Group 8: Amazon FTR 5

Clean Bot - System Architecture Documentation

Project Overview

Clean Bot is an autonomous cleaning robot that collects small trash items and debris from outdoor areas. This application allows user to control the robot from their devices. The app also provides real-time robot status monitoring, deployment controls, and project information display.

Tech Stack:

Frontend Framework: React 18.3.1

Language: TypeScript

Build Tool: Vite 5.4.19

UI Component Library: shadcn/ui (Radix UI primitives)

Styling: Tailwind CSS 3.4.17

Routing: React Router v6

State Management: React Hooks (local state) + React Query

Form Handling: React Hook Form + Zod

Charts: Recharts 2.15.4

Notifications: Sonner + Radix Toast

System Architecture

High-Level Architecture Diagram

Component Breakdown

Layout & Navigation Components

Navigation (Navigation.tsx): Sticky header with logo, desktop nav menu, mobile hamburger menu
Header (Header.tsx): Additional header UI component
Footer (Footer.tsx): Application footer

Content Sections (Marketing/Information)

HeroSection (HeroSection.tsx): Landing hero with CTA buttons, animated branding
TeamSection (TeamSection.tsx): Team member profiles
ProblemSection (ProblemSection.tsx): Problem statement / motivation

ProjectSection (ProjectSection.tsx): Project details and technical info

UserStoriesSection (UserStoriesSection.tsx): User stories and use cases

Interactive Dashboard Components

DashboardSection (DashboardSection.tsx): Main stateful container managing robot state

Manages: deployState, robotStatus, battery, location, cleaningProgress Handles deployment/stop logic with setTimeout-based state transitions
Integrates battery depletion simulation

DeployButton (DeployButton.tsx): Visual control button

States: idle, deploying, deployed

Disabled when deploying

Visual feedback with gradient colors and animations

RobotStatusCard (RobotStatusCard.tsx): Status display with connectivity indicator

Displays: current status, location, runtime, area cleaned

Shows connection status (Wifi/WifiOff icons)

Dynamic styling based on robot status (idle, cleaning, returning, charging, offline) **BatteryIndicator** (BatteryIndicator.tsx): Battery level visualization

Color-coded: red (<20%), orange (20-50%), green (50%+)

Shows charging state with pulse animation

Displays percentage and charging status

CleaningProgress (CleaningProgress.tsx): Progress visualization

Shows percentage and area cleaned in m²

Completion badge when 100%

Active/inactive state indicator

UI Component Library (/src/components/ui)

40+ shadcn/ui components built on Radix UI:

Form Components: form, input, textarea, select, checkbox, radio-group, toggle, switch **Display Components**: card, badge, alert, progress, skeleton, table **Dialog/Overlay**: dialog, alert-dialog, dropdown-menu, context-menu, popover, tooltip, hover-card **Navigation**: navigation-menu, menubar, pagination, tabs, breadcrumb **Layout**: sidebar, resizable, scroll-area, separator, sheet, drawer

Data Display: carousel, chart, accordion, collapsible

Misc: button, label, command, calendar, input-otp, aspect-ratio

Custom Hooks (/src/hooks)

use-toast (hooks/use-toast.ts): Toast notification management

use-mobile (hooks/use-mobile.tsx): Mobile responsiveness detection

Utilities (/src/lib)

utils.ts: cn() function for merging Tailwind class names with class-variance-authority

Data Flow & State Management

State Architecture

Local Component State (React Hooks)

All application state is managed locally within DashboardSection component

Uses useState for managing multiple state variables

State Variables in DashboardSection:

```
const [deployState, setDeployState] = useState<"idle" | "deploying" | "deployed"> ("idle")
const [robotStatus, setRobotStatus] = useState<"idle" | "cleaning" | "returning" | "charging" |
"offline">("idle")
const [battery, setBattery] = useState(78)
const [location, setLocation] = useState("Home Base")
const [cleaningProgress, setCleaningProgress] = useState(0)
```

Data Flow Diagram

User Action (Click Deploy Button)
↓
DeployButton.onDeploy() callback
↓
DashboardSection.handleDeploy()
↓
Update deployState → "deploying"
Show Toast Notification
↓
setTimeout (2000ms)
↓
setDeployState("deployed")
setRobotStatus("cleaning")
setLocation("Zone A - North Side")
Show Success Toast
↓
Child Components Re-render with new props
↓
UI Updates (button color, status badge, animations)

