COLORADOSCHOOLOFMINES
EARTH • ENERGY • ENVIRONMENT

CSCI 370 Final Report

Version Sentinels

Colin Wolff
Daniel Castelluccio
Joe Steele

Jun 13, 2025

CSCI 370 Summer 2025

Scott Jensen

Table 1: Revision history

Revision	Date	Comments
New	May 18, 2025	Completed Sections: I. Introduction II. Functional Requirements III. Non-functional Requirements IV. Risks V. Definition of Done
Rev - 2	May 25, 2025	Completed Sections: VI. System Architecture
Rev – 3	Jun 1, 2025	Completed Sections: VII. Software Test and Quality VIII. Project Ethical Considerations
Rev – 4	Jun 8, 2025	Completed Sections: IX. Project Completion Status X. Future Work XI. Lessons Learned
Rev – 5	Jun 10, 2025	Completed Sections: XII. Acknowledgements XIII. Team Profile XIV. Appendix A: Key Terms
Rev – 6	Jun 13, 2025	Revised all sections according to feedback.

Table of Contents

I. Introduction

ICR is a contractor for the intelligence and defense communities. A lot of the projects they work on, like most professional software projects, use some sort of API to interface with other systems hosted both internally and externally. Additionally, some of these APIs have frequent, significant changes. As a result, they need a way to efficiently understand these API changes, to better equip them to respond to an API update that breaks their code. The general idea of this project is to create a tool to make it easier to track and understand API changes, using automatically generated documentation (OpenAPI specifications) for each API version. The software needs to track APIs from the API consumer paradigm and represent changes in some kind of user interface to make them as easy to understand as possible. The main target of this project is developers as they are the ones directly interacting with APIs and depending on specific characteristics of them. Following this field session, this software would be maintained by an employee at ICR.

Types of API Objects that can change:

- Endpoints
- Methods
- Parameters
- Responses
- Schemas

Types of API Object changes:

- Add
- Remove
- Move
- Change

II. Functional Requirements

This project has two essential components, a backend which will scrape data from automatically generated documentation, and a frontend which takes the data collected by the backend and uses it to display differences between API versions.

Frontend:

- A way for the user to add APIs to be tracked to a list of currently tracked APIs.
- A way to select two versions for comparison and visually show the difference between them.
- A way to view any version of an API from the point it starts being tracked onwards.
- A way to visualize how the API changed over time at a higher level, with significant changes made apparent.

Backend:

- Periodically scrapes API data from sources specified from the front end, on a set schedule of 6 hours, as recommended by our client. This schedule ensures changes throughout the day are caught while not querying the API excessively.
- Performs a hash signature check on the scraped version of the API with the most recent stored version of the API. If it has been updated, store the scraped version in a database.
- Calculates and generates a diff tree object for the frontend to display based on comparison between two user-selected versions.
- Has an interface to allow the frontend to get data for each API version, using API endpoints.

III. Non-Functional Requirements

- Needs to use Angular 17 as this is how it will be maintained in the future.
- Backend scraping needs to support Swagger 3.0 Specs.

- The visualization methods the frontend uses need to effectively communicate changes between two API versions.

IV. Risks

Technology Risks:

- **Unlikely, Major:** Swagger Pages may not follow the exact format that our backend expects, can mitigate this by ensuring the system follows a standard and testing is done for a wide range of APIs
- **Unlikely, Moderate:** Database data may be lost, resulting in previously saved API version data no longer accessible

Skills Risks:

- **Minor:** Only one of our team members has experience with PostgreSQL
- **Moderate:** None of our team members have significant experience with frontend development, specifically Angular
- **Minor:** Only one of our team members has significant experience working with APIs, however none have experience with Swagger Pages

In all cases for skills risks, the team members with less or no experience are spending time learning these unfamiliar technologies.

V. Definition of Done

- The application can be run locally with minimal setup.
- Able to collect and store data about individual API versions.
- Two API versions can be compared, with differences being clear.
- Able to display general API changes over time.
- Application can be used through a GUI.
- All code is thoroughly tested.

