PROMISE SENTINEL

Protocol Hardening Edition

Team Lead: Shashwat Shekhar **Team Member:** Devesh Mudaliar
Team Lead: Harsh Sunil Phuke

December 29, 2025

Tagline: Crash-Proof Transactions for Unreliable Networks

FOR MERCHANT CONSOLE GO TO https://promise-sentinel.vercel.app/ and then click on INITIALIZE SENTINEL_OS

FOR GOD MODE/ TECH VIEW GO TO https://promise-sentinel.vercel.app/god-mode and then click on INITIALIZE SENTINEL_OS

Contents

1 Problem Statement

1.1 The Context: High-Density Chaos

In the modern ”Experience Economy”—music festivals, conventions, and pop-up markets—reliable connectivity is a myth. Events with 50,000+ attendees saturate local cell towers, creating a phenomenon known as **”Lie-Fi”**: devices show full signal bars but cannot transmit data.

1.2 The Technical Failure Points

Standard Point-of-Sale (POS) applications (like Square or web-based Stripe terminals) rely on ”Happy Path” assumptions:

1. **Stable Connectivity:** They assume a request sent to the server will receive a response within milliseconds. When this fails, the UI hangs, spins, or crashes.
2. **Benign Browser Environments:** They assume `localStorage` is available and writable. However, in high-stress environments, devices often enter ”Hostile States”:
   - **Safari Private Mode:** Silently blocks `localStorage` writes, causing data loss without error messages.
   - **Quota Exhaustion:** Cheap tablets fill up their storage cache quickly, causing write operations to throw fatal exceptions.
3. **The ”White Screen of Death”:** Most React applications do not handle storage-level exceptions gracefully. A single unhandled promise rejection during a transaction save can unmount the entire application tree, forcing a hard refresh and losing the customer’s cart.

1.3 The Business Impact

For a festival merchant, a 30-second POS crash during peak hours isn’t just an annoyance; it is a **revenue hemorrhage**. If a terminal goes down, the line moves to the next vendor. Reliability is not a feature; it is the product.

2 Solution Overview

2.1 Philosophy: Protocol Hardening

Promise Sentinel is not just a POS app; it is a **ruggedized protocol** designed for hostile digital environments. We inverted the standard web architecture: instead of treating the network as a requirement, we treat it as an optional utility.

2.2 The "SafeVault" Engine

At the core of Promise Sentinel is the **SafeVault**, a custom storage engine architected to withstand catastrophic failure.

- **Air-Gap Defense:** The system actively monitors the health of the browser's storage mechanisms. If it detects "hostility" (e.g., `localStorage` access is denied or full), it automatically hot-swaps the entire storage layer to an in-memory "RAM Vault." This transition happens in < 10ms, is invisible to the user, and guarantees the transaction is captured.
- **Cryptographic Integrity:** To prevent data tampering on unsecured festival tablets, every transaction is encrypted with **AES-GCM** before it is ever written to disk or memory.

2.3 Agentic Engineering with Kiro

This project serves as a case study in **AI-Augmented Engineering**. Rather than using AI to generate boilerplate code, we utilized the **Kiro IDE** as an autonomous architect.

- **Constitution-Led Development:** We defined a strict "Constitution" (`tech.md`) that forbade unsafe types and enforced architectural patterns.
- **Agentic Guardrails:** We deployed automated hooks (`ts-guard.json`) that monitored our codebase in real-time, preventing the introduction of weak typing (`any`) or loose interfaces.
- **Migration Strategy:** We successfully executed a live migration of the core engine from legacy JavaScript to strict TypeScript without disrupting the application's runtime logic.

