COLORADOSCHOOLOFMINES
EARTH • ENERGY • ENVIRONMENT

# CSCI 370 Final Report

**The Reactionaries**

Dean Coventry

Vincent Nguyen

Sean Williams

Revised June 12, 2024

CSCI 370 Summer 2024

Dr. Rob Thompson

Table 1: Revision history

| Revision | Date | Comments |
|---|---|---|
| New | 5/13/24 | Report document created with the template |
| Rev – 1 | 5/15/24 | Wrote requirements portion |
| Rev – 2 | 5/16/24 | Added more technical specificity in the introduction |
| | 5/16/24 | Rewrote/deleted aspects mentioning the previous technology decisions – solidified what technologies to use |
| Rev – 3 | 5/17/24 | Added to *Introduction* making it introduce the section better |
| Rev – 4 | 5/19/24 | Wrote the *Definition of Done*. Can be reduced and scaled back if needed later |
| Rev – 5 | 5/19/24 | Reworded some sections in *Functional Requirements* |
| Rev – 6 | 5/20/24 | Rewrote the *Definition of Done* to adhere to the input and request of what the client wants |
| Rev – 7 | 5/23/24 | Added *Technical Design Issues* – splitting *General Risks* into two sections |
| Rev – 8 | 5/23/24 | Added *System Architecture Diagram* and respective explanation |
| Rev – 9 | 5/23/24 | Reworded the technology part of *Non-Functional Requirements* |
| Rev – 10 | 5/29/24 | Wrote and added to *Software Test and Quality* |
| Rev – 11 | 5/30/24 | Rewrote *Software Test and Quality* to adhere to our Ethics presentation and plan. Broke up section into multiple subsections: *Testing* & *Additional Quality Assurance Considerations* |
| Rev – 12 | 5/30/24 | Added to *Relevant IEEE/ACM Principles* |
| Rev – 13 | 5/31/24 | Added to *Project Ethical Concerns* and *Testing*. Reorganized sections related to Ethics |
| Rev – 14 | 6/08/24 | Began writing the *Project Completion Status* section |
| Rev – 15 | 6/09/24 | Added to the *Project Completion Status, Future Work, And Lessons Learned.* |
| Rev – 16 | 6/09/24 | Updated project status to reflect new progress |
| Rev – 17 | 6/11/24 | Reformat, and update *General Risks* to be more relevant |
| Rev – 18 | 6/11/24 | Elaborated more on the *System Architecture Diagram* and explained it with more technical terminology |
| Rev – 19 | 6/11/24 | Added figures and references |
| Rev – 20 | 6/12/24 | Final proofread before peer review |
| Rev – 21 | 6/12/24 | Updated *Revision History* and *Table of Contents* |
| Rev – 22 | 6/12/24 | Added a brief description of the client to the *Introduction* and added headers |
| Rev – 23 | 6/15/24 | Fix minor errors, and edit the report to address peer feedback. |

# Table of Contents

# I. Introduction

**Client Information:** Qualcomm is an American semiconductor company founded in mid-1985 and is currently headquartered in San Diego, California but is multinational with offices in several places around the globe. They are most known for their work and manufacturing in the semiconductor and telecommunications industries–creating components used in many devices from individual, average consumer use to commercial applications. For example, Qualcomm's line of ARM architecture system-on-a-chip (SoC) central processing units (CPUs) that power some devices such as the majority of Android smartphones in the past leading up to the modern-day and, more recently, the ARM CPUs that are powering new AI-powered PCs with their *Snapdragon® Elite X* line of CPUs. Furthermore, Qualcomm also has a focus on manufacturing antenna components for cellular devices and also for cellular towers that feed data to mobile devices. Qualcomm has been and still is a key player in the mobile technology and telecommunications industry.

Staying on the topic of cellular technology, we are working with the office and team at Qualcomm's Boulder campus in Boulder, Colorado. They are primarily focused on the engineering and manufacturing of the Test Base Station (TBS), a component whose purpose is to test cellular tower base stations around the globe. A base station is a hub for wireless communications between devices and the purpose of the TBS is to simulate the way a real base station acts in a cellular network.

**Problem Background Information:** Like any object that goes through the engineering process, bugs/issues can arise in the creation of the hardware and software of this component and there needs to be ways that these problems can be tracked so that engineering teams can fix them. Alongside that, the issues need to be processed and delegated to specific teams that are responsible for certain parts of the product. This means that they are specialized to know a lot about the aspects of the individual part they are in charge of working on and can provide the best support to fix the issue promptly. Hence, Qualcomm used a tool named Jira, created by the Australian company Atlassian, to handle the requirements needed to manage and keep track of any bugs and issues.

