit `routers/connections.py`):** "Generate FastAPI endpoints: * `POST /connections/{recipient_id}`: Requires auth. Creates a Connection record with status 'pending'. Prevent duplicate pending requests. (Placeholder: Add notification trigger). * `PUT /connections/{connection_id}`: Requires auth. Accepts `status` ('accepted' or 'declined') in payload. Updates the connection status. Ensure only recipient can accept/decline. (Placeholder: Add notification trigger for acceptance). * `GET /connections`: Requires auth. Fetches user's incoming pending requests and existing accepted connections."

    - Modify `main.py` to include the connections router.
- **Manual Action (Frontend):**
    - Modify match cards (`app/dashboard/discover/page.tsx`) to include a "Connect" button. On click, call a `sendConnectionRequest(recipientId)` function (in `lib/api/connections.ts`) hitting the `POST /connections/{recipient_id}` endpoint. Show feedback (e.g., toast).
    - Create `app/dashboard/connections/page.tsx` (or integrate into Notification Center later).

> **Cursor Prompt (@Edit `app/dashboard/connections/page.tsx`):** "Generate a page to manage connections. * Fetch connection data using an API call to `/GET /connections` (create function in `lib/api/connections.ts`). * Display incoming pending requests with 'Accept'/'Decline' buttons. Buttons call `updateConnectionStatus(connectionId, status)` function. * Display a list of accepted connections (e.g., names/profiles)."

Step 3.5: Commit Progress

- **Manual Action:**
    - `git add .`
    - `git commit -m "Feat: Phase 3 - Matching Engine (pgvector), Interface, Connections"`
    - `git push`

Phase 4: Communication & Notifications

Goal: Implement real-time chat between connected users and an in-app notification system.

Step 4.1: Real-time Setup (Socket.IO) ([CH-01], [NT-01])

- **Manual Action (Backend):**
    - `pip install python-socketio[asyncio]` (Add to `requirements.txt`).
    - Modify `main.py`.

> **Cursor Prompt (@Edit `main.py`):** "Integrate Socket.IO with FastAPI. Create a `socketio.AsyncServer` instance (`sio`). Mount it to the FastAPI app using `socketio.ASGIApp`. Add basic `connect` and `disconnect` event handlers that print the socket ID (`sid`)."

- **(Authentication):** Implement socket authentication.

> **Cursor Prompt (@Chat):** "Show how to authenticate Socket.IO connections in FastAPI, likely by verifying a JWT token passed during connection or via an initial auth event." *(Implement the chosen method, potentially storing authenticated user ID in the socket session: `sio.save_session(sid, {'user_id': user_id})`).*

- **(Room Management):**

> **Cursor Prompt (@Edit `main.py` socket handlers):** "In the `connect` event handler, after successful authentication, make the socket join a room based on their `user_id` (e.g., `sio.enter_room(sid, f'user_{user_id}')`)."

- **Manual Action (Frontend):**
  - `npm install socket.io-client`
  - Create `hooks/useSocket.ts`.

> **Cursor Prompt (@Edit `hooks/useSocket.ts`):** "Generate a React hook `useSocket` that: * Initializes a Socket.IO client connection (use API base URL). * Handles connection (`connect`, `disconnect`) events. * Includes logic to send an authentication token upon connection (if needed based on backend auth). * Provides the `socket` instance and connection status. * Handles cleanup on unmount (`socket.disconnect()`)."

  - Use `useSocket` in a central layout or context provider to establish the connection when the user is logged in.

Step 4.2: Direct Messaging ([CH-01], [CH-02], [CH-03])

- **Manual Action (Backend):**
  - Create `models/message.py`.

> **Cursor Prompt (@Edit `models/message.py`):** "Generate SQLAlchemy model `Message`. Fields: `id`, `sender_id` (ForeignKey User.id), `recipient_id` (ForeignKey User.id), `connection_id` (ForeignKey Connection.id, optional), `content` (Text), `created_at` (DateTime), `read_at` (DateTime, nullable)." *(Add relationships).*

  - **Manual Action:** Run `alembic revision --autogenerate -m "Add message model"` and `alembic upgrade head`.
  - Create `routers/chat.py`.

> **Cursor Prompt (@Edit `routers/chat.py`):** "Generate endpoint `GET /chat/{connection_id}/history`. Requires auth. Fetches messages between the logged-in user and the other user in the specified accepted connection (verify user is part of the connection)."

  - Modify `main.py` or create `socket_handlers/chat.py`.