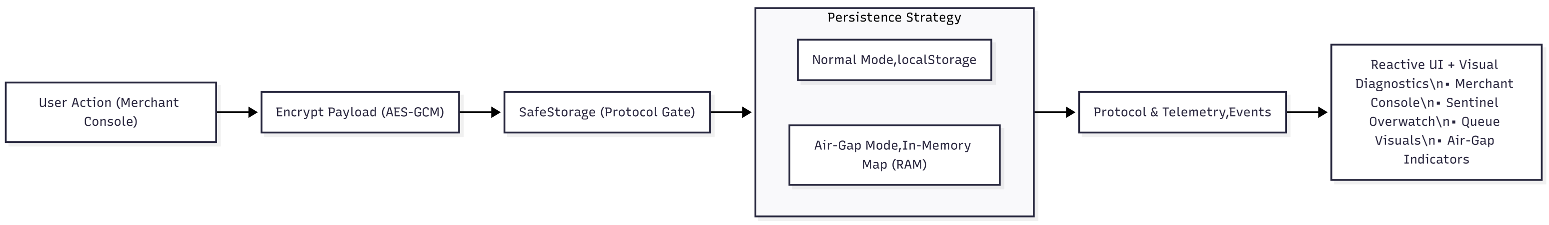
3 Technologies Used

3.1 Core Stack

- **Frontend Framework:** React 18 (Vite) – Chosen for its concurrent rendering features and lightweight production build.
- **Language:** Strict TypeScript (Migration Target) – Enforces type safety on all cryptographic payloads and storage interfaces.
- **Styling:** Tailwind CSS – Enables rapid UI development with a "Cyber-Industrial" aesthetic tailored for high-contrast visibility in outdoor environments.

3.2 Architecture & Storage

- **State Management:** React Context API + Custom "Theatrical" Providers.
- **Persistence:** `window.localStorage` (Primary) + `Map<string, string>` (Air-Gap Failover).
- **Encryption:** Web Crypto API (AES-GCM 256-bit).


User Action (Merchant Console)
Encrypt Payload (AES-GCM)
SafeStorage (Protocol Gate)
Persistence Strategy
Normal Mode,localStorage
Air-Gap Mode,In-Memory Map (RAM)
Protocol & Telemetry,Events
Reactive UI + Visual Diagnostics\n▪ Merchant Console\n▪ Sentinel Overwatch\n▪ Queue Visuals\n▪ Air-Gap Indicators

3.3 AI & Engineering Tooling

- **Kiro IDE:** The primary development environment.
- **Kiro Specs:** Used to reverse-engineer legacy code and generate formal `requirements.md` artifacts.
- **Kiro Hooks:** Custom JSON-based triggers used to enforce code quality standards on every file save.

3.4 Testing & Verification

- **Jest:** Test runner.
- **Fast-Check:** Property-based testing framework. Used to generate 100+ random permutations of transaction data to mathematically prove the ”SafeVault” cannot be crashed by malformed inputs.

4 System Architecture

4.1 The Singleton Vault Pattern

The architecture relies on a strict Singleton pattern to ensure only one ”Source of Truth” exists for transaction data, regardless of component re-renders.

Logic Flow:

1. **Initialization:** App boots → `SafeStorage.getInstance()` is called.
2. **Probe Phase:** The Vault attempts a ”Canary Write” to disk.
   - *Success:* System initializes in **Normal Mode**.
   - *Failure (Quota/Private Mode):* System initializes in **Air-Gap Mode** (RAM only).
3. **Transaction Lifecycle:**
   - User inputs amount → Data is serialized → AES-GCM Encrypted → Passed to Vault.
   - Vault writes to current storage medium (Disk or RAM).
   - Vault emits `SentinelStorageEvent` → UI updates instantly.

4.2 Protocol Hardening Pipeline

4.3 Data Synchronization Graph

Unlike traditional lists, sync status is managed via a **Force-Directed Graph**.

- **Nodes:** Represent individual transactions.
- **Edges:** Represent dependency chains (e.g., a refund dependent on a sale).
- **Color State:**
  - **Red:** Unsynced / Local Only.
  - **Amber:** Encryption/Hashing in progress.
  - **Green:** Synced to Cloud.

```
Legacy safeVault.js -> [Reverse Engineering] -> Kiro Spec Analysis
        |
        v
Strategy Decision -> [Define Constraints] -> tech.md / Constitution
        |
        v
SafeStorage.ts (Singleton) -> [Guardrail Check] -> ts-
guard.json Hook
        |
        v
Fast-Check Verification -> Production Deployment
```

Figure 1: The Protocol Hardening Migration Pipeline

5 Core Features

5.1 The "Air-Gap" Failover

This is the application's killer feature. It renders the application crash-proof against storage errors.