Notification System

Sonner: Toast notifications for user feedback (info, success)

Toast/Toaster: Secondary notification system from shadcn/ui Centralized notification dispatching via toast callbacks

State Transitions

Deploy Flow:

idle → deploying (2s) → deployed
↓
robot: cleaning

Stop Flow:

deployed → idle
↓
returning (3s) → idle

↓
robot: idle

Battery Management:

Simulated depletion during cleaning (every 100ms, -2% per tick) Auto-triggers return when battery ≤ 20%
Shows "Returning (Low Battery)" location

External Dependencies & Integrations

Core Dependencies

Package Version Purpose

react ^18.3.1 UI framework

react-dom ^18.3.1 DOM rendering react-router-dom ^6.30.1 Client-side routing @tanstack/react-query ^5.83.0 Data fetching & caching typescript ^5.8.3 Type safety

UI & Styling

Package Version Purpose

tailwindcss ^3.4.17 Utility-first CSS

@radix-ui/* Latest Accessible UI primitives

Package Version Purpose

shadcn/ui - Component library (bundled)

lucide-react ^0.462.0 Icon library

sonner ^1.7.4 Toast notifications

Form & Validation

Package Version Purpose

react-hook-form ^7.61.1 Form state management @hookform/resolvers ^3.10.0 Form validation resolvers zod ^3.25.76 Schema validation

Utilities

Package Version Purpose

clsx ^2.1.1 Conditional classnames tailwind-merge ^2.6.0 Merge Tailwind classes intelligently class-variance-authority ^0.7.1 CSS-in-JS

variants date-fns ^3.6.0 Date manipulation next-themes ^0.3.0 Theme management

Charts & Data Visualization

Package	Version	Purpose
recharts	^2.15.4	React charting library
embla-carousel-react	^8.6.0	Carousel component

Build Tools

Package	Version	Purpose
vite	^5.4.19	Build tool & dev server
@vitejs/plugin-react-swc	^3.11.0	SWC compiler for React
eslint	^9.32.0	Code linting
lovable-tagger	^1.1.11	Component tagging (dev)

Technical Debt & Issues

1: Monolithic State Management in Single Component

Category: Architectural Debt

Related Requirements: SR-UI-01, SR-UI-02, AC 1.5, AC 2.3

Related Design Components: DC-SW-02 (Status Reporting API), DC-SW-03 (Monitoring Dashboard API)

Description: All robot state (deployState, robotStatus, battery, location, cleaningProgress) is managed within a single DashboardSection component using local React state. There is no separation of concerns, no global state management solution (Redux, Zustand, Context API), and no clear API boundaries between UI components and business logic. This creates a monolithic structure where state logic is tightly coupled to presentation, making it difficult to test, scale, or reuse state management logic across the application.

Impact on Requirements:

> **SR-UI-01:** Cannot provide reliable status notifications without proper state management **SR-UI-02:** Cannot transmit real-time operational data without centralized state
> **AC 1.5, 4.3:** Status update notifications require persistent state management
> **AC 2.3:** Real-time location and activity updates need global state access

Remediation Plan:

> 1. Implement a global state management solution (Zustand recommended for simplicity) 2. Extract robot state logic into a dedicated store (useRobotStore)
> 3. Create a service layer for robot operations (robotService.ts) aligned with DC-SW-02 4. Implement clear separation between state management, business logic, and UI components 5. Add state persistence layer (localStorage or backend session) to support SR-UI-02 requirements

Files Affected: src/components/DashboardSection.tsx

2: Missing Backend API Integration Layer

Category: Architectural Debt

Related Requirements: SR-UI-01, SR-UI-02, SR-DATA-01, SR-DATA-02, AC 1.5, AC 2.3, AC 3.4, AC 3.5
Related Design Components: DC-COMM-01 (Wi-Fi/Cellular Communication), DC-SW-02 (Status Reporting API), DC-COMM-02 (Real-time Telemetry Link), DC-SW-03 (Monitoring Dashboard API), DC-SW-04 (Data Logging), DC-SW-05 (Data Export API)

Description: The application lacks any API communication layer. All robot operations are simulated client-side using setTimeout calls. There is no HTTP client setup (axios, fetch wrapper), no API service layer, no WebSocket implementation for real-time updates, and no abstraction for backend communication. The code directly manipulates state with hardcoded delays, creating a prototype that cannot connect to actual robot hardware or any backend server.