VI. System Architecture

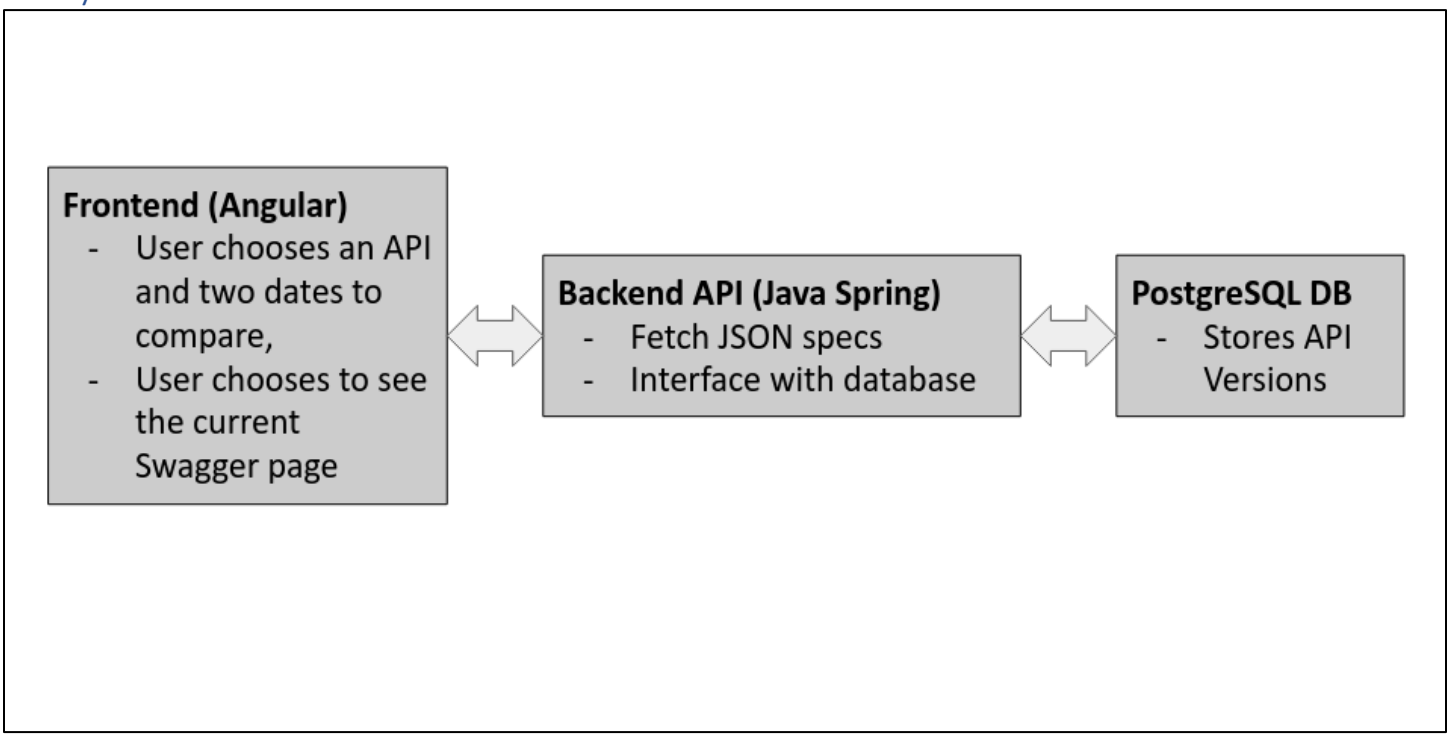

Figure 1: System Architecture

This system is designed as a full-stack web application for viewing and comparing different versions of third-party API specifications (Swagger). The architecture is composed of a frontend, a backend, and a database.

- Frontend (Angular): provides a UI for selecting APIs on various dates to assess their evolution over time.
- Backend (Java Spring): manages the core logic like scraping the API specs, computing the diffs to be displayed on the frontend, and storing and retrieving data from the database. As shown in Figure 1, the backend also serves as the interface between the frontend and the backend
- Database (PostgreSQL) stores the collected Swagger specifications with their metadata like timestamps and hash.

The JSON specifications that we are working with are formatted like a tree. Meaning that every endpoint has several “child” methods; every method has “child” responses, parameters, and so on. This means that processing this data and working with this data also like a tree is the optimal way to work with this data, to reduce future complexity and better model the original structure.

