

ZERO OS — Platform Architecture Spec

What this is: The full technical architecture for ZERO OS as a platform where users launch and manage trading agents on Hyperliquid. **What this is NOT:** A website redesign. This is the backend, auth, admin, agent lifecycle, and data layer that powers everything.

Product Requirements

For Users

- Connect Hyperliquid wallet (wallet IS the identity)
- Choose agent configuration (strategy, risk level, coin universe)
- Launch agent on their HL account
- View live performance of their agent(s)
- Stop/pause/reconfigure agents
- View decision stream, positions, equity curve for their agents

For Admin (You)

- See all users, all agents, all performance
- Approve/reject waitlist entries
- Monitor system health across all running agents
- Kill/pause any agent
- View aggregate metrics (total AUM, total trades, total users)
- Manage waitlist, invite codes, access tiers

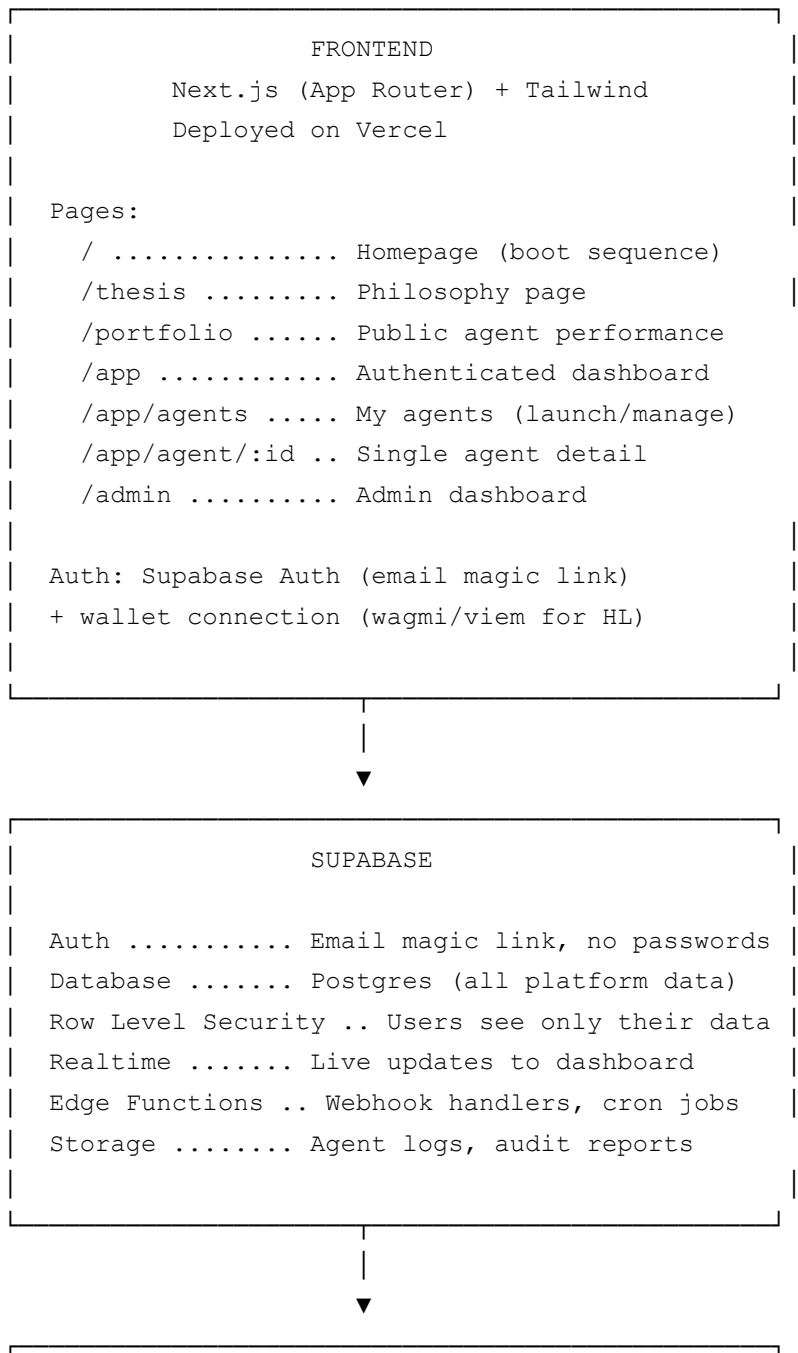
For the System

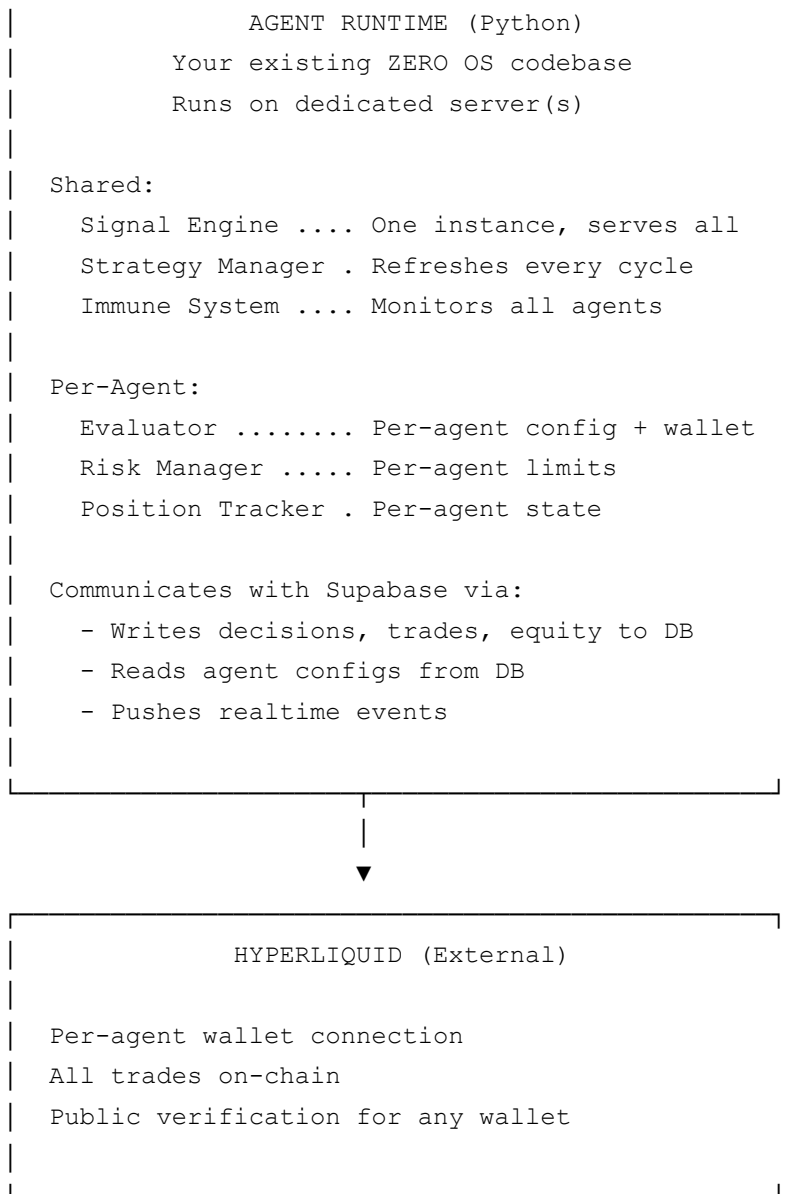
- Run multiple agents with different configs on different HL wallets
- Shared signal engine (one instance serves all agents)

- Per-agent risk isolation (one agent blowing up doesn't affect others)
 - Per-agent state management (positions, trades, equity)
 - Centralized immune system monitoring all agents
-

Tech Stack

