VEXEL Feedback

What's going well (with concrete evidence)

1. Clear modular intent and separation of concerns

- The SDK is sensibly split into focused components:
    - src/wallet/WalletManager.ts (key management, encryption, storage)
    - src/signature/SignatureInjector.ts (signing/verification)
    - src/badge/BadgeMinter.ts (badge minting with simulation fallback)
    - src/haap/* (HAAP protocol, KYC abstraction)
- The API gateway is similarly decomposed:
    - src/api/APIGateway.ts wires Express + Socket.IO + middleware + Swagger

Why this matters: the "units" are small enough to test, replace, and reason about. That's a big deal in a system that spans web2 + web3 + identity.

2. TypeScript is configured strictly

- tsconfig.json has "strict": true. That's the right pain now to avoid worse pain later.

3. You wrote tests for key primitives

- There are meaningful tests for the core SDK pieces:
    - src/wallet/WalletManager.test.ts
    - src/signature/SignatureInjector.test.ts
    - src/badge/BadgeMinter.test.ts
    - src/haap/HAAPProtocol.test.ts
    - API middleware tests under src/api/__tests__/

Why this matters: you're testing the cryptographic "spine" (wallet/signing) and HAAP flow logic, which is exactly where regressions become expensive.

4. The documentation effort is unusually thorough

- You have serious phase docs and reviews (examples: WORKFLOW.md, docs/SECURITY_REVIEW.md, docs/contracts/SECURITY_REVIEW.md, database/README.md).
- The database schema is well-documented and includes triggers and views (see database/schema.sql, database/README.md).

Why this matters: docs like that are the difference between "project" and "archaeological dig".

5. Smart contract testing exists and is coherent

- Contract tests in test/contracts/AgentHeartbeat.test.ts cover registration, heartbeat recording, inactivity detection, and threshold updates.

What needs work (actionable and justified)

1. Build/package boundary is inconsistent: “features exist” but are excluded from compilation
   In tsconfig.json, you exclude major capabilities:

- src/database/**/*
- src/ipfs/**/*
- src/knowledge-base/**/*
- plus src/example.ts and src/service.ts

That means those modules don’t ship in dist/ and can’t be imported/used by consumers even though docs and examples describe them as part of the system.

Rationale: this creates a mismatch between:

- what the repo claims to provide (data layer, IPFS, knowledge migration)
- and what the compiled package actually exports/runs.

2. Dependency hygiene is currently leaky and misleading
   A quick scan of imports in build-included src/** shows the runtime code only needs a small set of packages (express, socket.io, ethers, helmet, etc.). But

package.json includes a bunch of deps that are either:

- unused by the compiled output (e.g. redis, uuid, did-jwt, did-resolver, arweave), or
- type packages incorrectly placed in dependencies (multiple @types/*).

Also: dotenv is imported by runtime code (src/api/server.ts) but is listed in devDependencies. In a real production install (where devDependencies aren't installed), the API server can crash.

Rationale: this inflates install surface, increases supply-chain risk, and creates "works on my machine" dependency behavior.

3. npm test is set up to fail in a clean environment
   jest.config.js runs tests under roots: ['<rootDir>/src'], which includes:

- integration tests that import DB/IPFS modules (src/__tests__/repository.test.ts, src/__tests__/ipfs.test.ts)

But pg and kubo-rpc-client are not declared in root package.json, so those imports will fail unless someone happened to install them manually.

Rationale: test runners should either:

- have all required deps declared, or
- separate integration tests behind an explicit command so unit tests pass by default.

4. The API gateway's "agent" endpoints are stubs, so the dashboard can't actually work

src/api/routes/agents.ts literally returns messages like “requires database integration”.

Meanwhile, the dashboard’s dashboard/src/services/api.ts expects real agent objects and status fields.

Rationale: this is not a “minor missing feature”. It means one of the biggest visible deliverables (dashboard) can’t be end-to-end exercised.

5. WebSocket authentication is basically “trust me bro”
   In src/api/websocket/WebSocketServer.ts, you accept socket.handshake.auth.userId as identity. No JWT validation. A client can claim to be anyone.

Rationale: if your system includes authorization logic (and it does), identity cannot be a client-supplied string.

