Agentic Wallet Architecture

1. Overview

This project is a prototype agentic wallet built for AI agents on Solana devnet. It allows agents to autonomously create wallets, hold SOL and SPL tokens, sign and submit transactions, and interact with on-chain protocols — all without human intervention. The codebase is written in TypeScript and runs on Node.js.

The system is structured around a clear separation between **wallet operations** (key management, signing, submission) and **agent logic** (decision-making, strategy execution). A CLI exposes every capability so agent actions can be triggered, observed, and audited from the command line.

2. Architecture

```
┌──────────────────────────────────────────────────┐
│              Web Dashboard (Frontend)              │
│    Real-time agent & wallet management UI          │
└──────────────────────────────────────────────────┘
                          │
                          │
┌─────────────────────────▼────────────────────────┐
│            Express Server + REST API               │
│   Serves static files & JSON API on /api/*         │
└──────────────────────────────────────────────────┘
                          │
                          │
┌─────────────────────────▼────────────────────────┐
│                    CLI Layer                       │
│      Commands for wallets, tokens, agents          │
└──────────────────────────────────────────────────┘
                          │
                          │
┌─────────────────────────▼────────────────────────┐
│              Orchestrator + EventBus               │
│   • Spawns agents with isolated wallets            │
│   • Tracks spending and lifecycle                  │
│   • Emits events (spawned, started, stopped)       │
└──────────────────────────────────────────────────┘
            │                        │
            │                        │
┌───────────▼──────────┐  ┌──────────▼──────────────┐
│  SimpleAgentRunner   │  │  TradingAgent /          │
│  (SOL transfers)     │  │  LiquidityAgent          │
│                      │  │  (SPL token strategies)  │
└──────────────────────┘  └─────────────────────────┘
            │                       │
            │                       │
┌───────────▼───────────────────────▼──────────────┐
│              Wallet Service Layer                  │
│   • KeyManager (AES-256-GCM encryption)            │
│   • Transaction signing & submission               │
│   • ProgramAllowlist validation                    │
│   • Kora paymaster (optional gasless fees)         │
└──────────────────────────────────────────────────┘
                          │
                          │
┌─────────────────────────▼────────────────────────┐
│              Token Service Layer                   │
│   • SOL balances, airdrops, transfers              │
│   • SPL mint creation, minting, transfers          │
│   • Associated token account management            │
└──────────────────────────────────────────────────┘
                          │
                          │
┌─────────────────────────▼────────────────────────┐
│             Solana Devnet (JSON RPC)               │
└──────────────────────────────────────────────────┘
```

Source modules and their locations:

Module	Path	Responsibility
WalletService	`src/wallet/WalletService.ts`	Wallet creation, signing, transaction submission
KeyManager	`src/wallet/KeyManager.ts`	AES-256-GCM encryption/decryption of keypairs
TokenService	`src/tokens/TokenService.ts`	SOL + SPL token operations
Orchestrator	`src/orchestrator/Orchestrator.ts`	Agent lifecycle, spending tracking, multi-agent management
EventBus	`src/orchestrator/EventBus.ts`	Event emission for agent actions
SimpleAgentRunner	`src/agents/SimpleAgentRunner.ts`	Probabilistic SOL transfer loop
TradingAgent	`src/agents/TradingAgent.ts`	Price-based buy/sell strategy
LiquidityAgent	`src/agents/LiquidityAgent.ts`	Balance rebalancing strategy
ProgramAllowlist	`src/security/ProgramAllowlist.ts`	Transaction validation against allowed programs
KoraClient	`src/kora/KoraClient.ts`	Optional gasless transaction submission
CLI	`src/cli/index.ts`	Command-line interface for all operations
Server	`src/server/index.ts`	Express server, static file serving, SPA fallback
API Router	`src/server/api.ts`	REST API endpoints for agents, wallets, tokens
Dashboard	`public/index.html`	Single-page web dashboard (HTML/CSS/JS)

---

3. Wallet Design

3.1 Programmatic Wallet Creation

Wallets are created by calling `Keypair.generate()` from `@solana/web3.js`. The `WalletService.createWallet(name)` method generates a new keypair and immediately passes it to `KeyManager.saveKeypair()`, which encrypts the secret key and writes it to disk as a JSON file under the `keys/` directory. The public key is returned to the caller; the raw secret key is never exposed outside the encryption layer.