Recommended: Supabase + Next.js + Python Agent Layer





Why This Stack

Choice	Reason
Supabase (not Firebase)	Postgres under the hood, real SQL, row-level security, self-hostable if needed later, generous free tier, realtime subscriptions built in
Next.js (not Astro)	Need authenticated routes (/app/*), API routes, middleware for auth checks. Astro is great for static sites but this is becoming an app
Vercel	Zero-config deploy for Next.js, edge functions, preview deployments, free tier sufficient for launch
Python agent runtime stays	Don't mix the trading engine with the web stack. The agent runtime runs on a dedicated server (your current setup). It talks to Supabase DB to read

separate	configs and write results
Postgres (via Supabase)	Relational data model fits perfectly — users have agents, agents have trades, trades have fills. SQL is the right tool

Alternative: If you hate Supabase

Stack	Pros	Cons
Supabase + Next.js	Fastest to ship, auth built in, realtime built in, generous free tier	Vendor dependency, less control
PocketBase + SvelteKit	Self-hosted, single binary, SQLite, very fast	Smaller ecosystem, less mature auth
Postgres + Drizzle + Next.js (DIY)	Full control, no vendor lock	More code to write, no built-in realtime
Convex	Realtime-first, great DX	Less mature, harder to self-host

Recommendation: Supabase. Ship fast, migrate later if needed.