6. Authorization is present, but not resource-scoped
   ActionVerificationMiddleware checks roles for action types, but it does not check that:

- the human owns the target agentId, or
- the agent is allowed to mutate only itself

Rationale: role-based checks without ownership checks are how you end up with “any authenticated human can update any agent”.

7. Contract/subgraph mismatch
   You have:

- contracts/AgentHeartbeat.sol (Heartbeat-based events)

- subgraph/subgraph.yaml configured for an AgentRegistry contract with events that do not exist in this repo.

Rationale: either the contract is missing, or the subgraph is for a different design. Right now it's not deployable as-is.

8. Documentation drift is already showing
   Examples and docs reference flows/scripts/modules that don't line up with the shipped build and dependencies (e.g., database/IPFS/knowledge migration references in docs vs tsconfig excludes and missing deps).

Rationale: doc drift kills adoption faster than bugs, because it makes users distrust everything.

AI-driven task backlog (granular, with acceptance criteria)

I'm going to assume you want tasks that can be executed mechanically. Each task below has: goal, files, steps, acceptance criteria.

A) Build/package correctness (P0)

BUILD-01: Fix runtime vs dev dependency classification
Goal: runtime code must not depend on devDependencies.
Files:

- package.json

- src/api/server.ts (uses dotenv)
  Steps:

1. Move dotenv from devDependencies → dependencies.

2. Run a “production-style” install test (documented): npm ci --omit=dev then run node dist/api/server.js.
   Acceptance:

- API server starts without module-not-found errors when devDependencies are omitted.

BUILD-02: Decide and enforce what the published package includes
Goal: either ship DB/IPFS/knowledge-base modules or clearly quarantine them as “internal/experimental”.
Files:

- tsconfig.json

- src/index.ts

- docs/examples (multiple)
  Steps (Option A: ship them):

1. Remove src/database, src/ipfs, src/knowledge-base from tsconfig.json excludes.

2. Add missing runtime deps (pg, kubo-rpc-client) to dependencies.

3. Export the modules from src/index.ts (or a documented subpath export).
   Acceptance:

- npm run build produces dist/database/*, dist/ipfs/*, dist/knowledge-base/*.

- Consumers can import the modules without reaching into src/.

Steps (Option B: don't ship them yet):

1. Keep excludes, but move those directories under something like experimental/ or packages/.

2. Remove their references from "mainline" docs and root examples.
   Acceptance:

- README/SETUP/TESTING do not claim those features exist in the shipped SDK.

BUILD-03: Dependency audit and pruning
Goal: remove unused deps or wire them into real implementations.
Files:

- package.json

- codebase import scan results
  Steps:

1. Remove unused deps from dependencies (redis, uuid, did-jwt, did-resolver, arweave) unless you actually compile and ship the modules that use them.

2. Move all @types/* packages into devDependencies.
   Acceptance:

- All dependencies in dependencies are referenced by compiled dist/** output (or explicitly justified in docs).

BUILD-04: Add explicit package boundaries for multi-project repo
Goal: make dashboard/subgraph clearly separate install units.
Files:

- root README.md

- root package.json
  Steps:

1. Document clearly: root package is SDK+API+contracts; dashboard and subgraph have their own package.json.

2. Add root scripts: npm run dashboard:dev, npm run subgraph:codegen, etc (delegating into subfolders).
   Acceptance:

- New contributor can discover how to run each component from root scripts.

B) Testing & CI stability (P0)

TEST-01: Split unit tests vs integration tests
Goal: npm test runs fast and passes without Postgres/IPFS.
Files:

- jest.config.js

- src/__tests__/repository.test.ts

- src/__tests__/ipfs.test.ts

  Steps:

1. Create jest.unit.config.js and jest.integration.config.js (or use Jest “projects”).
2. Move DB/IPFS tests into src/__tests__/integration/ and configure integration runner to include them only.
3. Update scripts:
    - test: runs unit only
    - test:integration: runs integration

      Acceptance:

- npm test passes on a clean machine with no DB/IPFS.
- npm run test:integration runs DB/IPFS tests when services are available.

TEST-02: Fix Jest toolchain version mismatch risk

Goal: align Jest + ts-jest major versions.

Files:

- package.json
- jest.config.js

  Steps:

1. Pick a consistent pair:

    - Either downgrade jest to a version compatible with ts-jest@29, or
    - upgrade ts-jest to match Jest major version.