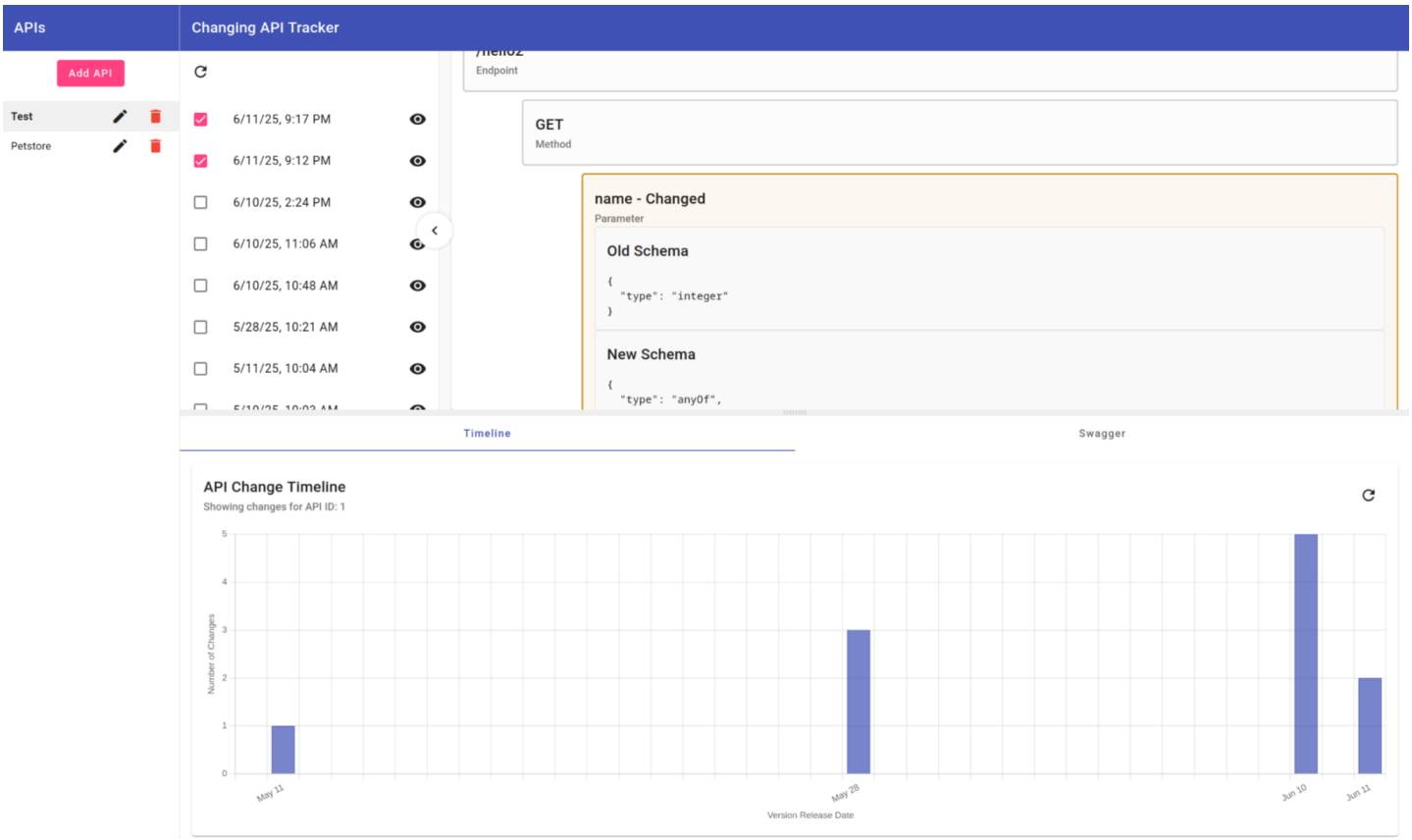

Figure 2: Frontend Design

The front end is structured as a single page web app, which is optimal for this use case where quick navigation is very important, and information is not so detailed that it needs to take up the whole page. This single page consists of several core components.