> **Cursor Prompt (@Chat in backend):** "Generate Socket.IO event handlers for chat: * `@sio.event async def send_message(sid, data):` Authenticates SID, gets sender ID. Extracts `recipient_id` and `content` from data. Verifies sender and recipient are connected. Saves message to DB. Emits `'receive_message'` event to the recipient's room

> (`f'user_{recipient_id}'`) with message details. * `@sio.event async def mark_as_read(sid, data):` Authenticates SID. Gets user ID. Extracts `message_ids` or `sender_id` from data. Updates `read_at` for relevant messages in DB where recipient is the current user."

  - **(File Attachments):** Add endpoint `POST /chat/attachments`. Handles `UploadFile`, saves to GCS (similar to profile pic), returns URL. Modify `Message` model to store `attachment_url`. Modify `send_message` event to handle attachment URLs.
  - **(Edit/Delete):** Add socket events `edit_message`, `delete_message`. Add logic to check ownership and timeframe. Update DB and emit updates to relevant users.
- **Manual Action (Frontend):**
  - Create `app/dashboard/chat/[connectionId]/page.tsx` or a reusable chat component.

> **Cursor Prompt (@Edit `app/dashboard/chat/.../page.tsx`):** "Generate a chat interface component: * Uses `useSocket` hook. * Fetches chat history from `/chat/{connectionId}/history`. * Displays messages (bubbles, timestamps). Use `Avatar` for sender pics. * Includes an input field and send button. On send, emits `send_message` socket event with recipient ID and content. * Listens for `receive_message` event and appends new messages to the display. * Implements basic read receipt logic (e.g., emit `mark_as_read` when chat window is focused or messages scrolled into view). * (Optional) Add file upload button triggering API call and sending message with attachment URL. * (Optional) Add emoji picker (`emoji-picker-react`). * (Optional) Add context menu/buttons for editing/deleting own messages, emitting corresponding socket events."

Step 4.3: In-App Notification Center ([NT-01])

- **Manual Action (Backend):**
  - Create `models/notification.py`.

> **Cursor Prompt (@Edit `models/notification.py`):** "Generate SQLAlchemy model `Notification`. Fields: `id`, `user_id` (ForeignKey User.id, index), `type` (String: e.g., 'connection_request', 'connection_accepted', 'new_message'), `related_entity_id` (Integer, nullable), `message` (String), `is_read` (Boolean, default False), `created_at`."

  - **Manual Action:** Run `alembic revision --autogenerate -m "Add notification model"` and `alembic upgrade head`.
  - Modify relevant backend actions (e.g., in `routers/connections.py` `POST /connections`, `PUT /connections`; in `socket_handlers/chat.py` `send_message`) to create `Notification` records for the relevant user.
  - Modify `main.py` or create `socket_handlers/notifications.py`.

> **Cursor Prompt (@Chat in backend):** "Generate Socket.IO logic: When a Notification record is created, emit a `'new_notification'` event to the target user's room (`f'user_{user_id}'`) containing the notification details (or just a count)."

  - Create `routers/notifications.py`.

> **Cursor Prompt (@Edit `routers/notifications.py`):** "Generate endpoints: * `GET /notifications`: Requires auth. Fetches user's notifications (unread first, then read, limit

> count). * `POST /notifications/mark-read`: Requires auth. Accepts list of notification IDs or 'all'. Marks them as read in DB."

- **Manual Action (Frontend):**
  - Add a Bell icon (e.g., `@radix-ui/react-icons`) to the main layout header.
  - Use `useSocket` to listen for `'new_notification'` event. Show a badge/indicator on the bell icon if unread notifications exist (fetch initial count via API).
  - Wrap the Bell icon in a Shadcn `Popover` or trigger a `Sheet`.

> **Cursor Prompt (@Edit Notification Popover/Sheet component):** "Generate a component that: * Fetches notifications from `GET /notifications`. * Displays notifications (message, time ago). * Links notifications to relevant pages (e.g., connection request -> connections page; new message -> chat page). * Includes a 'Mark all as read' button calling `POST /notifications/mark-read`."

  - Trigger the mark-read API call when the popover opens or via the button.

Step 4.4: Admin Contact Chat ([CH-04])

- **Manual Action (Backend):**
  - Add logic (e.g., in `routers/chat.py` or a dedicated endpoint) for Startup Admins/Freelancers to find their assigned Corporate Space Admin's `user_id`. This likely involves looking up the user's `space_id` and finding the User with 'CorporateAdmin' role assigned to that space.
- **Manual Action (Frontend):**
  - Add a "Contact Space Admin" button in the Startup/Freelancer dashboard ([DB-02] - Phase 5).
  - On click, find the Corporate Admin's user ID (via API call if needed).
  - Navigate the user to the chat interface (`/dashboard/chat/{admin_user_id}`), potentially needing to create a pseudo-connection ID or handle direct user-to-user chat lookup if not formally 'connected'.