**Project Description:** Jira is a proprietary issue-tracking platform developed by Atlassian that is used in many companies and projects to track bugs reported by employees and clients. Qualcomm uses this platform to keep track, delegate tasks, and track issues that arise and inform others about the status of the issue, information about it, its origin/creator, and a lot more information. However, Qualcomm's internal Jira instance is slow, lacks customization, and contains too many unused features, making it inefficient for the company's workflow. Our goal is to create a custom application that factors the Qualcomm team's most frequent actions and use cases while preserving the functionality of Jira. Therefore, this involves building a new web application that is an abstraction on top of the existing Jira instance and uses the Jira Rest API paired with the Jira Query Language (JQL) to formulate queries for creating, reading, updating, and deleting (CRUD) the data.

# II. Functional Requirements

The function requirements of this project begin with the fact that this application needs to be accessible to many people within the Qualcomm team in Boulder meaning that it has to be a web application such that it can be accessed in a browser. It should adhere to all the features that the customer service team(s) would need to view and manage issues and delegate them to the appropriate team to work on. A backend must also be developed to foster communication between the frontend and backend business logic where we need to interface with Jira.

Our client provided us with the following overarching goals for the application;
Ability to quickly filter on common issue properties

1. Filter presets
2. More accessible issue actions
    a. Escalate from Tier2 (customer support teams) to Tier3 (engineering teams)
    b. Report from System Integration and Test to Tier3 for release verification and bug reporting
    c. Handover from Tier2 to Tier3
    d. Approve for LTS merge and display LTS analysis for high visibility (prompt if not available)

    e. Add/manage feature tracker (with prompt)
    f. Add/manage releases (with prompt)
    g. Issue resolution (with the root cause for bugs, handle Tier2/3 flows)
    h. Add components to issues
    i. Link issues (prompt for TBS Jira with smart filtering)
    j. Schedule issues for Sprint
    k. Multi-select issues for bulk actions
3. Issue statistics
    a. Display the total number of issues
    b. Average resolution time
    c. Tier2 escalation rate
4. UI design/components
    a. Break up into user flows
    b. Order Jira fields by priority in a flexible layout
    c. Display issues as cards that can be expanded for more details
    d. Highlight full-stack builds into which the issue has been merged
    e. LTS merge analysis
    f. Visualization of dependencies (node graph)

Performance should be faster than the current Jira instance. Filtering should take less than 1 second, selecting a pre-set filter should take less than 5 seconds, and executing an action should take less than 3 seconds.

As the scope of all of these goals is outside what can be done within a 5-week period, a handful of items from this list were chosen for the project "definition of done" (described later). In general, the goal of our work during this period is to build a prototype and platform that can be extended to implement all the features mentioned above in the future.

Furthermore, the tool was implemented with a React frontend and Python backend to stay consistent with the Qualcomm team's existing infrastructure and workflow. The React app is served as a single bundled page with Vite by a simple web server to reduce complexity and unnecessary server tooling. The backend is a persistent Python/Hatch server that will await any HTTP requests sent from the frontend and handle them accordingly–serving as a mediator between the frontend graphical user interface (GUI) that the end-user deals with and the Jira instance's data and all the logic of that.

## III. Non-Functional Requirements

The non-functional requirements laid out in this project are:

1. Virtual machines and virtual private network (VPN)
    a. Grants us internal access to all Qualcomm services (i.e. Jira, GitHub, etc.)
    b. Gives us a powerful platform to build our application with no concern for the virtual machine, build, or deployment cost
2. Setup GitHub actions
    a. Already configured for us during the project creation
    b. Needs to successfully build every time code is pushed/merged into the main branch for an outputted, packaged application
3. The user interface (UI) is streamlined and accessible
    a. Easy to navigate and everything is clearly labeled and visually distinct
        i. Items that aren't labeled should use common recognizable icons, such as a hamburger icon for menus
    b. Pleasing to the eye
    c. UI components are accessible with typical keybindings, such as tabbing between form items and pressing space/enter to select an item.

4. The application integrates with Jira automatically, the user shouldn't have to reconfigure the application unless there is a major change
    a. Use Qualcomm's DNS and utilize known paths of accessing the database
5. Securely hosted
    a. Fortunately, this step is handled by the hosting network itself being only accessible by Qualcomm employees. However, we must ensure we're not accidentally leaving this software open to the internet.

# IV. Risks

## IV.I General Risks

As our team is working with confidential information, it is important to work within Qualcomm's constraints and clear any uncertainties with our contact. This includes keeping all code on our virtual development environments and not sharing details with people outside the company. We also must use secure and Qualcomm-monitored communication methods when discussing confidential information within our team.

Additionally, our application will be connected to their live, existing Jira database. Therefore, the effects of malformed, unauthorized, or malicious requests to the Jira database raise a considerable risk of genuine harm to Qualcomm's developer workflow. To mitigate this, our system architecture (described below) prevents direct requests to Jira, and instead, all requests are sent through our custom backend API - allowing for validation, verification, and other necessary safety measures.

