Cloudflare PM Internship Assignment Submission

Candidate: Bhavana Bafna
Email: bb3292@columbia.edu
Mobile: +1 646 506 7956

Project Links:

1. **Cloudflare Workers:** https://signalflow-bhavana.bhavana.workers.dev/

2. **Github:** https://github.com/bb3292/feedback-analyzer-cloudflare

Cloudflare Product Insights:

Throughout this exercise, I used Cloudflare products to build my prototype and experienced the platform as a customer would. Below is my friction log documenting the onboarding, setup experience, documentation quality, and UI/UX observations.

1. Silent Deployment Failures with Incompatible Runtime Versions

Title: Silent Deployment Failures with Incompatible Runtime Versions

Problem: When attempting to deploy using Wrangler with Node.js version 24, the deployment failed without providing any error messages or warnings indicating the root cause. This resulted in over 30 minutes of troubleshooting, investigating code errors and configuration issues, when the actual problem was simply an unsupported runtime version. The lack of feedback made it extremely difficult to diagnose and slowed down the development process significantly.

Suggestion: As a PM, I would implement explicit runtime version validation before initiating deployment. The system should check the Node.js version and display a clear, actionable error message such as "Error: Node.js 24 is not currently supported. Please use Node.js 18 or 20 to deploy your Worker." This validation should occur in both the `deploy` and `dev` commands, preventing wasted time on incompatible environments.

2. Language Diversity Gap in Documentation and Examples

Title: Language Diversity Gap in Documentation and Examples

Problem: The documentation exhibits a strong bias toward JavaScript, with minimal examples or guidance for developers using Python, Go, or Rust. When examples for alternative languages do exist, they tend to be overly simplistic and don't demonstrate practical use cases involving

bindings or complex workflows. This creates a barrier for developers who prefer or require non-JavaScript solutions, forcing them to translate concepts and potentially miss language-specific nuances.

Suggestion: Introduce tabbed code examples throughout the documentation that show identical functionality implemented in JavaScript, Python, and Go side-by-side. Additionally, create dedicated language-specific quick-start guides and establish a "Polyglot Examples" section showcasing common patterns across all supported languages. This would make the platform more accessible to a broader developer audience and reduce the learning curve for non-JavaScript developers.

3. Fragmented Documentation Architecture with Excessive Cross-References

Title: Fragmented Documentation Architecture with Excessive Cross-References

Problem: The documentation frequently redirects users to other internal pages mid-explanation, creating a disjointed learning experience. Following these cross-references often leads to a "rabbit hole" effect where users navigate through multiple pages and lose the original context. This fragmentation makes it difficult to complete tasks without maintaining multiple browser tabs and mentally piecing together information from disparate sources.

Suggestion: Consolidate critical information into comprehensive, self-contained "Getting Started" guides that include all necessary steps on a single page. Replace excessive external links with expandable accordion sections that reveal additional details inline without navigation. Implement clear breadcrumb navigation and contextual "back to guide" links to help users maintain their place in the learning journey.

4. Non-Actionable Error Messages in Wrangler CLI

Title: Non-Actionable Error Messages in Wrangler CLI

Problem: The Wrangler CLI often returns generic error messages such as "execution failed" without indicating whether the issue stems from syntax errors, permission problems, network connectivity, or configuration mistakes. This lack of specificity makes troubleshooting extremely difficult, as developers cannot distinguish between transient issues that might resolve on retry versus persistent configuration problems that require code changes.

Suggestion: Implement a structured error code system with specific, actionable resolution steps for each error type. Error output should include contextual help, common solutions, and relevant documentation links. For example, instead of "execution failed," the message could read "Error E2041: Deployment failed due to invalid binding reference. Check that 'MY_KV' exists in your account and the ID in wrangler.toml matches. See: [link to binding documentation]."

5. Manual Resource ID Management Across Accounts

Title: Manual Resource ID Management Across Accounts

Problem: When migrating projects between Cloudflare accounts or recreating resources, developers must manually locate and update multiple resource IDs in the wrangler.toml configuration file. There's no automated mechanism to synchronize these IDs or validate that they're correct before deployment, leading to failed deployments and time spent cross-referencing the dashboard with configuration files.