Step 4.5: Commit Progress

- **Manual Action:**
  - `git add .`
  - `git commit -m "Feat: Phase 4 - Real-time Chat & Notifications"`
  - `git push`

Phase 5: Dashboards & Administration

Goal: Create personalized dashboards for users and comprehensive admin panels.

Step 5.1: Basic User Dashboard ([DB-01])

- **Manual Action (Frontend):**
  - Create `app/dashboard/page.tsx`.

> **Cursor Prompt (@Edit `app/dashboard/page.tsx`):** "Generate the main user dashboard component. * Fetch logged-in user's profile data (`/users/me/profile`). * Use Shadcn `Card` and layout components (Grid/Flex). * Display a Profile Summary section (Avatar, Name, Role, Bio snippet). Link to full profile page. * Add quick access buttons/links (using Shadcn `Button`)

> to: Discover Matches, Chats, Notifications, Profile Edit. * Conditionally display a link to the Company/Startup profile page (`/company/{id}`) if `company_id` exists on user data."

Step 5.2: Startup/Freelancer Admin Dashboard ([DB-02])

- **Manual Action (Backend):**
  - **(Startup Only) Team View:** Create endpoint `GET /startups/me/members`. Requires Startup Admin auth. Fetches users belonging to the same `company_id` and `space_id`.
  - **(Startup Only) Request Add Member:** Create endpoint `POST /startups/me/request-member`. Requires Startup Admin auth. Creates a notification or task for the Corporate Admin of the space, indicating the request (include startup name/details).
  - **Usage Stats:** Create endpoint `GET /users/me/stats`. Requires auth. Returns basic stats like connection count.
- **Manual Action (Frontend):**
  - Modify `app/dashboard/page.tsx` or create specific components conditionally rendered based on user role.

> **Cursor Prompt (@Edit Dashboard components):** "Conditionally render these sections based on user role ('StartupAdmin' or 'Freelancer'): * **(Startup Only):** 'Team Members' section. Fetches from `/startups/me/members` and displays a list. * **(Startup Only):** 'Add Team Member' button. Calls `/startups/me/request-member` API. * 'Contact Space Admin' button (implemented in Phase 4). * 'Subscription Details' section (placeholder - link to billing page created later). * 'Usage Stats' section. Fetches from `/users/me/stats` and displays simple stats."

Step 5.3: Corporate Admin Dashboard ([DB-03])

- **Manual Action (Backend):** Create `routers/corporate_admin.py`. Add role-based protection.

> **Cursor Prompt (@Edit `routers/corporate_admin.py`):** "Generate endpoints for Corporate Admin: * `GET /corporate-admin/space/members`: Fetches own employees, onboarded Startups, Freelancers within their assigned space(s). Include user profiles. * `GET /corporate-admin/space/workstations`: Fetches space details (total workstations, assigned count). * `PUT /corporate-admin/space/workstations`: Updates total available workstations. * `POST /corporate-admin/space/assign-workstation`: Assigns a user (Startup/Freelancer) to a workstation slot (might just decrement count or manage specific assignments). * `GET /corporate-admin/space/member-requests`: Fetches pending requests from Startups to add members. * `POST /corporate-admin/space/member-requests/{request_id}/respond`: Handles Approve/Reject for member requests (updates request status, potentially assigns workstation if approved)."

  - Integrate access to company subscription details and basic space utilization/connection stats endpoints.
- **Manual Action (Frontend):**
  - Create `app/dashboard/corporate-admin/` folder and pages (e.g., `members/page.tsx`, `workstations/page.tsx`, `requests/page.tsx`).

> **Cursor Prompt (@Edit `app/dashboard/corporate-admin/members/page.tsx`):** "Generate a page for member management. Use Shadcn `Table` to display employees, startups,

> freelancers fetched from `/corporate-admin/space/members`. Allow viewing profiles." *(Adapt for Workstation management, Request handling)*.

- Add navigation within the dashboard for these sections.
- Add entry point for Recruiting Agent Tool ([AG-01] - Phase 6).
- Ensure access to "Contact Space Admin" chats (likely viewing incoming chats initiated by Startups/Freelancers).

Step 5.4: SYS Admin Dashboard ([DB-04])

- **Manual Action (Backend):** Modify `routers/admin.py` or create `routers/sys_admin.py`. Add strict SYS Admin role protection.