Database Schema

```
-- Users
create table users (
  id uuid primary key default gen_random_uuid(),
  email text unique not null,
  hl_wallet text unique,          -- Hyperliquid wallet address
  status text default 'waitlist', -- waitlist | approved | active | suspended
  invite_code text unique,        -- their personal invite code
  invited_by uuid references users(id),
  tier text default 'free',        -- free | pro
  created_at timestamptz default now(),
  approved_at timestamptz,
  last_seen_at timestamptz
);

-- Agents (each user can have multiple)
create table agents (
  id uuid primary key default gen_random_uuid(),
```

```

user_id uuid references users(id) not null,
name text not null,           -- "balanced", "aggressive", custom name
status text default 'stopped', -- stopped | starting | running | paused | err
config jsonb not null,        -- strategy params (risk level, universe, limi
hl_wallet text not null,      -- which HL wallet this agent trades on
created_at timestamptz default now(),
started_at timestamptz,
stopped_at timestamptz,
equity_start numeric,         -- equity when agent started
equity_current numeric,       -- latest equity
total_trades integer default 0,
total_pnl numeric default 0
);

-- Agent configs (the jsonb above looks like)
-- {
--   "preset": "balanced",
--   "risk_level": "medium",
--   "max_positions": 3,
--   "universe": "auto",      -- auto = system picks, custom = user picks
--   "live_mode": true,       -- false = dry-run
-- }
-- NOTE: exact thresholds NOT stored in user-facing config
-- The system maps "risk_level: medium" to internal params
-- Users never see the actual numbers

-- Trades
create table trades (
  id uuid primary key default gen_random_uuid(),
  agent_id uuid references agents(id) not null,
  coin text not null,
  direction text not null,    -- long | short
  entry_price numeric,
  exit_price numeric,
  size_usd numeric,
  pnl numeric,
  pnl_pct numeric,
  fees numeric,
  funding numeric,
  entry_time timestamptz,
  exit_time timestamptz,
  exit_reason text,          -- stop | signal_reversal | max_hold | manual
  status text default 'open' -- open | closed
);

-- Decisions (the decision stream)
create table decisions (

```

```

    id uuid primary key default gen_random_uuid(),
    agent_id uuid references agents(id) not null,
    timestamp timestamptz default now(),
    coin text not null,
    direction text not null,
    decision text not null,          -- entered | rejected | blocked | held | close
    reason text,                    -- category only: "funding", "book_depth", "al
    sharpe numeric,
    metadata jsonb                  -- any extra data (non-secret)
);

-- Equity snapshots (for charting)
create table equity_snapshots (
    id uuid primary key default gen_random_uuid(),
    agent_id uuid references agents(id) not null,
    timestamp timestamptz default now(),
    equity numeric not null,
    realized_pnl numeric,
    unrealized_pnl numeric,
    positions_count integer
);

-- Waitlist
create table waitlist (
    id uuid primary key default gen_random_uuid(),
    email text unique not null,
    referrer_code text,             -- invite code that brought them
    created_at timestamptz default now(),
    status text default 'waiting',  -- waiting | invited | converted
    invited_at timestamptz,
    notes text                      -- admin notes
);

-- System health (for admin dashboard)
create table system_health (
    id uuid primary key default gen_random_uuid(),
    timestamp timestamptz default now(),
    component text not null,        -- signal_engine | immune | evaluator
    status text not null,           -- healthy | degraded | down
    details jsonb
);

-- Row Level Security
alter table agents enable row level security;
create policy "Users see own agents"
    on agents for select using (user_id = auth.uid());

```

```
alter table trades enable row level security;
create policy "Users see own trades"
  on trades for select using (
    agent_id in (select id from agents where user_id = auth.uid())
  );

alter table decisions enable row level security;
create policy "Users see own decisions"
  on decisions for select using (
    agent_id in (select id from agents where user_id = auth.uid())
  );
```

Auth Flow

Signup (Waitlist)