Suggestion: Introduce a `wrangler sync` command that automatically detects and updates resource IDs based on the currently authenticated account. Additionally, implement pre-deployment resource validation that checks whether referenced bindings actually exist and displays clear warnings about any mismatches, along with suggestions for resolution. A `wrangler validate` command could also be useful for checking configuration health before deployment.

Architecture Overview:

This prototype leverages multiple Cloudflare products to create an integrated, edge-native feedback collection and analysis system. Below is an explanation of each product used and the rationale behind these architectural decisions.

Cloudflare Products Used:

Cloudflare Workers

Purpose: Serves as the primary compute layer, hosting both API endpoints and the dashboard user interface within a single Worker.

Why this product: After exploring Cloudflare's platform, Workers stood out as the ideal foundation for this prototype. The concept of deploying code globally across Cloudflare's edge network means users experience low latency regardless of their location, something traditional server-based architectures struggle with. The serverless model also removes the complexity of managing infrastructure or worrying about capacity planning, which is perfect for a prototype that needs to demonstrate value quickly. I consolidated all routing, data processing, and UI serving into one Worker to keep the architecture simple while still benefiting from edge performance. This felt like the right balance between simplicity and production-readiness.

D1 Database

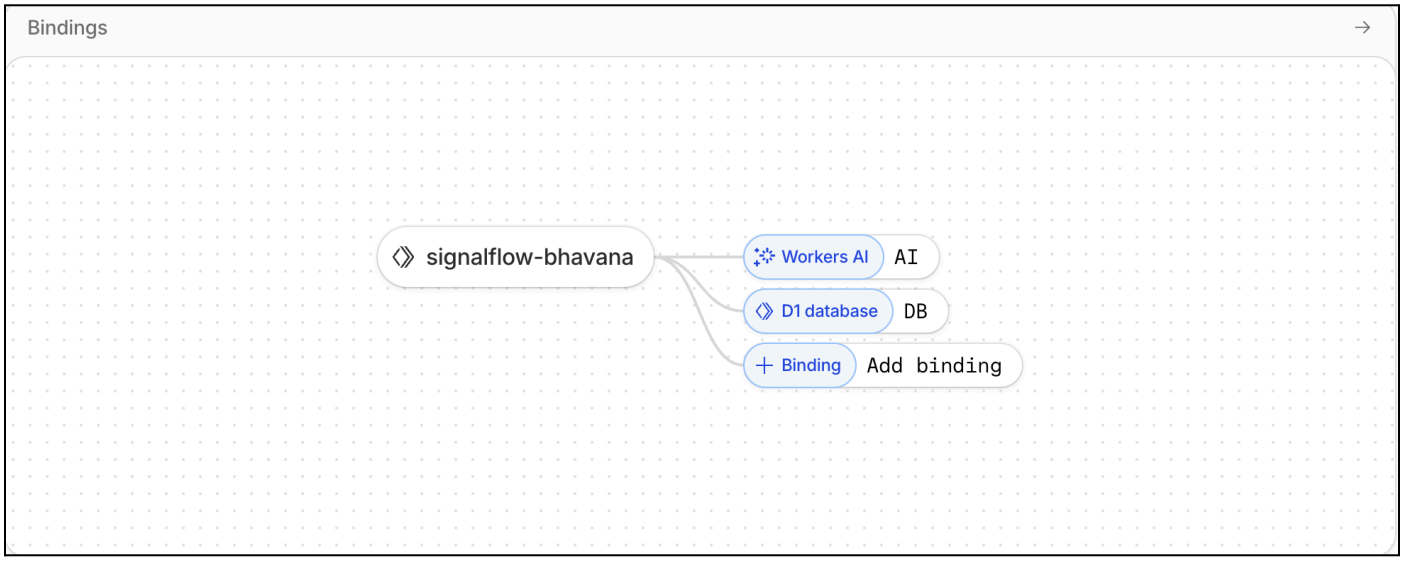
Purpose: Provides persistent data storage for feedback entries and aggregated analytics metrics.