> **Cursor Prompt (@Edit `routers/sys_admin.py`):** "Generate comprehensive SYS Admin endpoints for: * **User Management:** `GET /users` (list, filter, search, paginate), `GET /users/{id}`, `PUT /users/{id}` (change role/status), `POST /users/{id}/impersonate` (generate token for impersonation - use carefully!), `GET /users/reported` (link to moderation). * **Space Management:** `GET /spaces`, `POST /spaces`, `PUT /spaces/{id}`, `POST /spaces/{id}/assign-admin`. * **Stats:** `GET /stats/platform` (key metrics). * **Feedback:** `GET /feedback`, `PUT /feedback/{id}` (update status). * **Subscription/Promo:** `GET /subscriptions`, `POST /promo-codes`, `GET /promo-codes`. * **Moderation:** `GET /moderation/queue` (reported users/content), `POST /moderation/action` (warn, suspend, ban)."

- **Manual Action (Frontend):**
  - Create `app/sys-admin/` protected route/layout.
  - Build out the UI for each admin function. This is substantial. Use data tables extensively (`tanstack/react-table` recommended).

> **Cursor Prompt (@Chat):** "Show how to implement a data table with filtering, sorting, and pagination using `tanstack/react-table` v8 and Shadcn UI components in a Next.js Server Component or Client Component." *(Use this pattern for User Management, Spaces, Feedback, Moderation Queue etc.)*

  - Implement forms for creating/editing spaces, assigning admins, managing promo codes.
  - Display platform statistics.
  - Build the interface for the moderation queue with actions.

Step 5.5: Commit Progress

- **Manual Action:**
  - `git add .`
  - `git commit -m "Feat: Phase 5 - User & Admin Dashboards"`
  - `git push`

Phase 6: Agentic Features (Recruiting Agent)

Goal: Implement the AI-powered recruiting agent for Corporate Admins.

Step 6.1: Google ADK Setup & Agent Interface ([AG-01])

- **Manual Action (Backend):**
  - `pip install google-cloud-aiplatform` (Add to `requirements.txt`).
  - Set up Google Cloud Authentication (e.g., Service Account Key file, set `GOOGLE_APPLICATION_CREDENTIALS` env var).
  - Create `services/recruiting_agent_service.py`.
  - Modify `routers/corporate_admin.py`.

> **Cursor Prompt (@Edit `routers/corporate_admin.py`):** "Add endpoint `POST /corporate-admin/find-talent`. Requires Corp Admin auth. Accepts a schema with a text description field. It should initialize and call the `RecruitingAgentService` to process the request and return the suggested candidates."

- **Manual Action (Frontend):**
  - Add a new section "Find Talent for Your Space" to the Corporate Admin dashboard (`app/dashboard/corporate-admin/talent/page.tsx` or similar).

> **Cursor Prompt (@Edit Talent Finder Component):** "Generate a UI section with: * Display of Total vs Used Workstations (fetched data). * A Shadcn `Textarea` for the admin to describe the ideal profile. * A 'Find Potential Members' button that triggers an API call to `/corporate-admin/find-talent` with the description."

Step 6.2: Recruiting Agent Core Logic ([AG-02])

- **Manual Action (Backend):** This requires significant AI integration work.
  - Modify `services/recruiting_agent_service.py`.

> **Cursor Prompt (@Chat):** "Show how to structure a basic agent using the Google Vertex AI SDK (Agent Builder or Langchain with Vertex AI integration). Define tools for: * `internal_waitlist_search(query: str)`: Queries the User table (status='Waitlisted') using pgvector similarity search based on the query. Returns list of matching User profiles. * `external_web_search(query: str, location: str)`: Uses Google Search API (or similar `google-cloud-discoveryengine`) to find Startups/Freelancers matching query in the location (Munich). Returns list of URLs/snippets. * `(Attempt) find_contact_info(url: str)`: Tries to extract email from a webpage URL (mention risks/limitations)." *(Focus on `internal_waitlist_search` and `external_web_search` for reliable MVP)*. **Cursor Prompt (@Edit `services/recruiting_agent_service.py`):** "Implement the agent's main execution logic: Takes admin description, potentially space context (Munich), calls the defined tools, processes results, ranks/filters them, and returns a structured list of candidates (Name, Desc, Source [Web/Waitlist], Link [Profile/Website], Score)."