## IV.II Technical Design Issues

Due to Qualcomm's isolated network for our development environment, we've encountered several issues surrounding any library that "phones home" during initialization or runtime–in other words, pinging an external URL not whitelisted to be used within the environment. For example, on the frontend, *shadcn*—our desired React component library—could not be initialized because of an SSL error invoked when shadcn tried to grab config files from their website. Furthermore, an SSL error was also encountered when trying to set up the backend and its dependencies with the Python Hatch tool–our preferred build tool to help package and host our server-side Python backend.

These issues were mitigated for our team by switching to a Linux-based development VM. However, it should be noted that this issue could affect the future development/maintenance of this project if the same network restrictions apply. From the beginning of the field session to when we could begin tangible, full productivity work took three weeks of headache-inducing issues–the majority of which were out of our control.

# V. Definition of Done

As there may not necessarily be enough time to ship a finished build of our project meeting all the criteria/requirements laid out by our client line by line, we established a realistic goal to reach by the end of the session.

First, a finalized architecture diagram must be created and followed, to ensure that future teams working on our code will know how the application operates. Along with that, our working prototype should be on the company's GitHub and deployed to their Drekar server—this prototype must be integrated into a live Jira instance and include the top 5 prioritized filters (Release, Products, Issue Types, Priority, and Tier2/3 Support), two of the prioritized actions (Escalate from Tier2 to Tier3, and Tier3 to Tier3 Handover), and be easily extendable for future development. It must also be developed such that performance evaluation can be done, however it doesn't need to strictly meet performance criteria.

# VI. System Architecture

The large-scoped architecture diagram depicted below (see *Figure 1*) relates to the high-level components and data flow necessary for our Jira web application.

A basic web server serves up a single-page React application (SPA) to any client who accesses and requests it from a web browser. The React app then communicates with a separate Python FastAPI server to find, filter, and modify Jira issues using the Rest API that uses HTTP methods such as GET, POST, etc.

The Python server hosting an instance of FastAPI acts as a middleman that transforms typical web app API calls into requests the (existing) Jira API can interpret, likewise, it translates the returned Jira data into something that can be parsed/rendered by the web app.

Overall, this diagram flows in both directions from the webpage to Jira and vice versa.