Each wallet is identified by a human-readable name (e.g., `agent-1`). The corresponding encrypted file is stored at `keys/<name>.json`. This naming convention makes it straightforward to manage multiple wallets.

3.2 Encrypted Key Storage

The `KeyManager` class handles all cryptographic operations. When a keypair is saved:

1. A random 16-byte **salt** is generated.
2. The encryption key is derived from the user's `KEYSTORE_PASSPHRASE` using **scrypt** (a memory-hard key derivation function) with the salt.
3. A random 12-byte **IV** (initialization vector) is generated.
4. The 64-byte secret key is encrypted using **AES-256-GCM**.
5. The **authentication tag** is extracted from the cipher.
6. The resulting JSON file stores: `cipherText`, `iv`, `salt`, `tag`, `publicKey`, `name`, and `createdAt` — all Base64-encoded where applicable.

To load a wallet, the process is reversed: the salt derives the same encryption key via scrypt, the IV and auth tag are used to initialize the AES-256-GCM decipher, and the secret key is decrypted back into a `Keypair` object. If the passphrase is wrong or the file has been tampered with, GCM authentication fails and an error is thrown.

The stored keypair JSON structure:

```
{
  "name": "agent-1",
  "publicKey": "H6ine...",
  "cipherText": "<base64>",
  "iv": "<base64>",
  "salt": "<base64>",
  "tag": "<base64>",
  "createdAt": "2026-02-19T07:21:33.408Z"
}
```

3.3 Automated Transaction Signing

The `WalletService.signTransaction(tx, wallet)` method handles signing without any manual intervention:

1. If the transaction has no `feePayer`, the wallet's public key is assigned.
2. If the transaction has no `recentBlockhash`, one is fetched from the RPC.
3. The transaction is signed using the wallet keypair.

This means the agent simply builds a transaction and hands it to the wallet service — signing, blockhash management, and fee payer assignment are all automatic.

3.4 Transaction Submission

`WalletService.sendTransaction(tx)` handles submission with built-in safety:

1. The transaction is validated against the **ProgramAllowlist** (if configured).
2. The transaction is **simulated** before sending (unless explicitly skipped) to catch errors early.
3. If Kora is configured, the transaction is routed through the Kora paymaster for gasless submission; otherwise, it is sent via `Connection.sendRawTransaction()`.
4. The transaction is confirmed at the `confirmed` commitment level before returning the signature.

3.5 Token Operations

The `TokenService` provides all SOL and SPL token functionality:

- **SOL balance**: `getSolBalance()` queries `connection.getBalance()` and converts lamports to SOL.
- **SOL airdrop**: `requestAirdrop()` requests devnet SOL. A separate `AIRDROP_RPC_URL` connection is supported for cases where the primary RPC blocks airdrops.
- **SOL transfer**: `transferSol()` builds a `SystemProgram.transfer` instruction, signs, and sends.
- **SPL balances**: `getSplBalances()` uses `getParsedTokenAccountsByOwner()` to list all token accounts with their mint, amount, and decimals.
- **SPL mint creation**: `createSplMint()` creates a new token mint on devnet with the wallet as mint authority.
- **SPL minting**: `mintSpl()` creates/finds the associated token account and mints tokens to it.
- **SPL transfer**: `transferSpl()` creates/finds both source and destination associated token accounts and transfers tokens between them.

---

4. Security Considerations

4.1 Key Protection

- **Encryption at rest**: Every keypair is encrypted with AES-256-GCM before being written to disk. Raw secret keys exist only in memory during the signing operation.
- **Key derivation**: The encryption key is derived using scrypt with a unique random salt per keypair, making brute-force attacks computationally expensive.
- **Authentication**: GCM mode provides authenticated encryption — any tampering with the ciphertext, IV, or tag causes decryption to fail, preventing silent corruption.
- **No plaintext logging**: The codebase never logs or prints secret keys. Only public keys appear in CLI output and logs.
- **Passphrase isolation**: The encryption passphrase is loaded from the `KEYSTORE_PASSPHRASE` environment variable, keeping it out of source code and configuration files checked into version control.

4.2 Per-Agent Spending Limits

Each agent record in `data/agents.json` tracks two fields: `solSpent` (cumulative SOL spent) and `spendingLimit` (maximum allowed, defaulting to the `DEFAULT_SPENDING_LIMIT` environment variable). The Orchestrator exposes `checkSpendingLimit(id, sol)` and `trackSpending(id, sol)` methods. Before an agent executes a costly transaction, the limit is checked; if the new total would exceed the cap, the operation is rejected with an error. This prevents a misconfigured or runaway agent from draining its wallet.

