I'll draft a full, from-scratch technical spec and architecture plan incorporating the redesigned direction: RSC-first, Supabase PostgREST with RLS, Drizzle for migrations only, AI provider abstraction with streaming, tool-calling, pgvector memory, strong validation, and observability.

1) Product overview

- **Purpose**: AI-powered spiritual companion with chat (Beatrice), correspondences search/exploration, journals, rituals, daily check-ins, and user spiritual profiles.

- **Primary flows**

  - Chat with streaming responses, contextualized by profile, memories, recent activity, and cosmic context.

  - Search and browse correspondences; reference them in chat via tool-calls.

  - Journal entries, rituals planner/log, daily check-ins feeding memory and stats.

  - Personalized dashboard with moon phase and recent activity.

- **Non-goals (v1)**: Multi-tenant orgs, complex push notifications, offline-first.

2) Goals and non-functional requirements

- **Goals**

  - RSC-first for performance; minimal client hydration.

  - Secure-by-default with RLS and server-only sensitive ops.

  - AI streaming with tool-calling and cost controls.

  - Strong typing and validation at boundaries.

  - Observability and rate limiting.

- **NFRs**

  - p95 API latency: < 500ms for CRUD, < 2.5s first token for chat, sustained 20+ tokens/s.

  - Uptime: 99.9% for API.

  - Privacy: No PII stored in logs; secrets never in client.

  - Performance budgets: Initial load < 120KB JS, Lighthouse perf > 90.

3) Tech stack

- **Frontend**: Next.js 14/15 App Router, React 18, TypeScript, Tailwind CSS, shadcn/ui.

- **Backend**: Supabase (Postgres + Auth + Storage + Realtime + Edge Functions). pgvector enabled.

- **Data access**: Supabase PostgREST via `@supabase/supabase-js` (runtime); Drizzle Kit for migrations only.

- **AI**: Provider abstraction for Anthropic Claude (primary) and OpenAI (optional), streaming via SSE.

- **Workers/Jobs**: Supabase Edge Functions + CRON; Inngest/Trigger.dev optional for advanced orchestration.

- **Validation**: Zod schemas; typed env loader.

- **Observability**: OpenTelemetry traces, Axiom/Logflare or Vercel logging, PostHog product analytics.

- **Rate limiting**: Upstash Ratelimit (Redis) or Postgres-based limiter.

4) High-level architecture

- **App layer (Next App Router)**:

  - Server Components for data fetching/render.

  - Client islands for interactive pieces (chat input, editors).

- **Domain modules** (vertical slices): `chat`, `memory`, `correspondences`, `profile`, `journal`, `rituals`, `analytics`, `auth`.

- **AI subsystem**:

  - Provider adapters with unified interface and streaming.

  - Tool registry callable by the model (correspondences lookup, profile summary, ritual suggest).

  - Prompt templates versioned and A/B testable.

- **Background processing**:

  - Embedding pipeline on message creation, nightly summarization jobs, search analytics rollups.

5) Directory structure

```text
src/
  app/
    (app)/
```

```
  dashboard/
  chat/
  correspondences/
  journal/
  rituals/
  layout.tsx
 (auth)/
  login/
  register/
  layout.tsx
 api/
  chat/
   stream/route.ts     # SSE streaming
   history/[id]/route.ts
  correspondences/route.ts
  profile/route.ts
  health/route.ts
modules/
 chat/
  actions/            # server actions for mutations
   sendMessage.ts
  db/
   schemas.ts         # Zod I/O schemas (not DB)
   queries.ts         # Supabase calls
  services/
   beatrice.ts
   prompt.ts
  ui/
   ChatInterface.tsx
```

```
    ChatInput.tsx
  types.ts
 memory/
  actions/
    enqueueEmbedding.ts
  services/
    embeddings.ts       # provider for embeddings
    retrieval.ts        # hybrid search
    summarization.ts
  workers/
    onMessageCreated.ts  # Supabase Edge Function
  db/
    queries.ts
  types.ts
 correspondences/
  db/queries.ts
  services/search.ts
  ui/
    SearchBar.tsx
    Results.tsx
  types.ts
 profile/
  actions/updateProfile.ts
  db/queries.ts
  types.ts
 journal/
  actions/createEntry.ts
  db/queries.ts
  types.ts
```

```
 rituals/
  actions/createRitual.ts
  db/queries.ts
  types.ts
 analytics/
  actions/track.ts
  db/queries.ts
  types.ts
lib/
 ai/
  types.ts            # LLM types
  providers/
   anthropic.ts
   openai.ts
  stream.ts           # SSE helpers
  tools.ts            # tool registry/types
 auth/
  server.ts           # Supabase server clients
  client.ts           # Supabase client helper
 validation/
  api.ts              # common API envelope
  zod-helpers.ts
 env.ts               # typed env loader (Zod)
 observability/
  logger.ts
  tracing.ts
 utils/
  date.ts
  constants.ts
```

```
 db/
  migrations/          # Drizzle kit outputs
  seed/
   seed.sql
 styles/
 public/
 tests/
  unit/
  integration/
  e2e/
```

6) Data model (Postgres, snake_case)
- Reuse Supabase `auth.users` for users.

