# Professional Self-Assessment

Project: **Arrow Grain Calculator**

## Overview

Across the Computer Science program I grew from writing small, single-file scripts to designing, testing, and shipping a full web application. My capstone, the Arrow Grain Calculator, shows that trajectory: a React front end with an interactive SVG arrow (software design & engineering), algorithmic calculations for shaft grains and FOC% (algorithms & data structures), and a Node/Express + MongoDB layer to persist builds (databases). Each layer was refined to meet professional expectations for clarity, maintainability, and user value.

## Collaboration and Decision Making

I used practices that create a collaborative environment even on a solo project: a PR-first workflow, clear commit messages, and issue tracking for increments. Architectural decisions are documented (client vs. server calculations, schema constraints, and UX choices) to make the system understandable and reviewable. Iterations were guided by feedback—for example, improving hover states, grouped selection, and click-away behavior to reduce user confusion.

## Communication

I aimed for professional-quality communication across media: (1) visual, consistent motion/feedback such as hover lift and glow; (2) written, descriptive component/file names and comments where intent isn’t obvious; (3) oral, a code‑review video that explains architecture, trade‑offs, and enhancement rationale for a technical audience.

## Software Engineering and Databases

The front end uses modular React components (e.g., ArrowSVG) and predictable state via hooks. I emphasized cohesion by keeping calculator logic near the form and isolating visuals inside the SVG component. On the back end I designed an Express API with a Mongoose model that validates domain rules (component enums, non‑negative grains, timestamps). The UI now includes a collapsible, paginated Saved Builds panel (10 per page) to scale with data.

## Algorithms and Data Structures

Shaft grains are computed as GPI × arrow length; total grains sums component weights with validation; FOC% uses a weighted balance‑point formula across components (tip, insert, shaft, fletching, knock). I favor clarity and testability over premature optimization, and I recompute totals on the server to preserve integrity.

## Security Mindset

I added input validation and schema constraints, recomputed critical values server‑side, and kept a roadmap for future hardening: request validation (Joi/Zod), authentication/authorization for private builds, rate‑limiting, and sanitization to guard against injection. These measures anticipate adversarial behavior and protect data and resources.

## How the Artifacts Fit Together

The three enhancements, UI/UX, algorithms, and database, form a coherent, job‑ready story: I can design an experience, implement the logic behind it, and ship the data layer to support real users. This end‑to‑end capability demonstrates mastery of the program outcomes and my readiness for professional software development.

## Next Steps

Short‑term: add unit tests for FOC% and server endpoints, improve accessibility (keyboard focus and ARIA on SVG parts), and add optimistic UI for saves. Medium‑term: implement authentication and per‑user builds, add shareable links, and deploy CI with lint/test gates. Long‑term: analyze anonymized build data to recommend components and visualize performance trends.