- **Mechanism:** Uses a `try-catch` wrapper around low-level `setItem` calls. If a write fails, the system catches the error, flags the environment as "Hostile," and transparently routes the data to a private `MemoryStore`.
- **User Feedback:** The UI displays a "SYSTEM HARDENED" badge, notifying the operator they are running on RAM and should not refresh the page.

5.2 Cryptographic "Hex Inspector"

To build trust with technical operators, the application exposes its internal workings via the "God Mode" panel.

- **Visualizer:** A real-time stream of the raw hexadecimal bytes being written to storage.
- **Utility:** Allows developers to verify that data is actually encrypted at rest, ensuring GDPR/CCPA compliance even on lost devices.

5.3 "Theatrical" User Interface

Designed for high-stress environments where operator fatigue is real.

- **Audio Cues:** Uses distinct mechanical sounds ("Clunk", "Whir", "Chime") for successful actions. An operator can hear if a transaction failed without looking at the screen.
- **Glitch Effects:** When "Chaos Mode" is enabled (simulating network failure), the UI digitally artifacts and distorts, providing immediate visceral feedback that the system is under stress.

5.4 Chaos Mode Simulator

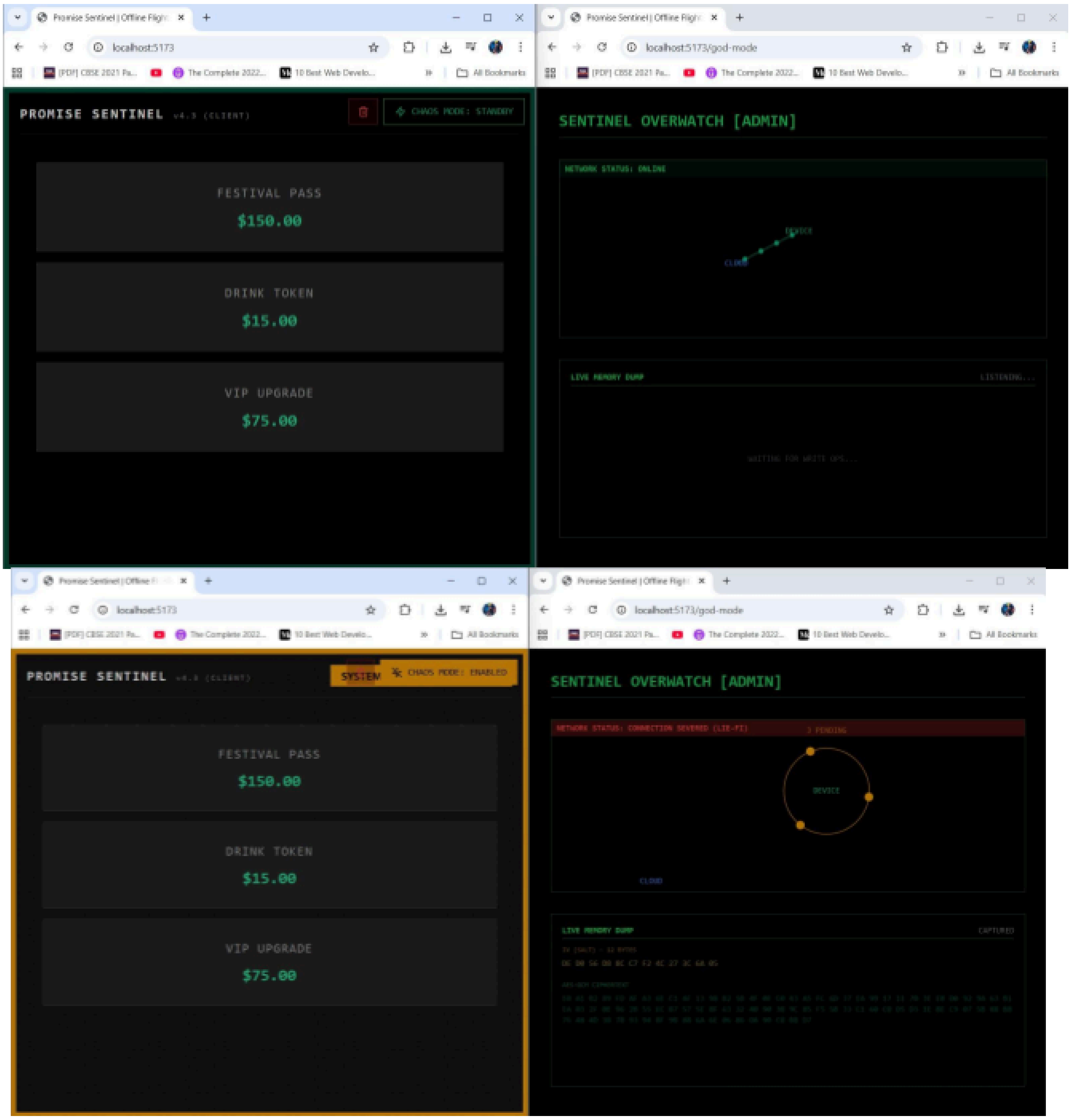
A built-in developer tool accessible via the dashboard.