- **API Selector:** Adding, deleting, editing, and viewing the list of tracked APIs
- **Version Selector**: Selecting for comparison, and viewing API specifications for specific API versions
- **Diff:** Displayed the diff of the two selected versions in Version Selector
- **Timeline:** Provides a general overview of when all API changes have happened, with magnitudes
- **Swagger:** Displays swagger pages for selected API version

The backend performs several tasks, such as:

- Recurringly grabbing API specifications from a list of Tracked APIs every 6 hours
    - The set of Tracked APIs is a list of URLs which contain the OpenAPI specifications for their respective API
    - New API specifications are only stored in the database if they are different from the previous API version
- Computing the difference between two API specifications including added, removed, or renamed endpoints parameters, response types, as well as general data changes
    - Detecting added or removed endpoints is relatively simple
    - To detect an endpoint or parameter that was moved or renamed, the added and removed structures in a single change need to be compared, and if they are more than 50% similar, the object has likely been moved, and is represented as such
- Providing an interface for the frontend to access and modify various stored and calculated data points such as:
    - Stored API specifications
    - Computed diff between API specifications
    - List of all APIs

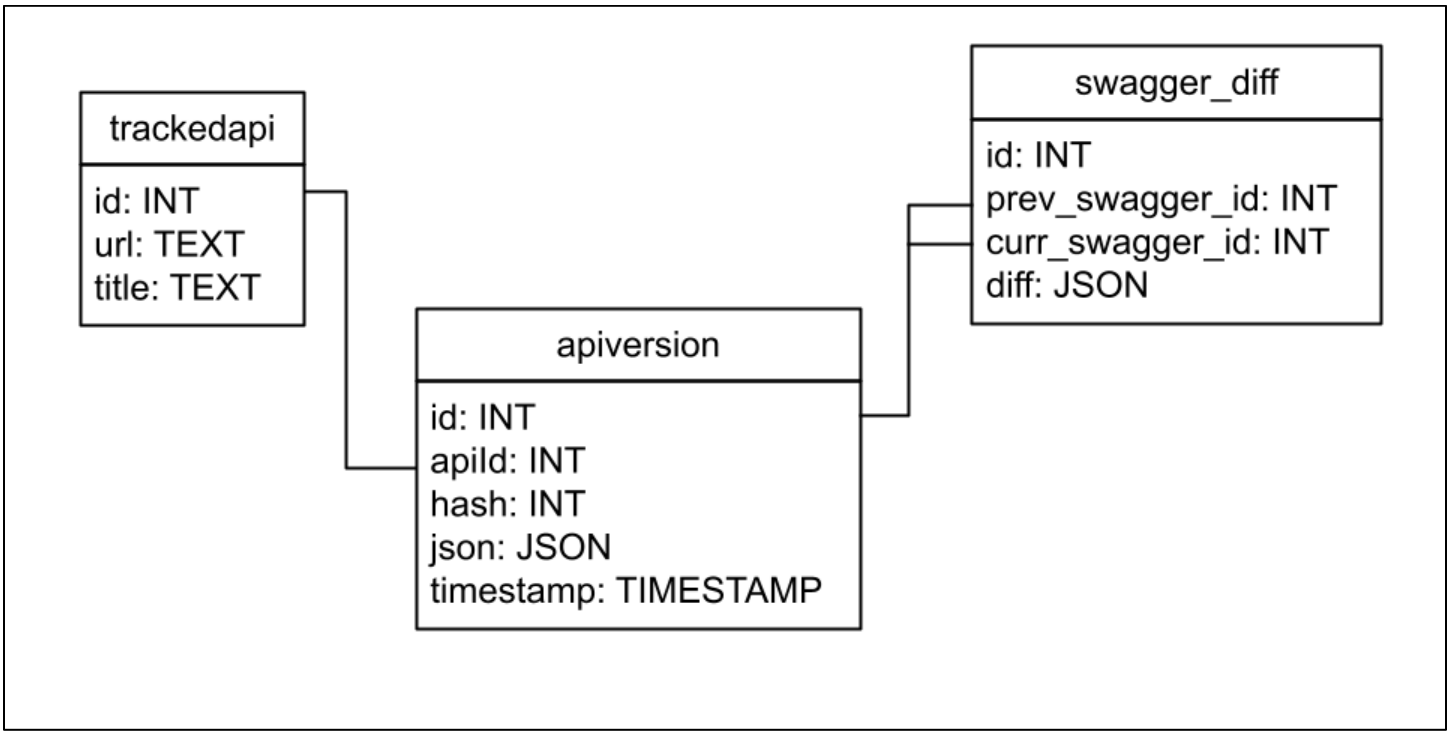

Figure 3: Database Schema

The database stores API specifications, diffs, and API locations, as shown in Figure 3. API specifications as well as diffs are stored as JSON as opposed to storing the separate pieces of that data (e.g. endpoints, parameters, etc.) in their own tables because JSON storage is much easier to setup and work with. Additionally, there is minimal advantage to storing the data in a way that it can be queried using relational database commands. In every case covered by the functional requirements, APIs are treated as a complete object by themselves, and individual endpoints or other pieces of data do not need to be tracked.