Why this product: When evaluating storage options, D1's SQL capabilities made it immediately familiar and accessible. Being able to write SQL queries while getting automatic scaling and backups felt like the best of both worlds. I designed a two-table schema: a `feedback` table for raw submissions and a `daily_metrics` table for pre-aggregated analytics. This structure enables efficient dashboard queries without repeatedly processing raw data. D1's SQLite compatibility also made local development and testing straightforward, which accelerated the prototyping process significantly.

Workers AI

Purpose: Powers the intelligence layer by performing real-time sentiment analysis and theme extraction on submitted feedback.

Why this product: Initially, I considered using external AI APIs, but Workers AI's integration caught my attention during exploration. Running AI models directly at the edge eliminates the latency and complexity of external API calls, and keeps sensitive feedback data within Cloudflare's network—an important consideration for enterprise users. The `@cf/meta/llama-3-8b-instruct` model proved capable enough for sentiment analysis and theme extraction without needing fine-tuning. What really impressed me was how seamlessly it integrated with the Worker runtime, no authentication juggling or rate limit concerns. This felt like a genuine advantage of building within the Cloudflare ecosystem.



Vibe-Coding Context:

Platform Used

Primary Development Environment: Cursor IDE with **claude-4.5-opus-high** integration

Development Approach

The prototype was built using an iterative, AI-assisted development workflow that leveraged conversational prompting to rapidly translate product requirements into functional code. This approach proved particularly valuable for navigating Cloudflare's ecosystem and implementing complex features efficiently.

Key Prompts Used

Prompt 1

"Using the Cloudflare platform, create a prototype tool that helps aggregate and analyze feedback so you can derive meaningful results from the feedback that you receive.

As a product manager building a tool for product managers, the emphasis is on dimensions such as sentiment analysis, theme detection, urgency, and value (feedback from power users) of the feedback in a manner that is actionable for anyone using the tool.
Use Cloudflare Worker AI leveraging LLM(like llama-3-8b-instruct) for sentiment analysis, theme detection, urgency, and value determination.

Create a dashboard that shows detailed graphs for each of the dimensions.

Create a feature to drill down into individual themes, with details like daily sentiment analysis trend for that theme, the number of feedback under that theme, split at channel (like discord, git etc.), etc.

Sentiment analysis: Use natural language processing (NLP) to score each feedback item as positive, neutral, or negative (and possibly intensity). Leverage Worker AI (llama-3-8b-instruct) cloudflare development product, for achieving this task. Use sentiment trends as a gauge (e.g., a drop in sentiment after a release). Also watch for specific urgency cues ("urgent", "now", "before next week") by keyword or model.

Create slicing and dicing for filters. Add filter criteria for each of the dimensions.

The feedback will come daily. Given that worker AI will aggregate and analyse the feedback daily, store the aggregate responses in a separate database table to power the dashboard view. For this purpose, use D1, which is a Cloudflare development product offering for MySQL databases. Add relevant indexes to power search and filters."

Prompt 2

“Thanks for this, I have a few update requests as below:

1. *Categorize the sentiment as positive, negative, or neutral. Better if you can think further about it.*
2. *For each of the dimensions: sentiment, urgency, value, theme, let's add a pie chart showing the percentage of each sub-division, like a pie chart showing what % of feedbacks are positive, what % is negative, and what % is neutral, similarly for other dimensions.*
3. *Add a section for the top theme as well.*
4. *Given that you have provided a section for “Themes needing attention”, where each of the themes is clickable. Upon clicking, a further drill-down of that theme is displayed. I want a sentiment trend based on daily feedback for each of the themes, as well as in the same window that opens on clicking individual themes. This trend will help product managers to analyse the sentiment of the customers against a particular theme in addition to the overall product.*
5. *Under the section for “Themes needing attention”, when each theme is clicked, some sample feedback is presented. In addition to that, provide a summary of recent feedback.*
6. *A bar chart for the priority/urgency breakdown.”*

Prompt 3

“Under the section for “Themes needing attention”, when each theme is clicked, rather than showing the sentiment analysis trend by plotting the absolute count of positive/negative/neutral feedback, plot % of each against the total volume of feedback under that theme. Also plot Net Sentiment score with the formula %positive -%negative.”