1. User enters email on getzero.dev
2. → Insert into waitlist table
3. User gets confirmation: "You're on the list. We'll email you."
4. Admin reviews and approves in admin dashboard
5. → User gets email: "You're in. Create your account."
6. → Supabase magic link → user created in users table
7. User connects HL wallet in /app
8. → hl_wallet stored in users table
9. User can now launch agents

Login (Returning User)

1. User enters email
2. Supabase sends magic link
3. Click → authenticated session
4. If wallet not connected → prompt to connect
5. If wallet connected → show dashboard with agents

Why Magic Link, Not Wallet-Only Auth

Wallet auth (Sign-In-With-Ethereum style) works but HL wallets are different from Ethereum wallets. Users may use HL's native wallet which doesn't support EIP-4361. Email magic link works universally. The wallet is CONNECTED after auth, not used FOR auth.

Agent Lifecycle

USER ACTION		SYSTEM RESPONSE
"Launch agent/balanced"	→	1. Validate user tier (free: 1 agent, pro: 3) 2. Validate wallet is connected 3. Create agent row in DB (status: starting) 4. Agent runtime picks up new agent config 5. Runtime validates HL wallet access 6. Runtime starts evaluator for this agent 7. Update agent status → running 8. Begin writing decisions + equity to DB
"Pause agent"	→	1. Update agent status → paused 2. Runtime stops evaluating (positions stay op 3. Immune system continues monitoring position 4. No new trades, existing positions managed
"Stop agent"	→	1. Update agent status → stopped 2. Runtime closes all positions (reduce_only) 3. Cancel all open orders 4. Final equity snapshot written 5. Agent summary generated
"Reconfigure agent"	→	1. Agent must be stopped first 2. Update config in DB 3. User relaunches with new config

Agent Presets (What Users See)

Users don't configure thresholds. They choose presets:

CONSERVATIVE

"Lower risk, fewer trades. Focuses on the highest-conviction signals."

→ Internally: strict filtering, low leverage, small universe

BALANCED

"Moderate risk, steady approach. The default configuration."

→ Internally: current production settings

AGGRESSIVE

"Higher risk, more trades. Wider universe, tighter stops."

→ Internally: relaxed filtering, wider universe, more positions

The mapping from preset → internal parameters is SERVER-SIDE ONLY. Users never see the

actual thresholds. This solves both the UX problem (users don't want to configure 15 parameters) and the information classification problem (thresholds are secret).

Admin Dashboard (/admin)

Overview Tab

ZERO OS ADMIN

Users: XX total · XX active · XX waitlist
Agents: XX running · XX paused · XX stopped
Trades: XXX today · XXX this week
Total AUM: \$XX,XXX across XX wallets
System: HEALTHY (all components green)

Users Tab

Email	Wallet	Status	Agents	Total P&L	Joined	Actions
a@...	0x1...	active	2	+\$45.20	Mar 21	[suspend] [view]
b@...	0x2...	active	1	-\$12.30	Mar 22	[suspend] [view]
c@...	—	approved	0	—	Mar 23	[remind] [view]

Waitlist Tab

Email	Referrer	Signed Up	Actions
x@...	—	Mar 21	[approve] [reject]
y@...	invite_abc	Mar 22	[approve] [reject]

[Approve all] [Export CSV]

Agent Monitor Tab

Agent	User	Status	Equity	P&L	Trades	Risk	Actions
balanced/0x1	a@..	running	\$742	+\$12	34	GREEN	[pause] [kill]
aggressive/0x1	a@..	running	\$305	-\$8	67	YELLOW	[pause] [kill]
balanced/0x2	b@..	paused	\$480	-\$12	22	GREEN	[resume] [kill]

System Health Tab

```
Signal Engine:    HEALTHY · last refresh 4m ago
Immune System:    HEALTHY · 0 alerts (24h)
Evaluator Pool:   3/3 agents served · avg cycle 12s
HL Connection:    HEALTHY · latency 45ms
Database:         HEALTHY · 2.3GB used
```

Admin Auth

Admin access is a simple Supabase role check. Add an `is_admin` boolean column to users table. Admin routes check this via middleware.

```
alter table users add column is_admin boolean default false;

-- Admin RLS policy
create policy "Admins see everything"
  on agents for select using (
    auth.uid() in (select id from users where is_admin = true)
  );
```