Impact on Requirements:

SR-UI-01: Cannot provide status notifications without backend API (DC-SW-02)

SR-UI-02: Cannot transmit real-time operational data without communication module (DC-COMM-02) **SR-DATA-01:** Cannot generate daily reports without data logging API (DC-SW-04) **SR-DATA-02:** Cannot export data without export API (DC-SW-05)

AC 2.3: Real-time location updates require WebSocket/telemetry link (DC-COMM-02) **AC 3.4, 3.5:** Daily reports and data export require backend integration

Remediation Plan:

1. Create an API client module (src/lib/api/client.ts) with axios or fetch wrapper (implements DC COMM-01)
2. Implement API service layer (src/services/robotService.ts) aligned with DC-SW-02, DC-SW-03:
   deployRobot() method
   stopRobot() method
   getRobotStatus() method
   subscribeToRobotUpdates() WebSocket method (DC-COMM-02)
   getDailyReports() method (DC-SW-04)
   exportData() method (DC-SW-05)
3. Add WebSocket client for real-time robot status updates (DC-COMM-02)
4. Create environment-based configuration for API endpoints
5. Implement request/response interceptors for authentication and error handling
6. Replace all setTimeout simulations with actual API calls

Files Affected: src/components/DashboardSection.tsx, new files: src/lib/api/, src/services/

3: Complete Absence of Test Coverage

Category: Test Debt

Related Requirements: All SR-UI-XX, SR-DATA-XX requirements (verification of compliance) **Related Design Components:** All DC-SW-XX components (verification of implementation)

Description: The codebase has zero test files. There are no unit tests, integration tests, or end-to-end tests. No testing framework is configured (Jest, Vitest, React Testing Library). This is particularly critical for AI-All

robot state (deployState, robotStatus, battery, location, cleaningProgress) is managed within a single DashboardSection component using local React state. There is no separation of concerns, no global state management solution (Redux, Zustand, Context API), and no clear API boundaries between UI components and business logic.generated code where "trust but verify" protocols are essential. Without tests, there is no verification mechanism to ensure AI-generated components function correctly, no regression protection, and no confidence in refactoring or adding new features.

Impact on Requirements:

Cannot verify compliance with SR-UI-01 (status notifications)

Cannot verify compliance with SR-UI-02 (real-time data transmission)

Cannot verify compliance with SR-DATA-01 (daily reports)

Cannot verify compliance with SR-DATA-02 (data export)

No regression protection when implementing user stories (AC 1.5, 2.3, 3.4, 3.5)

Remediation Plan:

1. Set up Vitest (recommended for Vite projects) or Jest with React Testing Library
2. Add test configuration files (vitest.config.ts or jest.config.js)
3. Create unit tests for all utility functions and hooks

4. Create component tests for critical UI components (DashboardSection, DeployButton, RobotStatusCard) 5. Implement integration tests for robot deployment flow
6. Add E2E tests using Playwright or Cypress for critical user journeys
7. Set up CI/CD pipeline to run tests on every commit
8. Establish minimum code coverage threshold (80% recommended)

Files Affected: New test files: src/**/*.test.tsx, src/**/*.spec.tsx, configuration files

4: Loose TypeScript Configuration Compromising Type Safety

Category: Architectural Debt

Related Requirements: All requirements (type safety affects all system components)
Related Design Components: All DC-XX components (type safety ensures correct API contracts)

Description: The TypeScript configuration has critical strictness flags disabled: noImplicitAny: false, strictNullChecks: false, noUnusedLocals: false, noUnusedParameters: false. This allows implicit any types, null/undefined errors, and unused variables, significantly reducing the benefits of TypeScript. This is a common issue in AI-generated codebases where strict type checking is disabled to avoid compilation errors, but it introduces runtime risk and makes the code harder to maintain.

