Task 2.3 – Code Review

We used two review approaches. First, we did a manual review by reading the description in generated in `README.md` and then walking through the backend (`server.js`), the frontend (`public/app.jsx`), the styles (`public/styles.css`) and the persisted data (`data/logs.json`). We tried to follow the same path a user follows: fill in the form, hit the API routes, persist to JSON, and show the declaration.

Second, we used AI-assisted analysis with GitHub Copilot chat (GPT-5.3-Codex) to challenge our own findings. We asked for a broad scan across maintainability, reliability, performance, and security, and then a focused pass on frontend state and data flow. We treated the AI output as suggestions, not as the answer, and we decided explicitly what to accept and what to defer.

Code smells we found

Manual review

1) Mixed responsibilities in the backend

A "god file" smell shows up in the backend: `server.js` is doing a bit of everything—app configuration, validation rules, compliance logic, persistence logic, all routes, and error handling—in a single place. It works, but as the file grows, safe changes get harder and route logic is more likely to be duplicated or drift out of sync.

2) Frontend root component is carrying too much

On the frontend, the root component in `public/app.jsx` is taking on too many roles at once: API client code, global state, mutation logic, and rendering all live in the same module and component. It's still readable now, but this structure makes accidental coupling easier and makes the codebase harder to test, extend, or refactor cleanly.

3) Reflection cannot be cleared

The reflection update endpoint rejects an empty string, which means users can add or change a reflection but can't remove it. This is a small UX rough edge, but it also makes it harder to correct mistakes cleanly.

AI-assisted review

4) Duplicated request and error handling logic

Several API helpers repeat the same `fetch` + `res.ok` handling + JSON parsing pattern. That repetition invites small inconsistencies and makes even minor future changes feel larger than they need to be.

5) JSON persistence has a race condition risk

The persistence approach follows a read–modify–write pattern on a shared JSON file without locking or a queue. With concurrent requests, one write can overwrite another—exactly the kind of data-loss bug that rarely shows up in light testing, but becomes real under parallel usage.

6) Magic numbers and scattered rules

Limits and banned terms are hard-coded in multiple places instead of being centralized. When requirements change, this makes it easier for behavior to drift across routes and for "almost the same" rules to quietly diverge.

7) Validation differs between create and update

The same reflection field is validated differently depending on endpoint: `POST /api/logs` applies stricter limits than `PATCH /api/logs/:id/reflection`. From a user perspective, it's confusing, and from a maintenance perspective, it's an easy way to end up with inconsistent data.

8) Client-controlled timestamp weakens audit value

In the original Task 2.2 implementation, the server accepts a client-supplied timestamp rather than always generating one server-side. For audit-style logging, trusting the client for an audit field reduces confidence in the integrity of the records.

9) CSP disabled while loading runtime scripts from CDNs

CSP is disabled while the UI loads React and Babel at runtime from CDNs. That's a reasonable prototype shortcut, but it weakens the security posture and increases exposure to injection and supply-chain risks compared to bundling and enforcing a restrictive CSP.

AI-assisted prompts and responses

Prompt 1, structural smell scan:

> Analyze `server.js`, `public/app.jsx`, and the HTML entry point that loads frontend scripts for structural code smells that affect maintainability, reliability, and security.
>
> Return 8 to 12 findings.
>
> For each finding, include severity, a short evidence snippet, and a fix suggestion.
>
> Also flag a few items that may be over engineering for the assignment scope.

Summary of response:

The response repeatedly surfaced duplication in request handling on the client side and suggested a shared helper to standardize `fetch`, error checks, and JSON parsing. It also highlighted reliability risks in the JSON persistence pattern,

where concurrent requests can overwrite each other. Several findings were about rules that are scattered across the backend, such as hard-coded limits and banned terms, plus validation that differs between endpoints for the same field. On the security side, it noted that client supplied timestamps weaken the audit value and that CSP is disabled while runtime scripts are loaded from CDNs.

Prompt 2, frontend data flow and responsibility split:

> Review `public/app.jsx` and return exactly 5 maintainability findings focused on data flow and separation of concerns. Include evidence snippets and refactors that fit a small assignment scope.

Summary of response:

The response emphasized that the root component mixes orchestration and UI, which increases coupling and makes testing harder. It also recommended pulling request and error handling into a small API module and reducing the amount of mutable global state handled directly in the root.

What we accepted and what we deferred

The AI output was treated as a second opinion, not a decision maker. The changes we accepted were the ones that reduce risk without expanding the scope too much. That includes de duplicating request and error handling, splitting frontend responsibilities into smaller pieces, aligning validation rules across routes, centralizing shared limits and rules, and generating timestamps on the server.

We deferred suggestions that would turn the prototype into a production system. Moving from a JSON file to a database and adding full authentication and authorization would be valuable in a real deployment, but the cost is high for this assignment. We also deferred the larger packaging and hardening work needed to stop loading runtime scripts from CDNs and to enforce a strict CSP, while still documenting the tradeoff clearly.

What we would fix first

The project meets the intended feature slice and is easy to run and understand. The remaining issues are mostly structural and prototype level, especially around persistence, validation consistency, and how responsibilities are grouped in the frontend.

With limited time, the first priority would be reliability through write serialization and consistent validation. Next would be maintainability through shared request helpers, centralized rules, and smaller frontend modules. Finally, security hardening would focus on server owned timestamps and reducing exposure from disabled CSP and CDN loaded runtime scripts.