Step 6.3: Agent Results Display & Contact Facilitation ([AG-03], [AG-04])

- **Manual Action (Backend):**
  - **(Internal Contact):** Modify `/corporate-admin/find-talent` response for internal leads. Add logic to trigger an in-platform notification/request to the waitlisted user when admin chooses "Initiate Contact". This might involve creating a specific Notification type or a Connection request initiated by the admin.
  - **(External Contact):**

    - Generate unique secure token for external invites. Store temporarily or link to the lead.
    - Modify `/corporate-admin/find-talent`. Add logic: When admin chooses "Initiate Contact" for an external lead, trigger sending an automated email (via `utils/email.py`) from `invitations@shareyourspace.com`. Include intro, inviting company name (Pixida), opportunity, and the unique link (`https://yourdomain.com/invite/{token}`). **CRITICAL: Do NOT include admin's direct contact info.**
    - Create `routers/invites.py`.

    > **Cursor Prompt (@Edit `routers/invites.py`):** "Generate endpoints: * `GET /invite/{token}`: Verifies token. Fetches minimal relevant info (inviting company name, SYS value prop). Renders the dedicated landing page info. * `POST /invite/{token}/reply`: Verifies token. Takes reply message from form data. Creates a new message/chat thread in the initiating Corporate Admin's inbox, clearly marked as originating from the external invite."

- **Manual Action (Frontend):**
    - Modify the Talent Finder component (`app/dashboard/corporate-admin/talent/page.tsx`).

    > **Cursor Prompt (@Edit Talent Finder Component):** "Below the input form, display the list of candidates returned by the `/corporate-admin/find-talent` API. * Clearly distinguish between 'Internal Waitlist' and 'External Web' leads. * For each lead, show Name, Description, Source, Link (if available), Score. * Provide two buttons: 'Contact Manually' (no action) and 'Initiate Contact via ShareYourSpace'. * The 'Initiate Contact' button should call a specific backend endpoint (e.g., `POST /corporate-admin/initiate-contact/{lead_id}?type=internal|external`) which triggers the appropriate backend logic (internal notification or external email)."

    - Create the external invite landing page: `app/invite/[token]/page.tsx`.

    > **Cursor Prompt (@Edit `app/invite/[token]/page.tsx`):** "Generate the external invite landing page: * Fetches invitation details using the token via `GET /invite/{token}` API call. * Displays info about ShareYourSpace, the inviting company (e.g., Pixida), and the opportunity. * Includes a simple Shadcn `Textarea` and 'Submit Reply' `Button`. * On submit, POSTs the reply message to `POST /invite/{token}/reply`. Shows success message. * Includes a secondary CTA to sign up for ShareYourSpace."

Step 6.4: Commit Progress

- **Manual Action:**
    - `git add .`
    - `git commit -m "Feat: Phase 6 - Recruiting Agent (ADK/Vertex AI), Contact Flow"`
    - `git push`

Phase 7: Monetization & Growth

Goal: Implement subscription payments, promo codes, referrals, and feedback.

Step 7.1: Subscription & Payment Integration ([MO-01], [MO-02], [MO-03])

- **Manual Action (Backend):**
  - Define subscription tiers (Freelancer fixed, Startup per member) maybe in `config.py` or DB table. Corporate is manual MVP.
  - `pip install stripe` (Add to `requirements.txt`). Add Stripe keys to `.env`, `config.py`.
  - Create `routers/billing.py`.

> **Cursor Prompt (@Edit `routers/billing.py`):** "Generate Stripe integration endpoints: * `GET /billing/plans`: Returns defined subscription plans/prices. * `POST /billing/create-checkout-session`: Requires auth. Takes plan ID. Creates a Stripe Checkout session for subscription. Include `client_reference_id` (user ID). Define success/cancel URLs. * `POST /billing/webhook`: Public endpoint to receive Stripe webhooks. Verify webhook signature. Handle events like `checkout.session.completed`, `invoice.paid` (update User/Subscription status, store Stripe customer/subscription IDs, next billing date), `invoice.payment_failed`."

  - Implement Paywall: Create a FastAPI dependency (`core/dependencies.py`).

> **Cursor Prompt (@Edit `core/dependencies.py`):** "Generate a FastAPI dependency `require_active_subscription` that checks if the authenticated user has an active subscription status OR a valid pilot promo code flag. Raise HTTPException 402 if not."

  - Apply this dependency to core feature endpoints (matching, chat, agent tools etc.).
- **Manual Action (Frontend):**
  - Create `app/dashboard/billing/page.tsx`.