Impact on Requirements:

Risk of runtime errors when implementing SR-UI-01, SR-UI-02 (API type mismatches) Risk of data corruption in SR-DATA-01, SR-DATA-02 (incorrect data types)
Difficult to maintain API contracts for DC-SW-02, DC-SW-03, DC-SW-04, DC-SW-05 **Remediation Plan:**

1. Enable strict TypeScript mode in tsconfig.json
2. Set noImplicitAny: true, strictNullChecks: true, strict: true

3. Enable noUnusedLocals: true and noUnusedParameters: true
4. Fix all resulting type errors incrementally
5. Add explicit type annotations where needed
6. Use proper null/undefined handling with optional chaining and nullish coalescing 7. Configure ESLint rules to catch TypeScript issues

Files Affected: tsconfig.json, all TypeScript files in src/

5: Missing Error Handling and Error Boundaries

Category: Architectural Debt

Related Requirements: SR-UI-01, SR-UI-02, AC 1.5, AC 2.3

Related Design Components: DC-COMM-01, DC-COMM-02 (communication failures), DC-SW-02, DC-SW-03 (API failures)

Description: The application has zero error handling mechanisms. There are no try-catch blocks, no error boundaries, no error states in components, and no retry logic for failed operations. When operations fail (e.g., deployment timeout, network errors), the application will crash or enter unrecoverable states. The

DashboardSection component's handleDeploy and handleStop functions have no error handling, and there are no error boundaries to catch React component errors.

Impact on Requirements:

SR-UI-01: Cannot provide reliable status notifications if communication fails (DC-COMM-01) **SR-UI-02:** Real-time data transmission fails silently without error handling (DC-COMM-02) **AC 1.5, 2.3:** Users cannot receive notifications if system crashes Robot control operations (deploy, stop) fail without recovery mechanisms

Remediation Plan:

1. Implement React Error Boundaries (src/components/ErrorBoundary.tsx)
2. Add try-catch blocks around all async operations
3. Create error state management in components
4. Implement retry logic with exponential backoff for API calls
5. Add user-friendly error messages and recovery actions
6. Set up error logging service (Sentry, LogRocket, or custom)
7. Add error handling to all state transitions in DashboardSection

Files Affected: src/components/DashboardSection.tsx, new: src/components/ErrorBoundary.tsx, src/lib/errorHandler.ts

6: No Authentication or Authorization System

Category: Architectural Debt

Related Requirements: SR-PWR-02 (docking station security), AC 2.5 (secure docking) **Related Design Components:** DC-HW-02 (Docking Station Physical Security)

Description: The application has no authentication mechanism. Anyone can access and control the robot

without login, token management, or permission checking. There is no user session management, no protected routes, and no role-based access control. This is a critical security issue for a production system controlling physical hardware, especially for beach security (AC 2.5) and university campus (AC 5.3) use cases.

Impact on Requirements:

> **SR-PWR-02:** Cannot enforce physical security without authentication (DC-HW-02) **AC 2.5:** Beach security officer use case requires authorized access only
> **AC 5.3:** University campus deployment requires role-based access control
> Unauthorized users can control robot, deploy it, or access sensitive location data

Remediation Plan:

1. Implement authentication service using Supabase Auth, Firebase Auth, or Auth0 2. Create protected route wrapper component
3. Add JWT token management and refresh logic
4. Implement role-based access control (admin, operator, viewer)
5. Add login/logout UI components
6. Secure API endpoints with authentication middleware
7. Add session persistence and auto-logout on token expiry

Files Affected: New: src/services/authService.ts, src/components/Auth/, src/App.tsx (route protection)

7: Hardcoded Values and Magic Strings Throughout Codebase

Category: Documentation Debt

Related Requirements: SR-PWR-01 (battery threshold 20%), AC 1.4 (auto-return on low power)
Related Design Components: DC-HW-01 (Battery Management System), DC-S-03 (Docking Beacon)

Description: The codebase contains numerous hardcoded values and magic strings that make the code difficult to read, maintain, and trace back to original requirements. Examples include: robot status strings ("idle", "cleaning", "returning"), location strings ("Home Base", "Zone A - North Side"), timing delays (2000ms, 3000ms), battery thresholds (20% - related to SR-PWR-01 and AC 1.4), and progress increments (3%). These values lack documentation explaining their origin or relationship to Agile requirements, making it unclear why specific values were chosen.