- `conversations`
  - `id uuid pk`
  - `user_id uuid fk auth.users not null`
  - `title text`
  - `last_message_at timestamptz`
  - RLS: owner-only

- `messages`
  - `id uuid pk`
  - `conversation_id uuid fk conversations not null`
  - `user_id uuid fk auth.users not null`
  - `role text check in ('user','assistant','system','tool') not null`
  - `content text not null`
  - `metadata jsonb default '{}'`

  - `created_at timestamptz default now()`
  - RLS: owner-only

- `memory_embeddings` (pgvector)
  - `id uuid pk`
  - `user_id uuid fk auth.users not null`
  - `content text not null`
  - `embedding vector(1536)` // choose dim based on provider
  - `importance_score real default 0`
  - `source_type text` // 'chat','journal','manual'
  - `source_id uuid`
  - `created_at timestamptz default now()`
  - RLS: owner-only

- `journal_entries`
  - `id uuid pk`
  - `user_id uuid fk auth.users not null`
  - `title text`
  - `body text`
  - `tags text[] default '{}'`
  - `created_at timestamptz default now()`
  - RLS: owner-only

- `rituals`
  - `id uuid pk`
  - `user_id uuid fk auth.users not null`
  - `name text not null`
  - `description text`
  - `scheduled_for timestamptz`

  - `completed_at timestamptz`
  - RLS: owner-only

- `daily_check_ins`
  - `id uuid pk`
  - `user_id uuid fk auth.users not null`
  - `mood text`
  - `notes text`
  - `created_at timestamptz default now()`
  - RLS: owner-only

- `user_spiritual_profiles`
  - `user_id uuid pk fk auth.users`
  - `display_name text`
  - `current_journey_phase text`
  - `preferred_deities text[] default '{}'`
  - `spiritual_practices text[] default '{}'`
  - `personality_tags text[] default '{}'`
  - `updated_at timestamptz default now()`
  - RLS: owner-only

- `correspondences`
  - `id uuid pk`
  - `name text`
  - `family text` // herb, crystal, color, deity, element...
  - `attributes jsonb` // correspondences, uses, associations
  - `search_vector tsvector` // for keyword search
  - RLS: read-all

- `search_analytics`
  - `id uuid pk`
  - `user_id uuid fk auth.users`
  - `query text`
  - `results_count int`
  - `created_at timestamptz default now()`
  - RLS: owner-only read, insert-allowed

- RLS policies (examples)
  - All user-owned tables enforce `user_id = auth.uid()` for select/insert/update/delete.
  - Public lookup tables (e.g., `correspondences`) enable `select` for `true`.

7) Runtime data access and validation
- All runtime DB operations use Supabase JS with user session context.
- Drizzle only for migrations & schema evolution; never used in request lifecycle.
- Each module exposes `db/queries.ts` with Zod-validated I/O.
- Common API envelope:
```ts
// lib/validation/api.ts
export type ApiResult<T> = { ok: true; data: T } | { ok: false; error: { code: string; message: string } };
```

8) AI subsystem
- **Provider interface**
```ts
// lib/ai/types.ts
export type ChatMessage = { role: 'system'|'user'|'assistant'|'tool'; content: string; name?: string };
export interface LLMProvider {
  streamChat(opts: {
```

```
    model: string;
    messages: ChatMessage[];
    tools?: ToolDefinition[];
    temperature?: number;
    maxTokens?: number;
  }): AsyncIterable<{ type: 'text'|'tool_call'|'tool_result'|'error'; data: any }>;
}
```

- **Anthropic adapter**: Implements `LLMProvider`, maps tool-calling to Anthropic's API, streams tokens.
- **Tools registry** (`lib/ai/tools.ts`):
  - `lookupCorrespondences(query: string)` → top N items
  - `getProfileSummary(userId: string)` → short JSON summary
  - `suggestRituals(context: {...})` → list of suggestions
- **Prompting**:
  - System prompt built from templates in `config/prompts/*`, versioned.
  - Retrieval: hybrid of `pgvector` KNN and keyword fallback, top-k 5–10.
- **Safety & cost**:
  - Token budgeting per request; max tokens; retries with exponential backoff.
  - Per-user rate limiting on chat sends.

9) Streaming API and UI

- **Route**: `POST /api/chat/stream` (SSE; `text/event-stream`)
  - Input: `{ conversationId?: string, message: string }`
  - Behavior:
    - Ensure conversation (create if missing).
    - Insert user message.
    - Retrieve short history + retrieved memories + profile context.
    - Stream assistant tokens; interleave tool-calls and tool-results.

  - Upsert assistant message progressively or finalize at end.

- **Client**:

  - Chat renders partial tokens in real-time, shows “thinking” when tool-calling, allows cancel.