2. Update lockfile.
   Acceptance:

- npm test runs without ts-jest preset errors.

CI-01: Add GitHub Actions workflow that actually runs what docs imply
Goal: enforce “green main” behavior.
Files:

- .github/workflows/ci.yml (new)
  Steps:

1. Add jobs for:
    - build (npm ci, npm run build)
    - unit tests (npm test)
    - contracts (npm run compile, npm run test:contracts)
2. Optional job: dashboard lint/test by running npm in dashboard/.
   Acceptance:

- PRs fail if build/tests fail.
- Workflow time stays reasonable (unit vs integration split makes this realistic).

CI-02: Add integration test workflow with Postgres service

Goal: make DB schema and repository tests real, not aspirational.

Files:

- .github/workflows/integration.yml (new)

  Steps:

1. Spin up postgres:14 service.
2. Set env vars used by DB client.
3. Run npm run test:integration.

   Acceptance:

- Integration workflow passes end-to-end including DB tests.

C) API gateway security and correctness (P1)

SEC-01: WebSocket authentication via JWT, not client-supplied userId

Goal: prevent identity spoofing.

Files:

- src/api/websocket/WebSocketServer.ts
- src/api/middleware/auth.ts
- examples/websocket-client-example.ts

  Steps:

1. Require JWT in handshake (e.g., socket.handshake.auth.token).

2. Verify token server-side, derive userId/role from token.

3. Reject connection if missing/invalid.

4. Update example client to send token.
   Acceptance:

- A client cannot connect with an invalid token.

- Connected socket sessions have server-trusted identity fields.

SEC-02: Fix CORS config contradiction
Goal: avoid invalid origin="*" with credentials: true.
Files:

- src/api/APIGateway.ts

- WebSocket CORS config in constructor
  Steps:

1. If origin is "*", set credentials: false.

2. If credentials are required, enforce explicit allowlist origins.
   Acceptance:

- Browser clients can authenticate without CORS blocking.

- Configuration is internally consistent.

SEC-03: Resource-scoped authorization for actions
Goal: prevent cross-agent tampering.
Files:

- src/api/middleware/actionVerification.ts

- future DB integration in routes
  Steps:

1. Extend verification to check ownership:

    - If action targets agentId, verify request user owns agentId (DB lookup) or matches agentId for agent role.

2. Add a policy matrix: action type → required role + ownership rule.
   Acceptance:

- Human users can only mutate agents they own.

- Agent role can only mutate itself.

SEC-04: Replace “token vending machine” login with a real dev/prod split
Goal: avoid shipping “POST /login, pick your role” into production by accident.
Files:

- src/api/routes/auth.ts
  Steps:

1. Gate the current login route behind NODE_ENV !== 'production' or a DEV_AUTH_ENABLED=true flag.

2. In production mode, require real auth (even a placeholder API key check is better than none).
   Acceptance:

- In production, /api/auth/login cannot mint admin tokens from thin air.

D) API <-> Dashboard integration (P1)

API-01: Implement real agent CRUD backed by DB repository
Goal: make /api/agents return actual agents, not apology messages.
Files:

- src/api/routes/agents.ts
- src/database/* (if you choose to ship/enable it)
- database/schema.sql
  Steps:

1. Initialize DatabaseClient in API gateway (configurable via env).
2. Use AgentRepository to implement:
   - GET /api/agents
   - GET /api/agents/:agentId
   - PUT /api/agents/:agentId/status
   - GET /api/agents/:agentId/capabilities

3. Normalize response shapes (don't nest agents inside data.agents while dashboard expects data array, etc).
   Acceptance:

- Dashboard getAgents() returns a list of real agents.

- Status updates persist and can be re-fetched.

API-02: Add request validation (zod or similar)
Goal: stop trusting req.body to behave like an adult.
Files:

- src/api/routes/auth.ts

- src/api/routes/agents.ts
  Steps:

1. Add schemas for each route input.

2. Validate and return 400 with specific errors.
   Acceptance:

- Malformed inputs get predictable 400 responses.

- No route relies on implicit types from JSON bodies.