VII. Software Test and Quality

Test 1: Add APIs to be tracked

- Purpose: Ensure that users can input a title and URLs for API documentation and successfully begin tracking them.
- Description: Enter a valid OpenAPI spec URL through the frontend. Confirm backend has pushed the new API to the required tables in the database.
- Tools: Cypress, Angular HttpClientTestingModule
- Threshold for Acceptance: URL must be validated, tracked and displayed within 5 seconds
- Edge cases:
    - Submitting a duplicate API
    - Submitting with invalid URL (must not add or break)
    - Submitting valid URL with non OpenAPI content
- Results:
    - All valid URLs successfully tracked
    - Duplicate and invalid URL rejected with proper errors
    - Valid URL with non OpenAPI content not added to database.

Test 2: Select two versions for comparison

- Purpose: Ensure UI lets users select two valid versions for diff comparison.
- Description: Load the version history of an API and select two entries. Confirm the correct IDs are passed to the diff view.
- Tools: Jasmine, Karma
- Threshold for Acceptance: Versions selectable; diff view enabled only with valid pair.
- Edge cases:
    - Selecting same version twice
    - One or both versions missing
    - Trying to diff versions from different APIs
- Results:
    - Only valid pairs trigger diff generation.
    - Proper UI warnings shown for invalid selections.

Test 3: Display diff visually between two versions.

- Purpose: Confirm the structural changes between versions are shown clearly.
- Description: Load two version documents, perform the diff on them, and display the diff tree.
- Tools: Jasmine
- Threshold for Acceptance: All major changes detected and rendered accurately (changes to results, parameters, endpoint structures).
- Edge cases:
    - Comparing large documents.
    - Comparing identical documents.
- Results:
    - Diff accurate and color coded.
    - Large diffs (10000 line file) handled within half a second.

Test 4: View any version document.

- Purpose: Allow user to load and view full content of any stored version.
- Description: Select a version from the list, fetch full JSON, display in bottom tab UI
- Tools: Jasmine
- Threshold for Acceptance: Version loads in <1 second, displayed without truncation
- Edge cases:
    - Very large version file
    - Malformed or incomplete file
- Results:

    - All versions rendered correctly
    - Errors shown for invalid content

Test 5: Visualize change over time.

- Purpose: Show historical evolution of APIs over time
- Description: Render a timeline summarizing change volume over time using stored metadata.
- Tools: Cypress, Chart.js
- Threshold for Acceptance: Major change dates are immediately obvious.
- Edge Cases:
    - APIs with many changes
    - Long periods of no changes
- Results:
    - Chart correctly reflects change volume history.

Test 6: Responsive Layout and Usability.

- Purpose: Ensure the frontend is usable on various screen sizes.
- Description: Manually and automatically test UI responsiveness on desktop, tablet and mobile.
- Tools: Cypress (viewport), Lighthouse
- Threshold for Acceptance: Layout adapts with no broken UI at common screen sizes.
- Edge cases:
    - Small mobile screens
    - Zoomed in views or accessibility modes.
- Results:
    - Layout adapts fluidly
    - No overlap or hidden content.

Test 7: Handle API submission

- Purpose: Confirm that submitted URLs are parsed and processed properly.
- Description: Send POST request with API URL; expect parsing and validation followed by initial diff and storage.
- Tools: JUnit
- Threshold for Acceptance: Valid Swagger / OpenAPI URLs return 200 OK; invalid 400/422
- Edge cases:
    - HTTP timeouts from target.
    - API returns invalid JSON.
    - API returns JSON in invalid format
- Results:
    - Proper HTTP responses sent.
    - Logs show parsing logic triggered as expected.

Test 8: Scheduled scraping runs every 6 hours.

- Purpose: Ensure backend scheduler functions reliably.
- Description: Simulate or override the scheduled method to verify timed scraping occurs.
- Tools: JUnit
- Threshold for Acceptance: All APIs scraped every 6 hours, or manual override triggers same result.
- Edge cases:
    - Network failure on one API
    - Unexpected format change in API output.
    - API spec in incorrect format
- Results:
    - Scheduling triggered confirmed.
    - Logs capture skipped or failed scrapes.