4.3 Program Allowlist

The `ProgramAllowlist` class (in `src/security/ProgramAllowlist.ts`) validates every transaction before it is submitted. It reads a list of allowed Solana program IDs from the `PROGRAM_ALLOWLIST` environment variable. When the allowlist is populated, `validateTransaction(tx)` iterates over every instruction in the transaction and throws an error if any instruction targets a program not in the list. When the allowlist is empty, all programs are permitted (the default for devnet flexibility). This mechanism prevents agents from interacting with unknown or malicious programs.

4.4 Transaction Simulation

Before submitting any transaction, `WalletService.sendTransaction()` calls `simulateTransaction()` against the RPC. If the simulation returns an error, the transaction is rejected before it reaches the network. This catches issues like insufficient funds, invalid instructions, or program errors before any SOL is spent on fees.

4.5 Per-Agent Wallet Isolation

Each agent gets its own unique keypair. There is no shared signer across agents. If one agent's key is compromised, other agents remain unaffected. The agent registry maps each agent ID to its dedicated wallet name and address.

4.6 Kora Gasless Transactions

The `KoraClient` provides optional integration with the Kora paymaster service. When `KORA_RPC_URL` is configured, `WalletService` routes signed transactions through Kora, which sponsors the transaction fees. If Kora is not configured, transactions use standard fee payment from the agent's wallet. The `isEnabled()` check ensures the system falls back gracefully without any code changes.

---

5. How the Project Interacts with AI Agents

5.1 Agent Spawning and Lifecycle

The `Orchestrator` is the central component that manages agent lifecycles. When `spawnAgent(strategy)` is called:

1. A unique wallet name is generated (e.g., `agent-1771485693255`).
2. A new keypair is created and encrypted to disk via `WalletService.createWallet()`.
3. An `AgentRecord` is created with a UUID, the wallet name, wallet address, strategy type, spending limit, and timestamps.
4. The record is appended to `data/agents.json`.
5. An `agent_spawned` event is emitted on the EventBus.

This means each agent starts with its own isolated wallet and a tracked identity. The CLI command `agent:spawn --strategy simple` triggers this entire flow.

5.2 Agent Decision-Making

The project implements three distinct autonomous agent strategies, each following the **observe** → **decide** → **execute** → **log** pattern:

SimpleAgentRunner — Implements a probabilistic SOL transfer loop. Each iteration rolls a random number; if it exceeds the `holdProbability` threshold, the agent transfers SOL to a target address. This simulates a basic autonomous agent that makes independent transaction decisions on a timer.

```
for each iteration:
    roll = random()
    if roll >= holdProbability:
        sign and send SOL transfer
    sleep(interval)
```

TradingAgent — Simulates a price-based trading strategy. A mock price starts at 1.0 and follows a random walk (±5% per iteration). The agent evaluates:

- If price < `buyThreshold`: execute a buy (mint SPL tokens to the wallet).
- If price > `sellThreshold`: execute a sell (transfer SPL tokens to a burn address).
- Otherwise: hold.

The `evaluateMarket()` method encapsulates the decision logic, returning an action and reason string for auditability.

LiquidityAgent — Simulates liquidity provision by monitoring SPL token balances against a target range. The agent evaluates:

- If balance < `minBalance`: add liquidity (mint tokens to restore the balance).
- If balance > `maxBalance`: remove liquidity (transfer excess tokens to a burn address).
- Otherwise: hold.

The `evaluateBalance()` method drives this decision, checking the actual on-chain SPL balance each iteration via `TokenService.getSplBalances()`.

5.3 Separation of Agent Logic from Wallet Operations

The architecture enforces a clear boundary:

- **Agent layer** (`src/agents/`): Contains only decision logic. Agents call `TokenService` methods like `mintSpl()`, `transferSpl()`, and `transferSol()` to execute actions. They never touch keys, signing, or RPC details directly.
- **Wallet layer** (`src/wallet/`): Contains only key management, signing, and submission. It has no knowledge of trading strategies, price thresholds, or rebalancing logic.
- **Orchestrator layer** (`src/orchestrator/`): Bridges the two. It loads the agent's encrypted wallet, instantiates the appropriate strategy runner, and manages the execution lifecycle.

This separation means agent strategies can be swapped, added, or modified without touching the wallet security layer, and the wallet layer can be hardened or extended without affecting agent logic.

