Cloudflare Feedback Agent

Zia Yan
Date: 2026-01-20

Feedback Agent is a lightweight internal tool that turns raw user feedback into structured labels (summary / sentiment / category) using Workers AI, and stores them in D1 for product teams to review and export.

Project Links

Deployed prototype (Workers)	https://feedback-agent.zia-feedback-agent.workers.dev
GitHub repository	https://github.com/ziayan-analytics/feedback-agent

Cloudflare Product Insights (Friction Points)

- Local vs Remote D1 Database Confusion
- Bindings Configuration Not Beginner-Friendly
- Workers AI Output Format Variability (JSON Stability Issues)
- Missing Post-Deploy Validation Workflow

1) Cloudflare Product Insights (Friction Log)

1. Local vs Remote D1 Database Confusion

Problem:

During prototyping, it's easy to confuse the local D1 environment (Wrangler dev) with the remote production D1 database. Commands like wrangler d1 execute behave differently depending on whether --remote is used, and the workflow is sensitive to a small flag without strong guardrails. This commonly leads to:

- Inserting seed data locally but deploying a Worker connected to an empty remote database
- Running schema creation/migrations against the wrong environment
- Debugging "missing table" issues that only occur in production

Impact:

Wasted ~15 minutes debugging an environment mismatch (not reproducible locally), slowing down iteration and increasing deployment risk.

Suggestion:

Add clearer guardrails so users always know which environment they are modifying.

- Dashboard UI: Show a prominent LOCAL vs PRODUCTION badge in the D1 database console
- CLI output: Print a clear banner before executing commands, e.g.
    i. EXECUTING ON: LOCAL D1 DATABASE
    ii. EXECUTING ON: REMOTE D1 DATABASE (production)
- Guided setup flow: Provide a wizard-style setup (e.g. wrangler d1 init) that:
    i. Creates schema in both local + remote environments
    ii. Prints exact commands to test each environment
    iii. Optionally shows a diff between local vs remote schema/state

2. Bindings Configuration Not Beginner-Friendly

Problem:

Adding Workers AI and D1 bindings required manually editing wrangler.jsonc, then running npx wrangler types to regenerate TypeScript runtime types. For first-time users, the correct sequence is not intuitive:

- Add bindings in wrangler.jsonc (D1 + AI)
- Run npx wrangler types to generate runtime types
- Restart the TypeScript server (IDE) to pick up new types
- Update code to use env.AI and env.feedback_db

Missing any step often results in TypeScript errors that don't clearly indicate the root cause (missing bindings / stale types).

Impact:

Spent ~10 minutes troubleshooting TypeScript errors before realizing the types needed to be regenerated.

Suggestion:

- Dashboard "Copy binding config": Provide a copy-ready snippet that includes both binding config and the next command (e.g. npx wrangler types)
- Auto-type generation: Consider running wrangler types automatically during wrangler dev / wrangler deploy when bindings change
- Better error messaging: When bindings are missing or types are stale, suggest running npx wrangler types

3. Workers AI Output Format Variability

Problem:

When calling Workers AI to label user feedback, the output format was inconsistent without strong constraints. Early iterations produced:

- Inconsistent field names (e.g., category vs type vs classification)
- Different casing (Positive vs positive)
- Extra fields not requested (urgency, priority)
- Occasionally plain text instead of JSON

This required multiple prompt iterations and defensive parsing to ensure stable, machine-readable output.

Impact:

Spent ~20 minutes refining prompts and adding JSON parsing fallbacks to reliably store results into D1.

Suggestion:

- Structured output mode: Provide an official "JSON schema constrained output" interface (or built-in schema validation helper) so developers can guarantee parseable output.
- Template library: Add pre-built prompt templates for common patterns like product feedback labeling:
  i. templates.classifyFeedback(text)
  ii. templates.extractSentiment(text)
  iii. templates.summarizeThread(messages)
- More end-to-end docs: Add a "Product Feedback Analysis" recipe in Workers AI documentation (AI → label → store in D1 → UI display).

4. Missing Post-Deploy Validation Workflow

Problem:

After running npx wrangler deploy, the Worker URL becomes immediately available, but there is no guided validation step to confirm that the deployment is truly production-ready, such as:

- Production D1 schema exists / tables are created
- Bindings are correctly connected (D1 / Workers AI)
- Key API endpoints return expected responses

This can result in a "successful deployment" that still fails on first request (e.g., a 500 error because the D1 table exists locally but not in production).

Impact:

Increased risk of shipping a broken production endpoint and spending additional time debugging issues that could be caught automatically. This is especially confusing for first-time users.