> **Cursor Prompt (@Edit `app/dashboard/billing/page.tsx`):** "Generate a Billing page: * Fetches user's current subscription status/plan (requires backend endpoint). * Fetches available plans from `/billing/plans`. * Displays current plan or prompts to subscribe. * Includes buttons to 'Subscribe'/'Change Plan' which call `/billing/create-checkout-session` and redirect the user to Stripe Checkout using `stripe-js`. * (Optional) Include button to Stripe Customer Portal for managing payment methods/cancellations (requires backend endpoint to create portal session)."

  - Install Stripe.js: `npm install @stripe/stripe-js @stripe/react-stripe-js`. Wrap app or billing page with `Elements` provider.
  - Display Paywall prompts/redirects for non-subscribed users accessing protected features.

Step 7.2: Promo Code & Guarantee ([MO-04], [MO-05])

- **Manual Action (Backend):**
  - Extend DB: Add `promo_code` field to User or Subscription table. Add `guarantee_end_date` to Subscription table.
  - Modify SYS Admin endpoints ([DB-04]) to manage promo codes (create, assign).
  - Modify subscription logic: Check for valid promo code during sign-up/checkout to bypass payment or apply discount. Set `guarantee_end_date` (e.g., 1 month from subscription start).

    - Implement tracking/notification for guarantee period end (e.g., scheduled job or check on admin login).
- **Manual Action (Frontend):**
    - Add promo code input field during sign-up or on billing page. Pass code to backend during checkout session creation.
    - Display guarantee terms clearly during payment flow.

Step 7.3: Referral Program ([GF-01], [GF-02])

- **Manual Action (Backend):**
    - Add `referral_code` (unique) and `referred_by_user_id` (nullable ForeignKey) to User model. Regenerate migrations.
    - Generate unique `referral_code` for users upon creation or on demand.
    - Modify registration: If referral code provided, store `referred_by_user_id`.
    - Implement logic to detect successful activation (e.g., referred user subscribes or becomes fully active). Trigger benefit granting.
    - Benefits Logic: Add `subscription_credit` (Decimal) and `badges` (JSON/Text[]) fields to User model. Update these fields when referral activation occurs. Apply credit logic to next Stripe invoice generation.
    - Create `routers/referrals.py`.

> **Cursor Prompt (@Edit `routers/referrals.py`):** "Generate endpoints: * `GET /referrals/me`: Requires auth. Returns user's `referral_code`, list of their referrals (status, activation date), and earned rewards (credits, badges)."

- **Manual Action (Frontend):**
    - Create `app/dashboard/referrals/page.tsx`.

> **Cursor Prompt (@Edit `app/dashboard/referrals/page.tsx`):** "Generate Referral page: * Fetches data from `/referrals/me`. * Displays user's unique referral code/link prominently. * Adds 'Copy' button and social share links (using simple `mailto:`, `twitter.com/intent/tweet` links). * Displays list/table of referred users and their status (Pending, Activated). * Displays earned rewards (Credits available: $X, Badges: Community Helper)."

    - Modify Profile page ([UP-01]) to display earned badges.

Step 7.4: Feedback Mechanism ([GF-03], [GF-04])

- **Manual Action (Backend):**
    - Create `models/feedback.py`.

> **Cursor Prompt (@Edit `models/feedback.py`):** "Generate SQLAlchemy model `Feedback`. Fields: `id`, `user_id` (ForeignKey User.id, nullable), `category` (String, nullable), `message` (Text), `status` (String: New, InProgress, Resolved, default 'New'), `created_at`." (Regen migrations).

    - Create `routers/feedback.py`.

> **Cursor Prompt (@Edit `routers/feedback.py`):** "Generate endpoint `POST /feedback`. Accepts schema with optional category and message. Links to logged-in user ID if available. Saves feedback to DB."

  - Integrate feedback viewing/management into SYS Admin dashboard ([DB-04]).
- **Manual Action (Frontend):**
  - Create a reusable `FeedbackButton` component.

> **Cursor Prompt (@Edit `components/feedback/FeedbackButton.tsx`):** "Generate a component using Shadcn `Button` (position fixed bottom-right?) that triggers a Shadcn `Dialog`. The dialog contains a form (`Textarea` for message, optional `Select` for category) to submit feedback via an API call to `POST /feedback`."

  - Include `<FeedbackButton />` in the main layout.

Step 7.5: Commit Progress

- **Manual Action:**
  - `git add .`
  - `git commit -m "Feat: Phase 7 - Monetization, Referrals, Feedback"`
  - `git push`

Phase 8: Security, Compliance & Final Polish

Goal: Implement security measures, compliance docs, and refine the UI/UX.