API-03: Make Swagger generation work from dist
Goal: /api-docs shouldn't silently break after compilation.
Files:

- src/api/APIGateway.ts
  Steps:

1. Change apis: ['./src/api/routes/*.ts'] to a path that exists at runtime:
    - use path.join(__dirname, 'routes/*.js') in dist
    - or include source files intentionally
      Acceptance:

- Swagger UI shows endpoints when running compiled dist server.

DASH-01: Align dashboard agent types with API reality
Goal: stop type drift (active/inactive vs ACTIVE/SLEEP/TERMINATED).
Files:

- dashboard/src/types/index.ts
- dashboard/src/services/api.ts
  Steps:

1. Either:
    - map API runtime status enum → dashboard status union, or
    - change dashboard to use API enum directly.
2. Update getDashboardStats() logic accordingly.
   Acceptance:

- Dashboard shows correct status counts from real API responses.

E) Wallet/key security hardening (P1)

WALLET-01: Enforce encryption key in production and fix file permissions
Goal: wallet storage isn't "security theater".
Files:

- src/wallet/WalletManager.ts
  Steps:

1. If NODE_ENV === 'production' and no WALLET_ENCRYPTION_KEY, throw at startup (not later).

2. When writing wallet files, set permissions 0o600.
   Acceptance:

- Production run fails fast without encryption key.

- Wallet files are not world-readable on disk.

WALLET-02: Stop returning mnemonic by default
Goal: reduce accidental secret exfiltration.
Files:

- src/wallet/WalletManager.ts

- src/wallet/WalletManager.test.ts
  Steps:

1. Add option returnMnemonic?: boolean default false.

2. Only include mnemonic when explicitly requested.
   Acceptance:

- createWallet() does not leak mnemonic unless configured.

F) HAAP protocol correctness & durability (P2)

HAAP-01: Replace weak token ID generation with crypto-grade randomness
Goal: avoid predictable token IDs.
Files:

- src/haap/HAAPProtocol.ts
  Steps:

1. Generate token IDs via crypto.randomBytes(16).toString('hex') or UUID v4.
   Acceptance:

- Token IDs are not based on time/random substring.

HAAP-02: Persist tokens and KYC status (or remove the pretense)
Goal: tokens surviving process restarts is table-stakes.
Files:

- src/haap/HAAPProtocol.ts

- src/haap/KYCService.ts
  Steps (choose one):

1. Implement Redis-backed storage (you already depend on redis but don't use it).

2. Or implement DB-backed storage.
   Acceptance:

- After restart, validateToken(tokenId) still works.

G) Contracts + subgraph coherence (P2)

CHAIN-01: Fix DID squatting / unauthorized registration in AgentHeartbeat
Goal: prevent attackers registering someone else's DID.
Files:

- contracts/AgentHeartbeat.sol

- test/contracts/AgentHeartbeat.test.ts
  Steps:

1. Require msg.sender == agentAddress in registerAgent, or require an EIP-712 signature proving control.

2. Add tests for attempted squatting.
   Acceptance:

- Unauthorized registration attempts revert.

- Tests cover the attack path.

SUBGRAPH-01: Make subgraph match the deployed contract set
Goal: subgraph should be deployable.
Files:

- subgraph/subgraph.yaml
- subgraph/abis/*
- subgraph/src/mapping.ts

Steps (pick one direction):

1. If AgentRegistry is the intended contract, add contracts/AgentRegistry.sol + hardhat compile artifacts + deployment scripts.
2. If AgentHeartbeat is intended, rewrite subgraph to index its events and update schema accordingly.

Acceptance:

- graph codegen and graph build succeed.
- Indexed events match actual chain events.

H) Documentation drift control (P2)

DOC-01: Create a “single source of truth” run matrix

Goal: stop docs lying by accident.

Files:

- README.md
- SETUP.md
- TESTING.md

Steps:

1. Add a matrix: component → where to run → command → required services.

2. Remove/flag any steps that reference excluded modules unless those modules are shipped.
   Acceptance:

- A new dev can run SDK tests, API server, dashboard, and contract tests without guessing.

DOC-02: Make examples compile against the published surface
Goal: examples should only import what's actually exported/built.
Files:

- examples/*

- src/index.ts
  Steps:

1. For each example, ensure imports target the public API.

2. Remove examples that depend on excluded modules, or move them under experimental/.
   Acceptance:

- npx ts-node examples/<file>.ts works after npm install.

-