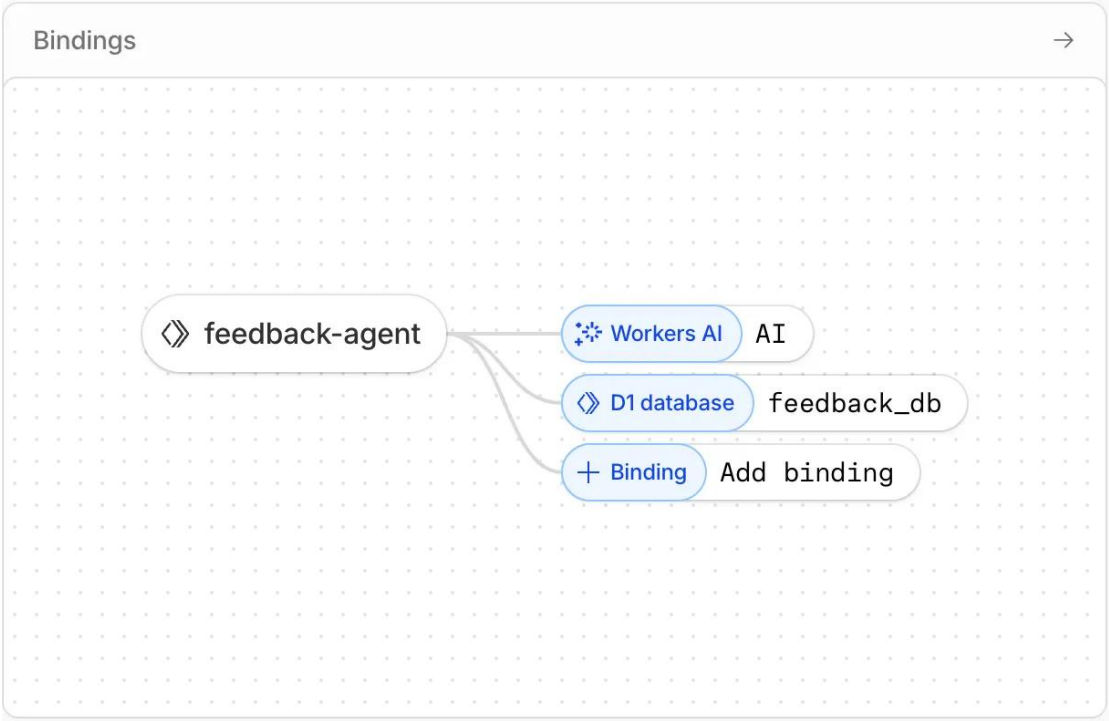
Suggestion:

- Post-deploy checks: After wrangler deploy, provide an optional guided validation flow that runs:
    i. endpoint ping (GET /)
    ii. basic D1 query (e.g., SELECT 1)
    iii. binding connectivity checks
    iv. basic API smoke test (GET /api/feedback)
- Smoke test template: Offer an option to auto-generate a smoke-test.sh (or wrangler smoke-test) script that can be rerun anytime to validate production health.

2) Architecture Overview

This prototype is built as a lightweight “feedback labeling + storage” pipeline using Cloudflare Workers as the core runtime. The architecture consists of four main components: a static frontend UI, a Worker backend API, Workers AI for labeling, and D1 for persistence.

Workers Bindings Screenshot



What this app does

Feedback Agent is an internal tool that helps teams quickly turn raw user feedback into structured, analyzable data. Team members paste one feedback message into the UI, and the Worker automatically calls Cloudflare Workers AI to generate a summary, sentiment, and category label, then stores everything in Cloudflare D1. The latest labeled feedback can be viewed in the UI or fetched via API, making D1 a simple prototype dataset for downstream analytics.

Components

1. Frontend UI (Static Assets on Workers)

- The user interacts with a simple webpage (public/index.html) to submit feedback text and view the latest feedback list.
- The UI is deployed as static assets served directly by the Worker.

2. Cloudflare Worker (API + Orchestration Layer)

- The Worker exposes REST endpoints:
    i. POST /api/feedback: accepts raw feedback text
    ii. GET /api/feedback: returns the latest feedback records
- It serves as the orchestration layer that connects the UI, AI model, and database.

3. Workers AI (Automatic Feedback Labeling)

- When feedback is submitted, the Worker calls Workers AI to generate structured metadata:
    i. summary
    ii. sentiment (positive / neutral / negative)
    iii. category (Bug / Feature Request / Praise / Other)
- This removes the need for manual tagging and standardizes feedback classification.

4. Cloudflare D1 Database (Persistent Storage)

- The Worker stores both the original feedback text and AI-generated labels into a D1 (SQLite) database.
- This enables the system to support historical feedback tracking and makes the data exportable for analytics workflows.

End-to-End Data Flow

- User submits feedback in the UI
- UI sends request to the Worker (POST /api/feedback)
- Worker calls Workers AI to generate labels
- Worker inserts feedback + labels into D1
- UI refreshes by calling GET /api/feedback and displays the latest feedback list

Cloudflare Bindings

This project uses Workers Bindings to connect the Worker runtime to external Cloudflare services:

- AI → Workers AI binding
- feedback_db → D1 Database binding

Why this architecture

This architecture keeps the prototype minimal while demonstrating an end-to-end Cloudflare-native workflow: UI hosting, serverless API, AI inference, and persistent storage — all deployed and managed within Cloudflare's developer platform.

3) Vibe-coding Context

This prototype was built using a vibe-coding workflow inside VS Code with AI coding assistance. I used AI mainly for:

- Quickly scaffolding a Cloudflare Workers + D1 project and validating Wrangler configuration
- Iterating on prompt design to make Workers AI output stable, parseable JSON
- Debugging deployment issues (local vs remote D1 confusion, missing schema in production)
- Improving documentation and adding a step-by-step setup guide

Tools used

- VS Code
- Claude Code (VS Code AI assistant)
- Cloudflare tooling: create-cloudflare, wrangler dev, wrangler deploy, wrangler d1 execute, wrangler types

Example prompts used

- “Generate a Cloudflare Worker API endpoint that writes to D1 database.”
- “Update the Worker to call Workers AI and return structured JSON with summary/sentiment/category.”
- “Make Workers AI output consistent JSON. Add prompt constraints and handle parsing errors.”
- “Help me create a README + setup guide explaining every step from CLI init to production deployment.”
- “Troubleshoot why D1 works locally but returns ‘missing table’ in production.”

This workflow helped me move faster by reducing time spent on boilerplate code, debugging deployment configuration, and documentation writing, while still keeping manual control over architecture and implementation details.