Step 8.1: Terms/Privacy & GDPR ([SEC-01])

- **Manual Action (Legal):** Draft comprehensive Terms of Service and GDPR-compliant Privacy Policy. Get legal review.
- **Manual Action (Frontend):**
  - Add mandatory checkbox to registration forms ([ON-01]) linking to ToS/Privacy pages. Disable submit button until checked.

> **Cursor Prompt (@Edit Registration Forms):** "Add a Shadcn `Checkbox` component with a label containing links to '/terms' and '/privacy' pages. Make form submission conditional on this checkbox being checked."

  - Create static pages: `app/terms/page.tsx`, `app/privacy/page.tsx`. Paste reviewed legal text here.
  - Add links to these pages in the application footer (in `MainLayout.tsx` or a dedicated `Footer` component).

Step 8.2: User Blocking & Reporting ([SEC-02])

- **Manual Action (Backend):**
  - Create `models/block.py`, `models/report.py`.

> **Cursor Prompt (@Edit `models/block.py`):** "Generate `Block` model: `id`, `blocker_id` (User FK), `blocked_id` (User FK), `created_at`. Add unique constraint on (blocker_id, blocked_id)."

> **Cursor Prompt (@Edit `models/report.py`):** "Generate `Report` model: `id`, `reporter_id` (User FK), `reported_user_id` (User FK, nullable), `reported_message_id` (Message FK, nullable), `reason` (Text), `status` (String: Open, Reviewed, ActionTaken, default 'Open'), `created_at`." (Regen migrations).

    - Modify endpoints fetching users/chats (e.g., matching, connections, chat history) to filter out blocked users (check `Block` table).
    - Create `routers/safety.py`.

> **Cursor Prompt (@Edit `routers/safety.py`):** "Generate endpoints: * `POST /safety/block/{user_id}`: Requires auth. Creates Block record. * `DELETE /safety/block/{user_id}`: Requires auth. Deletes Block record. * `POST /safety/report`: Requires auth. Accepts schema with `reported_user_id` or `reported_message_id`, `reason`. Creates Report record."

    - Feed reports into the SYS Admin Moderation Queue ([DB-04]).
- **Manual Action (Frontend):**
    - Add 'Block User' / 'Unblock User' and 'Report User' buttons/menu items on user profile views.
    - Add 'Report Message' option in chat message context menu.

> **Cursor Prompt (@Edit Profile/Chat components):** "Add buttons/menu items for Block/Report. On click, trigger a dialog confirming the action, then call the corresponding API endpoint from `/safety/*` (create functions in `lib/api/safety.ts`)."

Step 8.3: Final UI/UX Polish ([CP-02], [CP-03])

- **Manual Action (Frontend):**
    - Implement Dark Mode: `npm install next-themes`.

> **Cursor Prompt (@Chat):** "Show how to integrate `next-themes` with Shadcn UI in a Next.js App Router project, including setting up the provider and creating a theme toggle button." *(Implement the theme provider in `app/layout.tsx` and add a toggle button to the header/settings).*

    - Add Animations: Use `framer-motion`.

> **Cursor Prompt (@Edit Layout/Page components):** "Wrap main page content or specific elements with `motion.div` from `framer-motion` to add simple fade-in or slide-in animations on page load or component mount. Use `initial`, `animate`, `exit` props."

    - **Responsiveness Testing:** Thoroughly test the application on different screen sizes (Desktop, Tablet, Mobile) using browser developer tools. Adjust Tailwind classes (`sm:`, `md:`, `lg:`) as needed to ensure usability.
    - **Landing Page Wow Effect ([LP-01]):**
        - Install R3F: `npm install @react-three/fiber @react-three/drei three`.

> **Cursor Prompt (@Chat):** "Provide a basic example of integrating a simple `react-three-fiber` scene with a rotating cube or a loaded GLTF model into a Next.js component, triggered on scroll." *(Implement this in the hero section of the landing page. Start simple, potentially use a pre-built Drei helper or effect).*

    - Source or create animated SVGs/Lottie files (`npm install lottie-react`). Integrate them for icons or illustrations.

Step 8.4: Database Seeding ([CP-04])

- **Manual Action (Backend):**
    - Create `scripts/seed.py` (or use Alembic data migrations).

> **Cursor Prompt (@Chat):** "Generate a Python script structure (`scripts/seed.py`) that uses SQLAlchemy sessions to: * Create a SYS Admin user. * Create a Space ('Pixida Munich Hub'). * Create a Corporate Admin user (Pixida) assigned to the space. * Create a Company record for Pixida. * Create several sample Startup Admin users, associated Companies, assign them to the Pixida space, status 'Active'. * Create several sample Freelancer users, assign them to the Pixida space, status 'Active'. * Create some sample Waitlisted Startup/Freelancer users. * Generate profile details (bio, skills, interests using realistic sample data) and embeddings for active users. * Create sample connections between some active users. * Create sample chat messages between connected users."