5.4 Multi-Agent Support

The Orchestrator maintains a JSON registry (`data/agents.json`) of all agents. Each agent is an independent entry with its own wallet, strategy, spending tracker, and timestamps. The system supports:

- **Spawning** multiple agents, each with a unique wallet (`spawnAgent()`).
- **Listing** all agents and their state (`listAgents()`).
- **Running** agents independently — each agent runs its own loop with its own wallet (`runAgentLoop()`, `runTradingAgent()`, `runLiquidityAgent()`).
- **Stopping** agents gracefully, cancelling their loop and emitting a shutdown event (`stopAgent()`).
- **Tracking** each agent's spending independently (`trackSpending()`, `checkSpendingLimit()`).
- **Snapshotting** SPL balances per agent for monitoring (`updateSplSnapshot()`).

The `runningAgents` Map in the Orchestrator tracks which agents are currently executing, preventing duplicate runs and enabling graceful stops.

5.5 Event System

The `EventBus` class provides a publish/subscribe mechanism. The Orchestrator emits events at key lifecycle points:

Event	When
`agent_spawned`	A new agent and wallet are created
`agent_started`	An agent begins its autonomous loop
`agent_stopped`	An agent is gracefully stopped
`agent_completed`	An agent finishes its loop and returns signatures

External systems can subscribe to these events via `eventBus.on()` to observe agent activity in real time.

5.6 CLI as Observation Layer

The CLI (`src/cli/index.ts`) exposes every operation as a command, serving as the primary interface for observing and controlling agents. All output is structured JSON, making it suitable for both human observation and programmatic consumption. Key commands:

- `wallet:create`, `wallet:balance`, `wallet:airdrop` — Direct wallet management.
- `token:create-mint`, `token:mint`, `token:transfer`, `token:balances` — SPL token operations.
- `agent:spawn`, `agent:list`, `agent:run`, `agent:stop` — Agent lifecycle.
- `agent:trade`, `agent:liquidity` — Strategy-specific autonomous loops.
- `agent:snapshot` — Capture and display current SPL token holdings.

---

6. Protocol Interaction on Devnet

The project interacts with the **SPL Token Program** as its test protocol. This is a real on-chain Solana program (not a mock), and all operations are executed on devnet with actual transactions:

- **Create a token mint**: The wallet creates a new SPL token mint where the wallet is the mint authority. This is a multi-instruction transaction that creates a mint account, initializes it, and assigns authority.
- **Mint tokens**: Tokens are minted to the wallet's associated token account using `mintTo()`. This exercises the mint authority signing path.
- **Transfer tokens**: SPL tokens are transferred between wallets using `transfer()`. Associated token accounts are created automatically via `getOrCreateAssociatedTokenAccount()` if they don't exist.
- **Query balances**: Token accounts are queried via `getParsedTokenAccountsByOwner()` to get current holdings.

These operations go beyond simple SOL transfers and demonstrate the wallet's ability to interact with Solana's token infrastructure autonomously.

---

7. Web Dashboard (Frontend)

7.1 Overview

In addition to the CLI, the project includes a full-featured web dashboard that provides a real-time graphical interface for managing agents and wallets. The dashboard is a single-page application served by an Express 5 server. It is started with `npm run dashboard` and runs at `http://localhost:3000`.

The frontend architecture has three layers:

- **`src/server/index.ts`** — Express server that serves static files from `public/`, mounts the API router at `/api`, and provides an SPA fallback for client-side routing.
- **`src/server/api.ts`** — RESTful API router that exposes all agent, wallet, and token operations as JSON endpoints. Each endpoint instantiates the same `WalletService`, `TokenService`, and `Orchestrator` classes used by the CLI.
- **`public/index.html`** — Self-contained SPA with embedded CSS and JavaScript. No build tools, bundlers, or frontend frameworks are required.