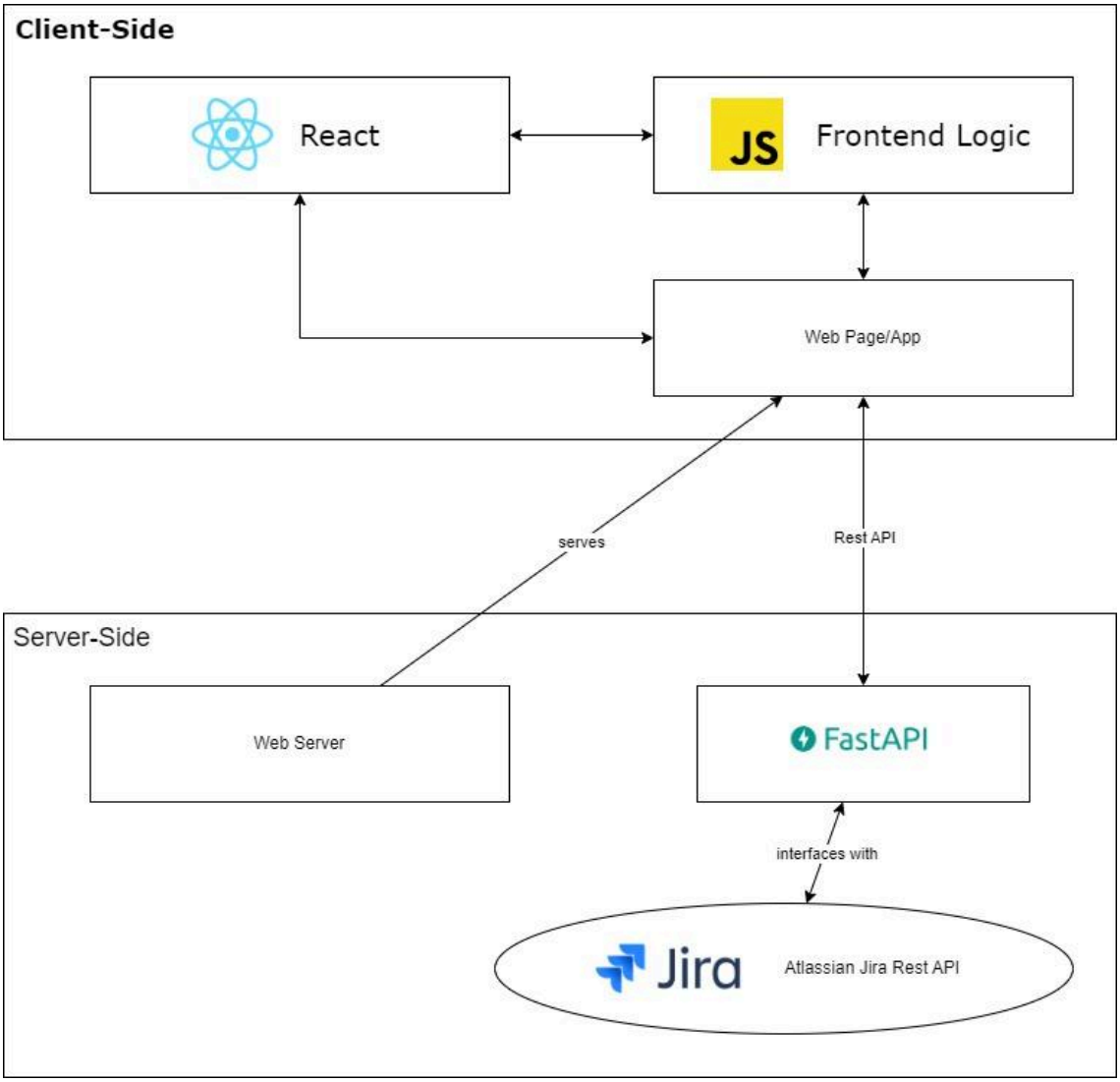


Figure 1 | Application Architecture Diagram

From the user's perspective, the SPA should allow for quickly filtering, viewing, expanding, and performing actions on Jira issues - displayed in the UI “as cards”. Our design for the frontend is based on the mockup and workflow depicted below in *Figure 2* and *Figure 3*. This design and user flow were created based on early mockups and discussions with our client.