Impact on Requirements:

> **SR-PWR-01, AC 1.4:** Battery threshold (20%) is hardcoded without traceability to DC-HW-01 specifications
> **AC 2.2:** Geo-fencing boundaries not configurable (should relate to SR-GEO-01)
> **AC 3.2:** Solar charging thresholds not documented (should relate to SR-PWR-03)
> Cannot verify if timing values match actual robot hardware specifications

Remediation Plan:

1. Create a constants file (src/lib/constants.ts) with all magic values
2. Extract robot status types to enum or const object

3. Create configuration file for timing delays and thresholds
4. Add JSDoc comments explaining the source of each constant
5. Link constants to original user stories/requirements where possible
6. Make timing values configurable via environment variables
7. Add comments explaining business logic behind hardcoded values

Files Affected: src/components/DashboardSection.tsx, src/components/RobotStatusCard.tsx, new: src/lib/constants.ts, src/lib/config.ts

8: Lack of Component Documentation and Traceability

Category: Documentation Debt

Related Requirements: All user stories (AC 1.1-5.4) and system requirements (SR-XX) **Related Design Components:** All design components (DC-XX)

Description: Components lack JSDoc/TSDoc comments, making it difficult for humans to understand the purpose, props, and behavior of AI-generated components. There is no traceability back to original Agile requirements or user stories from the traceability matrix. The PROJECT-RESET-REPORT.md exists but is not integrated with the codebase, and individual components don't reference their requirements. This makes maintenance difficult and increases the risk of misunderstanding component intent.

Impact on Requirements:

Cannot trace DashboardSection to SR-UI-01, SR-UI-02

Cannot trace BatteryIndicator to SR-PWR-01, AC 1.4, DC-HW-01

Cannot trace RobotStatusCard to SR-UI-02, AC 2.3, DC-COMM-02

Cannot verify component compliance with traceability matrix requirements

New developers cannot understand which components fulfill which user stories

Remediation Plan:

1. Add TSDoc comments to all exported components and functions
2. Document component props, return types, and side effects
3. Link components to user stories in comments (e.g., @see UserStory-123)
4. Create component-level README files for complex components
5. Add inline comments explaining non-obvious logic
6. Generate API documentation using TypeDoc
7. Maintain a requirements traceability matrix

Files Affected: All component files in src/components/

9: Vendor Lock-in to Lovable.dev Platform

Category: Architectural Debt

Related Requirements: All requirements (platform dependency affects entire system) **Related Design Components:** All DC-XX components (deployment platform affects all components)

Description: The application is currently dependent on Lovable.dev platform for development and

deployment. The codebase includes lovable-tagger package dependency, Lovable-specific documentation in README.md, and references to Lovable.dev in meta tags. The application cannot be deployed independently or run as a standalone "real" application without Lovable.dev infrastructure. This creates vendor lock-in and prevents the application from being a production-ready, independent system.

Impact on Requirements:

All SR-XX Requirements: Cannot deploy independently, blocking all production requirements **All AC-XX User Stories:** Application cannot function as standalone system **Production Readiness:** Application is not a "real" app - it's a prototype tied to development platform **Deployment Flexibility:** Cannot deploy to Vercel, Netlify, AWS, or other independent platforms **Development Workflow:** Team members must use Lovable.dev to work on the project **Vendor Risk:** If Lovable.dev changes or discontinues service, entire project is at risk **CI/CD:** Cannot set up independent CI/CD pipelines **Professional Credibility:** Application appears as a prototype, not a production system **Remediation Plan:**

1. Remove lovable-tagger dependency from package.json devDependencies
2. Remove componentTagger() import and usage from vite.config.ts
3. Update package.json metadata:

   Change name from generic "vite_react_shadcn_ts" to "terra-sweep-sparkle"

   Add proper description, version, and project metadata
4. Rewrite README.md:

   Remove all Lovable.dev references and instructions

   Add standard npm/yarn installation instructions

   Add independent deployment instructions (Vercel, Netlify, etc.)

   Add project-specific documentation
5. Update index.html meta tags:

   Remove Lovable.dev OpenGraph images

   Remove Lovable Twitter references

   Add project-specific branding
6. Set up independent CI/CD:

   Create GitHub Actions workflows (.github/workflows/ci.yml, deploy.yml)

   Configure automated testing and deployment
7. Create deployment configuration:

   vercel.json or netlify.toml for platform deployment

   .env.example for environment variables

   Deployment documentation
8. Verify independence:

   Test build without Lovable dependencies

   Test deployment to independent platform

   Verify all functionality works post-decoupling

Files Affected:

package.json (remove lovable-tagger, update metadata)

vite.config.ts (remove componentTagger import and usage)

README.md (complete rewrite - remove Lovable references)

index.html (update meta tags)

.github/workflows/ (new - CI/CD pipelines)

vercel.json or netlify.toml (new - deployment config)

.env.example (new - environment variables)