10) Server actions

- `sendMessage` (wraps DB insert + queue embed).

- `updateProfile`, `createJournalEntry`, `createRitual`, `logCheckIn`.

- All actions validate input with Zod and return `ApiResult`.

11) Background jobs and memory

- Trigger on message creation:

  - Edge Function `onMessageCreated`: fetch content, embed via chosen model, insert into `memory_embeddings` with `source_type='chat'`.

- Nightly summarization:

  - CRON invokes summarizer to compress long threads into distilled memories.

- Retrieval:

  - `services/retrieval.ts` combines vector KNN and `to_tsvector` search.

12) Security

- RLS on all user-owned tables.

- Secrets: provider keys server-only; never shipped to client.

- CORS: allow-list specific origins; only for routes that need it.

- Input validation with Zod; sanitize logs; structured error taxonomy.

13) Observability

- Request ID per request; structured logs with child loggers.

- Tracing spans: DB calls, AI provider calls, tool invocations.

- Metrics: tokens in/out, latency, error rates, rate-limit hits.

- Product analytics: page views, feature usage (PostHog).

14) Performance and caching

- RSC default; avoid client-side DB calls.
- Cache control: `revalidate` tags for read-mostly data (correspondences).
- Client state: React Query for cache; Zustand only for UI preferences.
- Images: Next Image; font optimizations; code-splitting for heavy UI.

15) Testing strategy

- Unit:
  - Zod schemas, provider adapter parsing, prompt builders.
- Integration:
  - Route handlers with mocked Supabase and provider.
  - Tool registry with DB.
- E2E:
  - Auth + chat streaming flows, profile update, correspondences search.
- AI record/replay:
  - Snapshot fixtures for provider JSON to make tests deterministic.

16) Deployment and environments

- Environments: `development`, `preview`, `production`.
- Vercel:
  - Node runtime for chat streaming route; Edge for static/public routes.
- Supabase:
  - Separate projects per env; automated migration via CI.
- Feature flags:
  - Config via env or Postgres table `feature_flags`; read on server.

17) Environment variables (typed via `lib/env.ts`)

- `NEXT_PUBLIC_APP_URL`

- `SUPABASE_URL`, `SUPABASE_ANON_KEY`
- `SUPABASE_SERVICE_ROLE_KEY` (workers only)
- `ANTHROPIC_API_KEY` (server only)
- `OPENAI_API_KEY` (optional)
- `AXIOM_TOKEN`/`SENTRY_DSN` (optional)
- `UPSTASH_REDIS_REST_URL`, `UPSTASH_REDIS_REST_TOKEN` (if using Upstash)

18) Initial database migrations (Drizzle Kit)

- Enable `pgvector` extension.
- Create tables listed in section 6 with indices:
  - `messages(conversation_id, created_at)`
  - `memory_embeddings USING ivfflat (embedding vector_cosine_ops)` with `lists` tuned
  - GIN index on `correspondences.search_vector`
- Seed minimal correspondences dataset.

19) API contracts (selected)

- `POST /api/chat/stream` (SSE)
  - Request: `{ conversationId?: string, message: string }`
  - Events: `token`, `tool_call`, `tool_result`, `done`, `error`
- `GET /api/chat/history/:id`
  - Response: `{ conversationId: string, messages: Message[] }`
- Server Action `updateProfile(input)`
  - Input: `{ displayName?: string, ... }`
  - Output: `ApiResult<Profile>`

20) Implementation milestones

- Milestone 1: Project scaffolding, env loader, CI, RLS policies, initial migrations, public dashboard page.
- Milestone 2: Profile + auth flows via server actions; RSC rendering.

- Milestone 3: Chat minimal: history fetch, message send, streaming with Anthropic adapter; no tools.

- Milestone 4: Memory embeddings pipeline + retrieval in prompt.

- Milestone 5: Tool-calling (correspondences lookup, profile summary).

- Milestone 6: Journals, rituals, daily check-ins; dashboard widgets.

- Milestone 7: Observability, rate limiting, test hardening, performance pass.

21) Risks and mitigations

- Provider instability → fallback to alternative model, exponential backoff, circuit breaker.

- Token cost overrun → budget per user/session, shorten context via summarization.

- RLS misconfig → add integration tests ensuring cross-user access denied.

- Streaming reliability → graceful degrade to non-streamed response if needed.

22) Acceptance criteria (v1)

- Authenticated users can chat with streaming and create/read history.

- Conversations use retrieved memories and profile context.

- Users can edit profile, create journals, rituals, and daily check-ins.

- Correspondences searchable and usable via tool-calls in chat.

- All user-owned data protected by RLS; secrets safe; CORS locked down.

- Logs and metrics available; tests green; Lighthouse performance > 90.

- Provided a comprehensive greenfield spec: architecture, modules, data model, APIs, AI subsystem with streaming and tools, background memory, RLS security, observability, testing, deployment, and milestones.

- Included a concrete directory structure, key interfaces, and contracts to guide implementation.