- **Function:** Intentionally breaks the network connection and floods the `localStorage` with garbage data.
- **Purpose:** Allows instant demonstration of the Air-Gap failover mechanism to stakeholders without needing complex environment setups.

FOR MERCHANT CONSOLE GO TO https://promise-sentinel.vercel.app/ and then click on INITIALIZE SENTINEL_OS

FOR GOD MODE/ TECH VIEW GO TO https://promise-sentinel.vercel.app/god-mode and then click on INITIALIZE SENTINEL_OS

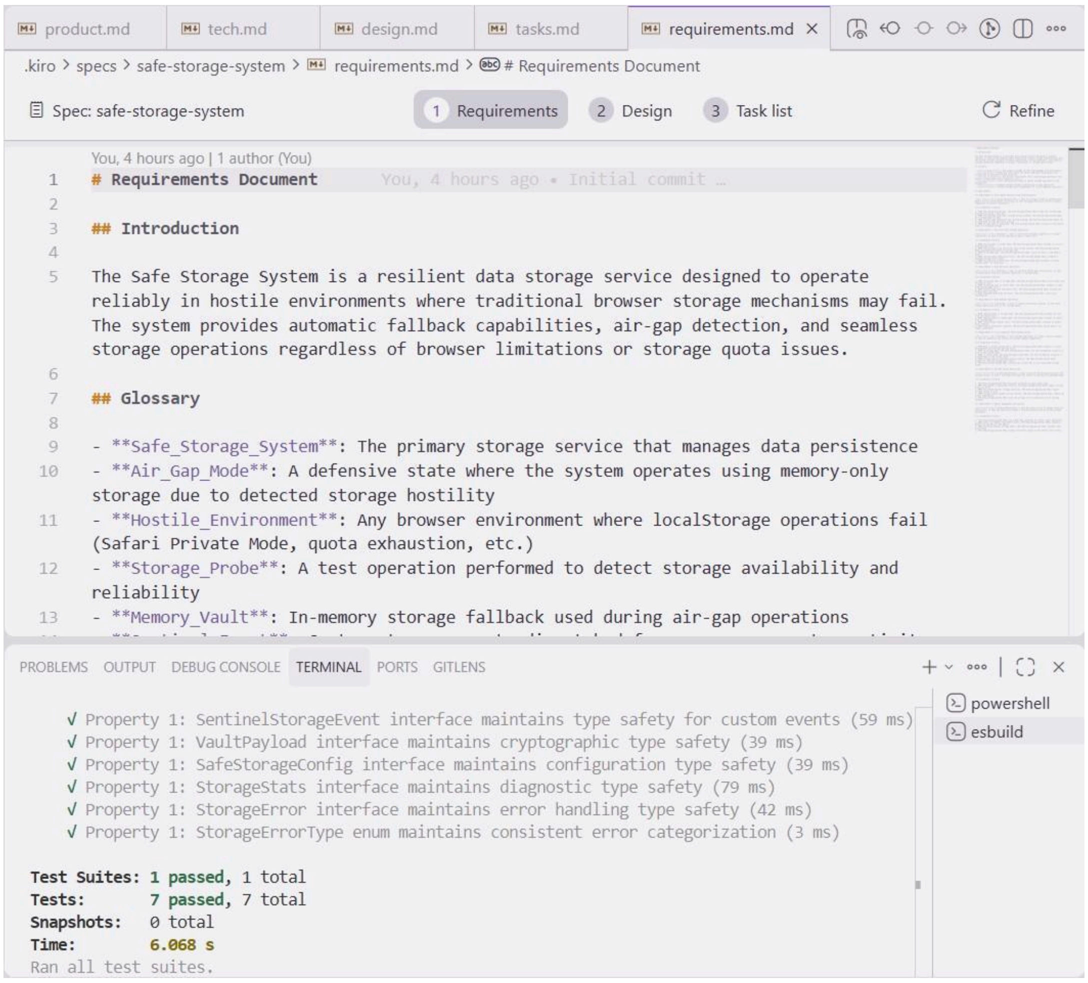
6 Visuals



7 Challenges & Learnings

7.1 Challenge 1: The ”Any” Type Trap

Problem: When migrating legacy JS to TypeScript using AI, LLMs often default to using `any` types to silence errors quickly. This defeats the purpose of ”Hardening.”
Solution: We implemented the **Kiro Hook (`ts-guard.json`)**. This was a breakthrough moment. By configuring the IDE to actively ”scold” us (or the agent) whenever an `any` type was saved, we gamified the strictness of our codebase.



```
# Requirements Document        You, 4 hours ago • Initial commit …

## Introduction

The Safe Storage System is a resilient data storage service designed to operate
reliably in hostile environments where traditional browser storage mechanisms may fail.
The system provides automatic fallback capabilities, air-gap detection, and seamless
storage operations regardless of browser limitations or storage quota issues.

## Glossary

- **Safe_Storage_System**: The primary storage service that manages data persistence
- **Air_Gap_Mode**: A defensive state where the system operates using memory-only
storage due to detected storage hostility
- **Hostile_Environment**: Any browser environment where localStorage operations fail
(Safari Private Mode, quota exhaustion, etc.)
- **Storage_Probe**: A test operation performed to detect storage availability and
reliability
- **Memory_Vault**: In-memory storage fallback used during air-gap operations
```

```
  √ Property 1: SentinelStorageEvent interface maintains type safety for custom events (59 ms)
  √ Property 1: VaultPayload interface maintains cryptographic type safety (39 ms)