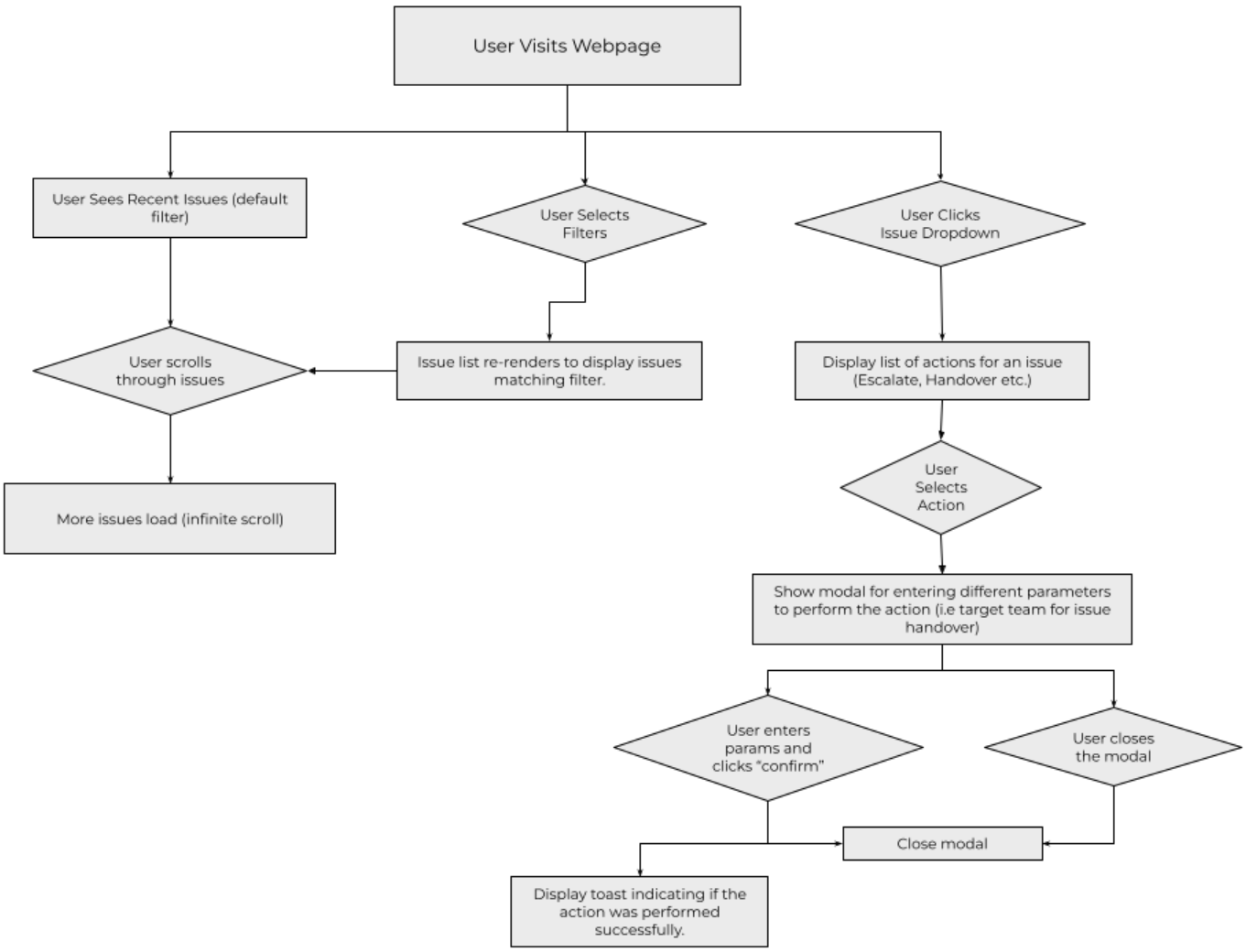


Figure 2 | Frontend Workflow Diagram

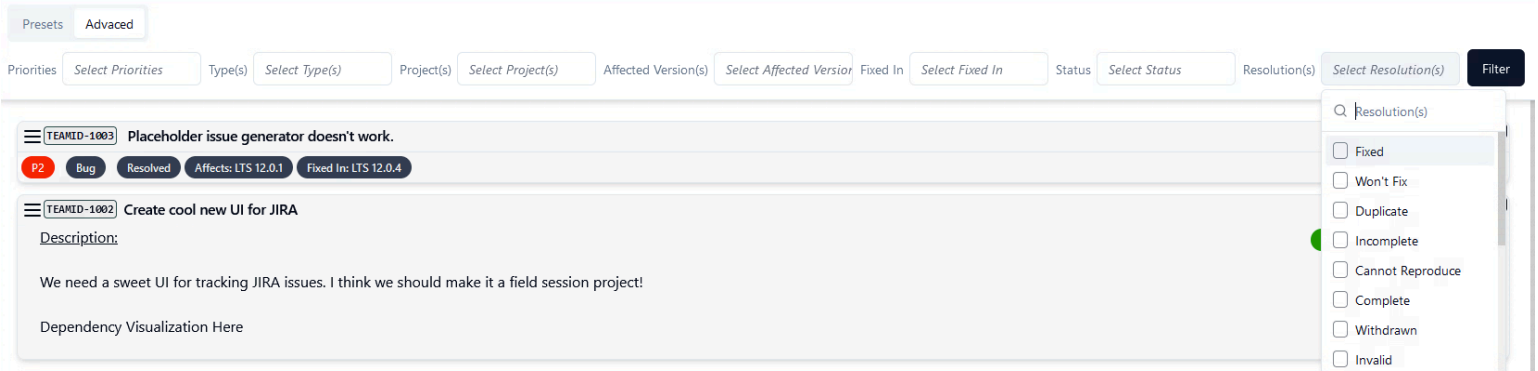


Figure 3 | Frontend Mockup.

# VII. Software Test and Quality

## VII.I Testing

The majority of this application's functionality stems from transforming frontend requests into something Jira's API can understand, and likewise sending/interpreting Jira's response back up at the frontend. Implementing this leads to requests going through a series of transformations from frontend to backend to Jira and back. Due to this series of several transformations, validating the integrity of a request within each layer of our application architecture is our priority for testing. Following this flow of data gives clear insight into what and where our tests need to be for each component of the application;

Starting with the frontend, we need to verify that any submitted filter request properly converts the UI form values into a Restful GET request that follows our custom-defined API format. Everything that the user entered as filter parameters should be within this request, and in the format that the backend expects.

- This was tested manually by filling the filter form with some values and comparing the differences with the GET request URL against what we're expecting those filter params to create. In the future, this layer of testing would benefit from a tool like Jest for more robust checks.

Assuming the request is properly sent from the frontend to our Python backend, we also need to verify that the backend properly and deterministically transforms the GET request it received into a valid and accurate Jira Query Language (JQL) query to send to Jira. Likewise, we should validate that Jira is responding with the issues we expect for a given query, or responding at all.

- Our transformation is tested with PyTest using fake queries and verifying the query to the JQL method is deterministic and accurate
- We tested with directly sent API queries and verified that those queries are transformed into the JQL we expect
- The combination of both of these tests is ideal, as Pytest has strict expectations such that we can catch/address issues immediately while a manual test allows us to quickly change with query values and verify those changes are reflected accurately in the transformed statement.
- Testing Jira's response can also be done with Pytest, given some sample queries validate that Jira's response is what we expect. However, we were unable to implement these tests due to time constraints.

Returning up, we need to verify that the backend transforms Jira's response into the issue "card" data format that the frontend expects. We expect this transformed data to be accurate, with nothing "lost in translation."

- Since our backend mainly just forwards the Jira issue JSON data straight up to the frontend, this doesn't require any robust tests other than verifying the information is forwarded at all.

Now, assuming the above has all worked properly, we test that the frontend properly interprets and renders the Jira "cards" that the server passes up. All of the cards should be accurately displayed concerning the issue they represent, with no duplicate cards or any other strange behavior.