How Agent Runtime Talks to Supabase

The Python agent runtime (your existing codebase) connects to Supabase via the Python client:

```
from supabase import create_client

supabase = create_client(SUPABASE_URL, SUPABASE_SERVICE_KEY)

# Read agent configs (on startup and periodically)
agents = supabase.table('agents').select('*').eq('status', 'running').execute()

# Write a decision
supabase.table('decisions').insert({
    'agent_id': agent_id,
    'coin': 'FARTCOIN',
    'direction': 'long',
    'decision': 'rejected',
    'reason': 'book_depth',
    'sharpe': 2.19
}).execute()
```

```
# Write equity snapshot (every 60 seconds per agent)
supabase.table('equity_snapshots').insert({
  'agent_id': agent_id,
  'equity': 749.43,
  'realized_pnl': 2.62,
  'unrealized_pnl': 0.30,
  'positions_count': 1
}).execute()

# Update agent status
supabase.table('agents').update({
  'status': 'running',
  'equity_current': 749.43,
  'total_trades': 89
}).eq('id', agent_id).execute()
```

The service key bypasses RLS — the runtime is trusted. User-facing queries go through the frontend Supabase client which respects RLS.

HL Wallet Connection

Users need to grant ZERO OS permission to trade on their behalf. This requires an HL API wallet (agent wallet), NOT the user's main wallet.

Recommended Flow

1. User creates a sub-account / agent wallet on HL
2. User funds the agent wallet with USDC
3. User provides the agent wallet's private key to ZERO OS
4. ZERO OS stores the key encrypted in Supabase Vault
5. Agent runtime uses the key to execute trades

NOTE: The agent wallet should have LIMITED funds.
 Users keep their main wallet separate. The agent wallet is the "trading budget" — if something goes wrong, only the agent wallet is at risk.

Security

- Private keys encrypted at rest (Supabase Vault or separate secrets manager)
- Agent runtime accesses keys only in memory, never logs them

- Each agent trades only on its assigned wallet
- Users can revoke access by draining the agent wallet

Alternative: HL API Key (If Available)

If HL supports API keys with trade permissions (check their docs), that's cleaner than private keys. The user generates an API key on HL, gives it to ZERO OS, and the key has only trade permissions (no withdrawal).

Deployment Architecture

Vercel (Frontend)

- └─ Next.js app
 - └─ Public pages (/, /thesis, /portfolio)
 - └─ Auth pages (/app/*)

Supabase Cloud (Database + Auth)

- └─ Postgres
- └─ Auth
- └─ Realtime
- └─ Vault (encrypted secrets)

Dedicated Server (Agent Runtime)

- └─ Python processes
 - └─ Signal engine (shared, 1 instance)
 - └─ Agent evaluators (1 per running agent)
 - └─ Immune system (shared, monitors all agents)
 - └─ Supabase client (writes decisions, equity, trades)

Scaling Consideration

Currently: 1 server, 1 agent, your money. Target for beta: 1 server, up to 10 agents, 10 users' money.

At 10 agents, the signal engine is still shared (it evaluates the same universe for everyone). The per-agent evaluators are lightweight — they just apply the agent's config to the shared signals. The bottleneck is HL rate limits (1200 weight/min), not compute.

At 50+ agents: need to shard agent evaluators across multiple processes, implement agent-level rate limit budgeting, possibly multiple HL API connections. This is a good problem to have and a long way from today.

Implementation Order

WEEK 1: Foundation

- └─ Set up Supabase project (DB + Auth)
- └─ Create tables (users, agents, waitlist)
- └─ Next.js app with auth (magic link)
- └─ Waitlist page (public)
- └─ Admin dashboard (users + waitlist management)

WEEK 2: Agent Management

- └─ /app/agents page (list agents, launch/stop)
- └─ Agent preset selector (conservative/balanced/aggressive)
- └─ Wallet connection flow
- └─ Agent runtime: read configs from Supabase
- └─ Agent runtime: write decisions + equity to Supabase

WEEK 3: Live Dashboard

- └─ /app/agent/:id – per-agent performance view
- └─ Decision stream (from Supabase realtime)
- └─ Equity chart (from equity_snapshots)
- └─ Admin: agent monitor tab
- └─ Admin: system health tab

WEEK 4: Polish + Beta

- └─ Invite code system
- └─ Email notifications (agent started/stopped/alert)
- └─ Rate limiting and abuse prevention
- └─ Security audit on key storage
- └─ First 10 beta users from waitlist