AI & System Risk Assessment

Risk 1: AI Hallucination in State Management Logic

Category: Reliability/Hallucination

Related Requirements: SR-UI-01, SR-UI-02, SR-PWR-01, AC 1.4, AC 2.3

Related Design Components: DC-HW-01 (Battery Management System), DC-SW-02 (Status Reporting)

Description: The DashboardSection component contains complex state transition logic with multiple setTimeout calls and useEffect hooks that manage interdependent state updates. AI agents may have hallucinated the timing values (2000ms deploy, 3000ms return, 2s battery tick) without basis in actual robot hardware specifications from the Clean Bot Specification. The battery depletion simulation (linear -1% per 2 seconds) may not match real robot behavior, and the state machine logic (idle → deploying → deployed → cleaning → returning) may have race conditions or edge cases that AI didn't account for.

Impact:

Robot state may desync from actual hardware state (violates SR-UI-02)

Timing mismatches could cause UI to show incorrect robot status (violates SR-UI-01) Race conditions could leave robot in unrecoverable state

Battery simulation inaccuracies could mislead operators (violates SR-PWR-01, AC 1.4) Real-time location updates (AC 2.3) may be incorrect if state is desynced

Mitigation:

1. Verify all timing values against actual robot hardware specifications
2. Add comprehensive unit tests for state transitions
3. Implement state machine library (XState) to formalize transitions
4. Add integration tests with mock robot hardware
5. Implement state validation and recovery mechanisms
6. Add logging to track all state transitions for debugging

Files at Risk: src/components/DashboardSection.tsx

Risk 2: Prompt Injection and Data Leakage in Future AI Integration

Category: Security & Ethics

Related Requirements: SR-UI-02, SR-DATA-02, AC 2.3, AC 3.4, AC 3.5

Related Design Components: DC-COMM-02 (Real-time Telemetry), DC-SW-05 (Data Export API)

Description: While the current codebase doesn't directly process user prompts, the application is built on Lovable.dev (an AI development platform) and may integrate AI features in the future (e.g., natural language robot commands for AC 1.1, AI-powered scheduling for AC 1.2). Without proper input validation and sanitization, the system is vulnerable to prompt injection attacks where malicious users could inject

commands that manipulate AI behavior. Additionally, robot status data (location, battery, cleaning progress) is currently stored only in client-side state with no encryption or access controls, creating data leakage risks. This is particularly critical for beach security (AC 2.3) and environmental data (AC 3.4, 3.5) use cases.

Impact:

Malicious prompts could cause robot to perform unauthorized actions (violates AC 2.2 geo-fencing)
Sensitive location data could be exposed (violates SR-UI-02, AC 2.3)
Environmental data could be leaked (violates SR-DATA-02, AC 3.5)

User data could be leaked through unvalidated AI inputs

Compliance violations (GDPR, CCPA) if personal data is processed

Beach security location data (AC 2.3) could be compromised

Mitigation:

1. Implement input validation and sanitization for all user inputs
2. Add rate limiting for API endpoints
3. Implement Content Security Policy (CSP) headers
4. Encrypt sensitive robot data in transit and at rest
5. Add user input validation layer before any AI processing
6. Implement audit logging for all robot commands
7. Add role-based access control before AI feature integration

8. Conduct security audit before adding AI-powered features

Files at Risk: Future AI integration points, src/components/DashboardSection.tsx (data handling)

Risk 3: Dependency on Lovable.dev Platform and lovable-tagger ⚠ IN PROGRESS

Category: Dependency Risk

Related Requirements: All requirements (platform dependency affects entire system) **Related Design Components:** All DC-XX components (deployment platform affects all components)

Description: The application had a dependency on lovable-tagger package and was tightly coupled to Lovable.dev platform for deployment and development workflow. This created vendor lock-in and risked development workflow disruption if the platform changed or discontinued service.

Impact (Before Decoupling):

Development workflow disruption if lovable-tagger breaks (blocks all SR-UI-XX, SR-DATA-XX) Deployment pipeline failure if Lovable.dev platform changes (blocks production deployment) Inability to develop locally if platform dependencies fail (blocks AC 1.5, 2.3, 3.4, 3.5) Vendor lock-in preventing migration to other platforms
Potential build failures on future dependency updates

Cannot fulfill any system requirements if platform fails

Mitigation (Completed):