- This is done with some manual checks against Jira's existing UI, given some known responses.

Lastly, we should test the entirety of the above together, to ensure nothing is "lost in translation" overall. This is to catch anything we may have missed within the individual checks between the aspects of our application. In other words, do a final test from the user's perspective - enter a query and get the appropriate Jira cards. If there's any discrepancy or issue in this final stage, it's an indication that we need to go back to our earlier tests and improve them.

- We should have a couple of manual/robotic tests that enter different filter options into the UI, and validate that the returned cards are what we expect in their entirety.

Similarly, tests will be done later for any card/issue modification requests. Following the same layers for any push-like actions for editing Jira cards yields similar testing needs at each stage of data flow in our application;

- Verify the action is in the correct format and targets the correct card with the changes specified.
- Validate that the change request is properly parsed and converted by our backend and sent to Jira.
- Verify the changes that occurred on Jira's side.
- Multistep actions like escalation will need to verify every part of that process.
- Verify the changes will now be reflected on the frontend (i.e. trigger the UI to update).
- Additionally, the UI should have optimistic changes (for better UX) that are later rectified if needed.
- Finally, validate that the system works overall, from the user's perspective.
- All of the aforementioned tests should be maintained and utilized after every major change to our code, to ensure that none of our changes has led to unexpected bugs or behavior.

Unfortunately, since we implemented the card modification code at the very end of the project, we did not have adequate time to fully implement robust tests by the guidelines above.

### VII.II Additional Quality Assurance Considerations

To make sure the client is satisfied with our work, we often check in with them, to make sure we don't stray from their vision or waste time on unimportant features. Additionally, the team should be pulling, testing, and working with up-to-date code to verify that each part of the project integrates and to catch any "works on my machine" incidents early into development.

Refactoring was done before any major change. This is to ensure the quality of the code is maintained as well as allowing the team to verify that we all fundamentally understand how each component of the application works. Refactoring is a great opportunity to study and ask questions about the code someone else made.

Any other performance or metric-based tests use relevant flame graphs, network analysis, or debugging tools. For example, looking for unnecessary re-renders in React is done using flame-graph tools, such as the one included in the "React Developer Tools" Chrome extension. While these bugs or performance hiccups cannot be easily found with a standard test suite (as they technically don't cause 'incorrect' behavior), they should still be observed to ensure our code is well-written and optimized.

## VIII. Project Ethical Considerations

Fortunately, this project mainly serves as a custom and business-internal interface for existing technologies, so there's little potential for harm or impact on public well-being. However, there are still plenty of ethical principles that should be kept in mind for the duration (and future of) this project;

### VIII.I Relevant IEEE

IEEE 2.05; "Keep private any confidential information gained in their professional work, where such confidentiality is consistent with the public interest and consistent with the law." and ACM 1.7 "Honor confidentiality"

- Since we are working with Qualcomm's internal Jira database, maintaining confidentiality is a vital part of this project as we have visibility of projects, development cycles, and bugs not released to the public.
- This sets certain precedents with how we develop our code, for example, we need to be careful to scrub out any confidential information if posting errors to places like StackOverflow or similar.
- Additionally, we need to work only within Qualcomm's internal network, code should not be shared, edited, or otherwise taken outside of their corporate GitHub.

IEEE “3.10. Ensure adequate testing, debugging, and review of software and related documents on which they work.”

- Our team is building this software from the ground up, meaning we are responsible for ensuring that everything is tested and functional.

IEEE 3.11 “Ensure adequate documentation, including significant problems discovered and solutions adopted, for any project on which they work.”

- This project will not be fully completed within the time we have, therefore it is paramount that we write and maintain good documentation and notes about our code and solutions for the continued development of this software to run smoothly afterward.
- In particular, we’ve dealt with several unique issues and workarounds that should be professionally reported to support future development teams.

IEEE 6.08. Take responsibility for detecting, correcting, and reporting errors in software and associated documents on which they work.

- We have been in, and intend to maintain, constant contact with our client at Qualcomm via Teams to communicate any errors, difficulties, or roadblocks we’ve encountered for the duration of this project's development.

### VIII.II ACM Principles

ACM 1.2 “Avoid Harm”

- As this project will be directly linked with Qualcomm's existing Jira instance, it’s paramount that this software acts as expected, especially in the realm of modifying issues within their database.
- Any unexpected behavior could cause genuine harm and issues if they cause issues to be deleted or modified in a way the user did not expect.

ACM 2.4 “Accept and provide appropriate professional review.”

- As part of our constant contact with the Qualcomm team, we reviewed the progress of our project with them frequently and used this as an opportunity to receive feedback and suggestions to drive development forward.

## IX. Project Completion Status

The frontend of this project has been nearly fully designed with all the main components of the application and is in a good state to add the remaining tooling that we were unable to implement. Placeholder UI elements exist for the components of the aspect we did not complete. The frontend looks and behaves consistently in both Firefox and Chromium-based browsers.