  √ Property 1: SafeStorageConfig interface maintains configuration type safety (39 ms)
  √ Property 1: StorageStats interface maintains diagnostic type safety (79 ms)
  √ Property 1: StorageError interface maintains error handling type safety (42 ms)
  √ Property 1: StorageErrorType enum maintains consistent error categorization (3 ms)

Test Suites: 1 passed, 1 total
Tests:       7 passed, 7 total
Snapshots:   0 total
Time:        6.068 s
Ran all test suites.
```

7.2 Challenge 2: Browser Event Limitations

Problem: The native `StorageEvent` in browsers only fires when storage changes in *another* tab. It does not fire for changes in the *current* tab. This broke our reactivity.
Solution: We engineered a custom `SentinelStorageEvent` that extends the native interface. The `SafeStorage` singleton manually dispatches this event to the `window` object, forcing the UI to update instantly across all components without needing a heavy state management library like Redux.

7.3 Challenge 3: Vercel Build Conflicts

Problem: Our animation library (`use-dencrypt-effect`) relied on an older React peer dependency, causing modern build pipelines to crash.
Solution: We learned to override Vercel's default build settings to use the `-legacy-peer-deps` flag, ensuring we could maintain our "Theatrical" aesthetic without rewriting core dependen- cies.

8 Future Roadmap

Phase 2: Peer-to-Peer Mesh Sync

Goal: Allow devices to sync data with *each other* via WebRTC or Bluetooth Low Energy (BLE) when the cloud is unreachable.
Status: Architecture planning phase. The Graph Sync model is already compatible with mesh topology.

Phase 3: Hardware Security Keys

Goal: Integrate WebAuthn (YubiKey) support for authorizing high-value transactions (e.g., VIP upgrades > $500).
Status: Research phase.

Conclusion

Promise Sentinel proves that web applications can be as robust as industrial embedded systems. By combining **Protocol Hardening** architecture with **Kiro's Agentic Workflow**, we have built a POS terminal that refuses to die. We didn't just write code; we engineered a survival mechanism for the offline economy.