Test 9: Detect and store only changed versions.

- Purpose: Avoid storing duplicate versions.

- Description: On each scrape compare the fetched spec with most recent version stored in the database by comparing hash signatures, and store only if differences exist between the two.
- Tools: JUnit
- Threshold for Acceptance: No identical version stored twice.
- Edge cases:
    - Whitespace or comment changes only.
    - Reordering of fields.
    - Changes to unimportant fields (documentation related changes like “title” and “description”)
- Results:
    - Only true content changes trigger new version.

Test 10: Store and retrieve version data.

- Purpose: Ensure correct database persistence.
- Description: Test database write on scrape, and subsequent reads for frontend access.
- Tools: PostgreSQL Docker, PgAdmin, testcontainers
- Threshold for Acceptance: Version data stored with timestamp, fetched accurately.
- Edge cases:
    - Concurrent writes
    - DB container restart
- Results:
    - Data remained intact
    - Accurate data fetch confirmed.

Test 11: Page load and performance.

- Purpose: Ensure fast load times and interaction times.
- Description: Run performance audits for tracked list, diff view, and timeline view.
- Tools: Lighthouse, chrome dev tools
- Threshold for Acceptance: Pages load and become interactive <2s
- Edge cases:
    - 200+ APIs in the list
- Results:
    - All pages performant under load

Test 12: Security, input and XSS protection.

- Purpose: Prevent malicious input from breaking frontend or corrupting backend.
- Description: Submit script injected and malformed payloads to form fields and URL inputs.
- Tools: OWASP ZAP
- Threshold for Acceptance: No script executes, No DB corruption
- Edge cases:
    - Encoded XSS payloads
- Results:
    - All malicious inputs sanitized.

VIII. Project Ethical Considerations

Throughout this project we encountered many ethical issues relating to the ACM Code of Ethics [1].

- **ACM 2.6: Perform works only in areas of competence.** None of us were experienced in web development and front-end work. We also had no members who were familiar with our backend technology, Java Spring. This could put us at risk of violating ACM 2.6 professional responsibility.
- **ACM 1.3: Be honest and trustworthy.** When advertising whether a change is breaking, that information needs to be correct, because an incorrect result could lead to users introducing bugs into their software. If some specific feature isn’t supported, that information should be made clear to the user. This relates to ACM Code of

Ethics guideline 1.3 because the project needs to be honest and open about the shortcomings, especially when the consequences are larger.

- **ACM 1.2: Avoid harm.** The tracker should not be adding any noticeable stress to API's servers (limiting querying). Frequently sending requests to a server, especially considering different users of this application may all individually be sending requests, may result in additional stress which could harm the APIs performance for standard users which will have negative consequences.

IX. Project Completion Status

As of the time of writing this report, the application has achieved every aspect of the definition of done we laid out at the beginning of this project:

- **App runs locally:** We have created a docker compose file and containerized every aspect of our app, including frontend, backend and database. The whole project can be run simply by entering a docker compose command to start the system.
- **Stores API Version Data:** The app has a scheduled fetch job that queries an API for its version specs, and it compares the hash of this version to other versions to check if it is new. If it is new, the spec will be stored in the database.
- **API Diff:** The diff panel in the app takes two specs for two different versions of an API. These specs are parsed, and the diff algorithm is run to produce a raw diff tree. This diff tree is visualized in an organized, color-coded tree which clearly indicates what has changed.
- **API Timeline:** The timeline panel clearly shows how an API has changed over time in terms of volume of changes. The Y-Axis indicates the number of changes since the previous version. This gives an indication of how often updates are pushed as well as how big these updates are in terms of number of changes.
- **App GUI:** The app has an Angular single page site that is clean, robust and clearly displays the requisite API information.
- **All code is tested:** A set of unit tests for every component in angular (10+) using a mock testbed that can simulate the execution of a component's functions without interacting with the backend. Similarly, the backend was tested with a set of JUnit tests. The whole app was tested with a series of manual end-to-end tests, with edge cases and every feature tested thoroughly.