7.2 REST API

The API provides full feature parity with the CLI. Every operation available from the command line is also accessible via HTTP:

Agent endpoints:

Method	Path	Description
GET	`/api/agents`	List all agents with live SOL balances and running status
GET	`/api/agents/:id`	Get a single agent's details including SPL balances
POST	`/api/agents/spawn`	Spawn a new agent with a dedicated wallet
POST	`/api/agents/promote`	Promote an existing standalone wallet to a registered agent
POST	`/api/agents/:id/fund`	Request a devnet SOL airdrop to fund the agent
POST	`/api/agents/:id/stop`	Stop a running agent's strategy loop
POST	`/api/agents/:id/snapshot`	Capture the agent's current SPL token holdings
POST	`/api/agents/:id/run`	Run the simple transfer strategy (destination auto-generated if omitted)
POST	`/api/agents/:id/trade`	Run the price-based trading strategy
POST	`/api/agents/:id/liquidity`	Run the liquidity rebalancing strategy

Wallet endpoints:

Method	Path	Description
GET	`/api/wallets`	List all wallets with public keys and balances
POST	`/api/wallets/create`	Create a new encrypted wallet
GET	`/api/wallets/:name/balance`	Get SOL balance for a wallet
GET	`/api/wallets/:name/spl-balances`	Get all SPL token balances for a wallet
POST	`/api/wallets/:name/airdrop`	Request a devnet airdrop to the wallet

Token endpoints:

Method	Path	Description
POST	`/api/tokens/create-mint`	Create a new SPL token mint

POST	`/api/tokens/mint`	Mint SPL tokens to a wallet
POST	`/api/tokens/transfer`	Transfer SPL tokens between wallets

System:

Method	Path	Description
GET	`/api/health`	RPC connection health check with Solana version info

The agents list endpoint enriches each agent record with a live `balanceSol` value queried from the RPC and a `running` boolean flag from the Orchestrator's in-memory `runningAgents` map, allowing the UI to reflect real-time state.

7.3 Dashboard UI

The dashboard is a Solana-themed dark interface built with vanilla HTML, CSS, and JavaScript. It provides:

Header: Displays the project name and a live RPC connection status indicator (green dot when connected, red when disconnected). The health check polls `/api/health` on page load.

Agents Table: Shows all registered agents with columns for ID, wallet address, strategy badge, SOL balance, spending progress bar, and creation date. Each row has contextual action buttons:

- **Fund** — Opens a modal to request a devnet airdrop (configurable SOL amount).
- **Run / Trade / Liquidity** — Each opens a strategy-specific modal with configurable parameters. These buttons are only visible when the agent is idle.
- **Details** — Opens a slide-out panel showing full agent information, spending limits, and SPL token holdings.
- **Stop** — Visible only when the agent is actively running a strategy loop. Calls the stop endpoint and refreshes the table.

The Run/Trade/Liquidity vs. Stop button toggling provides clear visual feedback about each agent's current state.

Wallets Table: Lists all encrypted wallets on disk with their names, public keys, and SOL balances. Each row has:

- **Airdrop** — Request devnet SOL to the wallet.
- **Promote** — Convert a standalone wallet into a registered agent. This button is automatically disabled if the wallet is already associated with an agent, preventing duplicate registrations.
- **Details** — Slide-out panel with balance and SPL token information.

Spawn Agent: A floating action button opens a modal to create a new agent with a configurable wallet name, strategy selection, and spending limit.

Token Operations: Dedicated modals for creating SPL mints, minting tokens, and transferring tokens — all accessible from the Agents table context.

Auto-refresh: The agents and wallets tables refresh every 15 seconds to reflect balance changes and running state updates.

Toast Notifications: All operations display success or error feedback via toast notifications that appear at the bottom of the screen.

7.4 Promote Wallet to Agent

The dashboard introduces a "Promote" feature not available in the CLI. Standalone wallets (e.g., `agent-1` created directly via `wallet:create`) appear in the Wallets table but not in the Agents table. The Promote button calls `POST /api/agents/promote` which:

1. Checks the agent registry for duplicate wallet names.
2. Loads the existing encrypted wallet via `WalletService.loadWallet()` (does **not** create a new keypair).
3. Creates an `AgentRecord` with the existing wallet's address and a default strategy.
4. Appends the record to `data/agents.json` and emits an `agent_spawned` event.

This bridges the gap between standalone wallets and the agent management system.

7.5 Transaction Resilience

The `TokenService.transferSol()` method includes retry logic to handle Solana devnet congestion. Each transfer attempt:

1. Fetches a fresh `blockhash` and `lastValidBlockHeight` from the RPC.
2. Constructs the transaction with the explicit blockhash context.
3. Sends with `maxRetries: 3` and `preflightCommitment: "confirmed"`.
4. If the transaction fails with a `block height exceeded` or `Blockhash not found` error, it retries (up to 3 attempts total) with a new blockhash.
5. Non-blockhash errors are thrown immediately without retrying.

This ensures the simple agent's transfer loop can complete reliably even when devnet is under load.