1. Remove lovable-tagger from vite.config.ts and package.json
2. Update package name from generic "vite_react_shadcn_ts" to "terra-sweep-sparkle"
3. Rewrite README.md with project-specific documentation (remove Lovable

references) 4. Update meta tags in index.html (remove Lovable branding)
5. Set up independent CI/CD pipeline (GitHub Actions)
6. Create deployment configuration for Vercel/Netlify
7. Document independent deployment process

Remaining Tasks:

Set up GitHub Actions workflow for CI/CD
Create deployment configuration files
Test deployment on independent platform
Update documentation with deployment instructions

Files at Risk: vite.config.ts, package.json, deployment configuration

Risk 4: AI-Generated Component Complexity Without Verification

Category: Reliability/Hallucination

Related Requirements: SR-UI-01, SR-UI-02 (dashboard components)

Related Design Components: DC-SW-02, DC-SW-03 (dashboard implementation)

Description: The codebase contains 49 UI components in src/components/ui/, but the PROJECT-RESET REPORT indicates most are unused. AI may have generated these components anticipating future needs, but without verification, some components may have bugs, accessibility issues, or incorrect implementations. The DashboardSection component uses complex useEffect logic with intervals that could have memory leaks or incorrect cleanup. Without tests, these AI-generated components cannot be verified for correctness. This is critical since the dashboard is the primary interface for SR-UI-01 and SR-UI-02 requirements.

Impact:

Unused components add to bundle size unnecessarily (affects performance for AC 1.5, 2.3)
Components may have hidden bugs that surface when used (violates SR-UI-01)
Memory leaks in useEffect hooks could degrade performance (affects real-time updates
SR-UI-02) Accessibility issues in UI components could violate compliance (affects all user stories)
Cannot verify dashboard components fulfill SR-UI-01, SR-UI-02 requirements

Mitigation:

1. Audit all UI components for actual usage
2. Remove or tree-shake unused components
3. Add tests for all used components
4. Review useEffect dependencies and cleanup logic
5. Run bundle analysis to identify unused code
6. Add ESLint rules to catch common React pitfalls
7. Conduct accessibility audit (axe-core, WAVE)

Files at Risk: src/components/ui/* (49 components), src/components/DashboardSection.tsx

Risk 5: Missing Input Validation and Sanitization

Category: Security & Ethics

Related Requirements: SR-GEO-01, SR-GEO-02, SR-AUTO-01, AC 1.2, AC 2.2, AC 4.4 **Related Design Components:** DC-A-02 (Geo-Fencing Algorithm), DC-A-04 (Exclusion Zone Mapping), DC SW-01 (Scheduler)

Description: While the current application has limited user input, the DashboardSection component directly manipulates state without validation. Future features may accept user input (scheduling for AC 1.2, zone selection for AC 2.2, exclusion zones for AC 4.4), and without a validation layer, the system is vulnerable to injection attacks, XSS, and invalid data causing state corruption. The application uses React but doesn't leverage Zod (already in dependencies) for runtime validation. This is critical for geo-fencing (SR-GEO-01, SR GEO-02) and scheduling (SR-AUTO-01) features.

Impact:

- Invalid input could corrupt robot state (violates SR-UI-02)
- XSS vulnerabilities if user input is rendered (affects all SR-UI-XX)
- Injection attacks if input is passed to backend/robot (violates AC 2.2 geo-fencing)
- State corruption leading to robot malfunction
- Invalid geo-fence coordinates could cause robot to leave boundaries (violates SR-GEO-01, AC 2.2)
- Invalid schedule data could cause robot to operate at wrong times (violates SR-AUTO-01, AC 1.2)

Mitigation:

1. Implement Zod schemas for all user inputs
2. Add input validation layer before state updates
3. Sanitize all user-provided strings
4. Use React Hook Form with Zod resolver (already available) 5. Add server-side validation when backend is implemented 6. Implement Content Security Policy
7. Add rate limiting for user actions

Files at Risk: All components accepting user input, future form components

Backlog Items

Refactor Monolithic State Management (IN PROGRESS)

- Set up global state management
- Extract state logic from DashboardSection component
- Refactor DashboardSection component
- Add state persistence

Remove Lovable Platform Dependencies (IN PROGRESS)

- Remove Lovable dependencies in codebase
- Update package metadata
- Rewrite README
- Update branding and meta tags
- Set up independent CI/CD pipeline

- Create deployment configuration
- Verify independence from Lovable

Set up Testing Infrastucture (IN PROGRESS)

- Install and configure testing framework
- Create test configuration
- Create test file structure
- Configure CI/CD pipeline for testing