One feature that was discussed initially but was removed as a requirement for the project and thus not completed was being able to customize the amount of time between fetching each API. This would be useful to adapt the time between fetches based on how often an API is expected to change, so that the server is not repeatedly trying to grab data from an API which almost never changes. Another feature that we considered implementing was a Kubernetes cluster for product deployment, where Kubernetes is a framework for setting up and deploying systems of containerized applications. We decided against committing to this feature because of the time constraints of the project and the complexity associated with such a task.

X. Future Work

1. Making the amount of time between API fetches customizable: this would require an option in the frontend interface to specify this amount of time, a field in the database to store this value, and a potential larger rework to the backend fetching, depending on the specificity of how often the API is fetched. This should not require any more knowledge than the already completed part of the project and would likely take a couple of days to implement.

2. Ensuring that the application works as well on mobile as it does on desktop: using Material UI, some of these issues of handling these differences are automatically handled, however more testing needs to be done to ensure everything works and is displayed well in all scenarios. This would require additional knowledge of how to handle different browser types and would likely take a couple of days. However, that could vary a lot, depending on the amount of work that is revealed during testing.

3. Supporting multiple users with different sets of tracked APIs: this would require adding a table to the database for the multiple users, as well as adding a new column to the API database to store which user is tracking which API.

XI. Lessons Learned

1. Using as many prebuilt components in the frontend as possible is the best way to get things done, this meant using Angular Material in our frontend to have a variety of prebuilt components that became very useful for our application. Using these features significantly decreased development time, as well as increasing cross platform consistency.

2. Taking full advantage of the features of whatever backend framework is being used creates more understandable, and less buggy code. For example, using a specific Spring feature to recurringly fetch API specifications, as well as using Spring APIs to interact with the database.

3. Don't overcomplicate things. If there is no advantage to storing data in a more complex way in the database, then the simpler way is better.

4. Giving everyone at least some knowledge of every component of the project can speed up development time, as opposed to everyone having specific domains. This allows everyone to be able to debug and fix issues that might be happening across many components of the project.

XII. Acknowledgments

We would like to thank our points of contact at ICR, Jasper Mesenbrink and Jimmy Baldwin. Jasper, our primary contact, provided essential guidance on structuring the project and offered valuable insight into writing maintainable, extensible code. Although we met with Jimmy only once, his advice on testing and UI improvements was highly impactful. We also thank Scott Jensen for consistently encouraging us to stay focused and realistic in our goals, ensuring we fully understood our project's scope.

XIII. Team Profile

Daniel Castelluccio

Computer Science – Computer Engineering Focus
Hometown: Maple Valley, WA
Work Experience: None
Hobbies: Running

Colin Wolff

Computer Science – Robotics and Intelligent Systems Focus
Hometown: Woodinville, WA
Work Experience: None
Hobbies: Skiing, Tennis

Joe Steele

Computer Science – General Track
Hometown: Littleton, CO
Work Experience: DSP Engineering, Zeta Associates
Hobbies: Gardening, Writing

References

[1] “ACM Code of Ethics and Professional Conduct,” Code of Ethics, https://www.acm.org/code-of-ethics (accessed Jun. 13, 2025).

Appendix A – Key Terms

Term	Definition
Angular	*Typescript based free and open-source web development framework for single page web applications.*
API	*Application Programming Interface, a specific way of communicating with a given process*
Cron job	*A recurring scheduled process*
Cypress	*Browser-based, frontend testing tool*
Docker	*Platform for packaging applications and their dependencies in lightweight containers that can run across different environments*
GUI	*Graphical User Interface*
Jasmine	*A framework for behavior-driven JS testing with unit tests for frontends*
JSON	*JavaScript Oriented Notation*
Karma	*Executes JS tests in a browser and returns the results*
Kubernetes	*Open-source container orchestration system for software deployment*
Lighthouse	*Analyzes the quality of a webpage to expose failures in performance, best practices, accessibility, etc.*
Material UI	*Component library that implements Google’s Material Design Specification in Angular*
OWASP ZAP	*Zed Attack Proxy, this is a security tool used to test web applications for vulnerabilities*
pgAdmin	*Web interface for PostgreSQL databases*
PostgreSQL	*Commonly used database management system*
Swagger	*The OpenAPI JSON format documentation for APIs*
Spring Boot	*Java-based framework for building and deploying backend applications*
Testcontainers	*Creates disposable database instances for to test their usability in containers*
XSS (Cross-Site Scripting)	*A web security vulnerability where attackers execute negative scripts through a legitimate website*