The backend has been integrated with their internal Jira database and successfully queries just the Jira issues our client is interested in. Additionally, the data flow and “translation” of our Jira issue up to the frontend works. Due to the simple JSON format for issues that Jira provided, a majority of the information processing is handled client-side due to the ease of processing issues as-is with a simple JavaScript object.

The frontend can query and store contextual information from the backend such as the relevant projects and versions associated with our client's team. It can then use this information to map Jira issues to this cached context to improve performance and prevent unnecessary queries. Additionally, this system allows our frontend to easily implement and use up-to-date information for determining the properties to filter issues.

The filter system and actions have been mostly implemented, however, there are still some buggy aspects and some questionable behavior. The system is complete with the frontend to backend filter request API written. To dive into it a little, the input is received into the backend FastAPI as a URL containing the Rest API endpoint and query parameters appended onto the end of the URL. Then, the query parameter is translated back into an object in our backend where a JQL statement is constructed to make a GET request from the Jira Rest API. The frontend is configured to dynamically purge the previous data loaded in the cards when a filter is submitted which clears the list of cards and awaits the data returned from the FastAPI backend.

We were unable to complete many of our project's stretch goals and polishing. Although, as mentioned, we have a lot of placeholder and setup code for components we were unable to complete. For example, we did not have enough time to implement the “bulk action” system for our issues; but the code is written to select multiple cards and have the selected issue IDs tracked client-side such that future development can attach bulk action code with relative ease. Among these, we have a few extra stretch goals that we were unable to finish such as a parser for Jiras unique markdown format and the Jira issue dependency visualization system.

We did not have time to bundle and test our software whole, meaning a lot of the internal API calls still point to localhost and our UI is tested within the local Vite development server. To account for this, the frontend has a configuration file pointing to the relevant backend address, such that this can easily be changed based on how the software is deployed.

In the end, we were able to successfully deliver our application to our client by creating a pull request within GitHub to merge the final code from our branch to the *main* branch. The repository was set up in a way that only one of the team members in Qualcomm can approve and create a release for the application. We were able to push to our respective branches and were able to merge with the main branch, allowing GitHub’s automated deployment actions to be executed.

## X. Future Work

As our project has created a reliable channel to receive data from the Jira Rest API, this program can be extended to display, modify, or otherwise work with any data the API supports. This provides massive opportunity and ease of extension by working with this data. For example, visualizations and summaries such as time to resolve could be implemented by parsing through issue data.

A recommended next step would be introducing a sort/order by system for the issues. Currently, our application renders the issues in the order that Jira provides, which fortunately provides Jira issues in an easy newest to oldest list grouped by project. However, this default ordering may not be ideal for every use-case of our client and therefore would be a good next step for future development of this application.

## XI. Lessons Learned

Overall, we experienced major setbacks within the course of this project in the span of the five-week session. A major takeaway was teaching us how to deal with the corporate development environment. As we spent almost more than half of the field session project dealing with Qualcomm’s internal onboarding, setup, and troubleshooting, we learned the inner loops of how software development in a big, multinational company works. We had to jump through several hoops dealing with Qualcomm’s information technology (IT) team requesting and waiting for the approval of virtual machines for us to work on.

When working in Qualcomm’s internal network, we had to jump through several hoops to be able to access internal resources. This involved having to log into their virtual private network (VPN), a way to establish a secure tunnel to Qualcomm, installed on our machines. Then, we had to log in and work through an internal virtual machine (VM) delegated to us by the Qualcomm IT team.

Furthermore, after all of that was set up, we experienced a secure socket layer (SSL), a protocol for encryption and security throughout the web, certificate issue with being able to install packages from Python’s *pip* and NodeJS’s *npm* package managers. The reason behind this was that instead of pinging the official repositories for the respective package

managers, instead, they pinged the internal Qualcomm servers and the internal repositories they have for the package managers. This taught us that corporate companies have their own, proprietary ways of doing typical tasks such as installing packages for development or even building and deploying projects. Overall, the only way that this problem could be tackled in the future would be through advocating for ourselves and asking someone for help or referencing documentation that the companies provide which, in our situation, was a Confluence site for Qualcomm that the whole company referenced.