    - Run the seed script: `python scripts/seed.py`.

Step 8.5: Comprehensive Testing

- **Manual Action (Backend & Frontend):**
    - **Unit Tests:** Write tests for critical utility functions, security logic, complex service methods.

> **Cursor Prompt (@Chat):** "Generate pytest unit tests for the `verify_password` and `get_password_hash` functions in `core/security.py`."

    - **Integration Tests:** Test API endpoints using `pytest` with `httpx` or FastAPI's `TestClient`.

> **Cursor Prompt (@Chat):** "Generate a pytest integration test for the `POST /auth/login` endpoint using FastAPI's TestClient, covering success and failure cases."

    - **End-to-End Testing:** Manually test all user flows described in Section 2 of `mvp-plan-v3.md`. Test as different user roles (SYS Admin, Corp Admin, Startup Admin, Freelancer, Waitlisted). Cover registration, login, profile completion, matching, connecting, chatting, admin actions, agent usage, payment flow (with Stripe test mode), referrals, feedback, blocking/reporting. Test edge cases.

Step 8.6: Commit Progress

- **Manual Action:**
    - `git add .`
    - `git commit -m "Feat: Phase 8 - Security, Compliance, Polish, Seeding, Testing"`
    - `git push`

Phase 9: Deployment & Launch

Goal: Deploy the application to production environments and prepare for pilot launch.

Step 9.1: Infrastructure & Deployment ([Tech Stack Section])

- **Manual Action (Infra):**
  - Set up production PostgreSQL instance (e.g., Google Cloud SQL, Supabase, Neon). Configure users, network access. Update production `.env` files.
  - Set up production Redis (optional, if needed for scaling Socket.IO or caching).
  - Set up Google Artifact Registry (or Docker Hub) for backend container images.
- **Manual Action (Backend):**
  - Create `Dockerfile` for the FastAPI application.

> **Cursor Prompt (@Chat in backend root):** "Generate a multi-stage Dockerfile for a Python FastAPI application using Uvicorn. Include stages for dependency installation and copying application code. Ensure it exposes the correct port."

  - Configure Google Cloud Run service (set environment variables, secrets, database connections, CPU/memory, scaling).
  - Configure Google Vertex AI Agent Engine or deploy agent logic within Cloud Run service. Ensure correct API access and authentication.
- **Manual Action (Frontend):**
  - Create a Vercel project. Link it to your frontend repository's main branch.
  - Configure Vercel environment variables (e.g., `NEXT_PUBLIC_API_BASE_URL` pointing to Cloud Run URL, Stripe public key).
- **Manual Action (CI/CD):**
  - Create `.github/workflows/` directory.

> **Cursor Prompt (@Chat):** "Generate a GitHub Actions workflow YAML file for deploying a Next.js application to Vercel on push to the main branch." **Cursor Prompt (@Chat):** "Generate a GitHub Actions workflow YAML file for building a Docker image for a FastAPI application, pushing it to Google Artifact Registry, and deploying it to Google Cloud Run on push to the main branch." *(Adjust based on chosen registry/hosting)*.

Step 9.2: Final Checks & Pilot Launch

- **Manual Action:** Run database migrations (`alembic upgrade head`) against the production database.
- **Manual Action:** Run seed script against production *only if needed for initial pilot data* (e.g., creating Pixida space/admin). Usually, production starts empty.
- **Manual Action:** Perform final smoke tests on the production environment.
- **Manual Action:** Onboard Pixida pilot users: Create accounts or guide them through signup, manually assign space/roles via SYS Admin panel if needed, provide pilot promo codes ([MO-04]).
- **Manual Action:** Set up monitoring/logging/error tracking (Google Cloud Monitoring/Logging, Vercel Analytics, Sentry).

Step 9.3: Post-Launch

- **Manual Action:** Actively monitor system health and user activity.
- **Manual Action:** Collect pilot user feedback via the feedback tool ([GF-03]) and direct communication.
- **Manual Action:** Plan for iterative improvements based on feedback and MVP validation results.

Conclusion: Following these steps meticulously, leveraging Cursor's AI capabilities for code generation, refactoring, and explanation, should result in a fully functional ShareYourSpace 2.0 MVP as defined in the project plan. Remember to test thoroughly at each stage and commit your progress frequently. Good luck!