This project serves as a massive learning opportunity in terms of both Jiras Rest API and full-stack development as a whole. Below are some notable takeaways from the development of this project:

- TanStack's react-query serves as a powerful tool for fetching, mutating, and synchronizing data with a server and allows for easy work with pagination or "infinite" queries. However, the classic use of useEffect for quickly grabbing initialization data that you don't expect to change or re-query is still a reasonable approach to reduce complexity. React-query is extremely powerful and useful, but there are certainly times when it's not needed for communicating with the backend.
- Zustand provides a good starting point so that web developers can focus on the functionality of an application without needing to focus so much time on making frontend components that look good or provide functionality that is needed.
- FastAPI is a very powerful framework for writing Rest APIs and it focuses on the speed and efficiency of running and developing a Rest API that rivals other technologies such as ExpressJS. Leveraging Python's faster code execution compared to Javascript, backend business logic can be executed without sweat or hassle, unlike Javascript. This is due to the fact Python was built as a general-use language.
- Typescript interfacing is extremely powerful during development, but there's no real way for it to enforce things at runtime, especially when dealing with parsing data from a server. While you're able to anticipate and create types for the responses from an API, you still always need to account for missing data or nulls potentially showing up.
- Visual Studio Code can forward the development server hosted on a remote client, meaning you're still able to test and preview your frontend and backend code outside of the server you're developing on.
- Goals and subgoals need to be constantly re-addressed and modified for the duration of a project, there is no way to anticipate every roadblock or problem for a project at the very beginning. Communicating with the client and your team consistently is the key to the steady development of a project.

## XII. Acknowledgments

Our team would like to thank our client, Kevin Wolver, and his team for their continued support and patience for the duration of this project. We greatly appreciated their early support in getting us started when dealing with Qualcomm's information technology (IT) team and familiarizing us with their systems since it was extremely daunting and difficult. Additionally, their continued advice and feedback during this project have helped keep us on track and informed on every part of Jira's and Qualcomm's internal conventions and workings. We're grateful for Kevin Wolver's ability to take time out of his days at the start of the field session to show us around Qualcomm's campus, which was a unique and fun introduction to the company.

Furthermore, we would like to thank our advisor, Rob Thompson, for guiding us through this project's development and classroom side. He helped push us to shrink our deadlines way before any work was due to ensure we had time to make changes towards the due date and be able to work ahead–not having to worry about doing any last-minute work.

## XIII. Team Profile

- Dean Coventry
    - Dean is a rising junior in the general CompSci program with a personal interest in frontend web development. In his free time, he often brews many different kinds of imported tea and reads.
- Sean Williams

    - Sean is a general track CS major going into his (hopefully) last year at Mines. He has interests in game development, art, and music.
- Vincent Nguyen
    - Vincent is a rising sophomore studying computer science at Mines. He is interested in working out, programming, and spending time with friends. In his free time, he also works on personal programming projects and is personally flexible on any component code-related he works on easy or hard.

## References


[1] ACM Code Task Force, “ACM Code of Ethics and Professional Conduct,” Code of Ethics, https://www.acm.org/code-of-ethics (accessed Jun. 11, 2024).

[2] “Code of ethics,” IEEE Computer Society, https://www.computer.org/education/code-of-ethics (accessed Jun. 11, 2024).

# Appendix A – Key Terms

Include descriptions of technical terms, abbreviations, and acronyms

| Term | Definition |
|---|---|
| *Base station* | *A base station is a central point for communication between cellular devices in an area. It is most commonly used in telecommunication and network towers/devices.* |
| *Test Base Station (TBS)* | *Test Base Station is a component whose purpose is to test cellular tower base stations by simulating a production-ready base station.* |
| *issue* | *A proposed bug/problem that is within a product. Within this report, an "issue" can refer to any JIRA issue - a description of a bug fix, feature request, or goal for a software development team to address.* |
| *Jira* | *Jira is an issue tracker / agile project management suite used internally by Qualcomm. Often within this document our reference to "Jira" actually means Qualcomm's internal Jira database for tracking issues/features/etc for different internal projects.* |
| *JQL* | *Jira Query Language. Similar to SQL, it is the language used to filter, query, or perform actions on Jira's end. Simplifies actions.* |
| *Tier2* | *Tier2 issues / the Tier2 teams refer to the "customer engineering" / customer support teams within Qualcomm.* |
| *Tier3* | *Tier3 issues / Tier3 teams refer to the software/hardware engineering teams within Qualcomm.* |
| *Escalation* | *Escalating an issue is when an issue reported by a customer/customer engineering (Tier2) needs to be re-classified and re-assigned as a software/hardware issue for the Tier3 team to fix. Within Jira, this involves creating a new issue for Tier3 and making it reference the old Tier2 issue.* |
| *Single Page Application (SPA)* | *Single Page Application (referring to the web application interface for our project). A web application that has no routes that lead to other pages in the frontend application. All the interaction is contained on one page where modals and information show up as needed without going to another page.* |
| *Long Term Support (LTS)* | *Categorization of a release in software that signifies that the version is meant for long-term extended use–software support for a very long time.* |
| *UI* | *Short for user interface, the interaction of the user with the application* |
| *shadcn* | *A React component UI library for easily putting together functional UIs without writing code from scratch. See ui.shadcn.com* |
| *frontend* | *The part of the application that the user interacts with* |
| *backend* | *The part of the application that powers the frontend application and executes all the required business logic–the user should not have to worry about this part* |