Margaret Wozniak
margwoz@umich.edu
EECS 481
12/2/2025

HW 6b Report - Hydrant

Name and Uniquname:

This report is being written and submitted by Margaret Wozniak, margwoz@umich.edu. The purpose of this report is to present an open source GitHub project that I contributed to as per the EECS 481 HW6b specifications.

Selected Project:

In homework 6a, I originally selected the OpenMetadata project on GitHub to contribute to. However, after spending roughly two and a half weeks working on my selected task for the project, my local build, which was previously working, began to fail. I lost the ability to build the UI frontend in developer mode, along with all of my work towards my task. I reached out to the support channel in the OpenMetadata Slack regarding my issue and was told that there was nothing they could do as the UI wasn't officially supported on Windows/WSL (Figure 1). While it was suggested that I instead build in a native Linux environment, I was unable to follow this advice as I don't have access to a second virtual machine in which I can run a native Linux environment. Additionally, I encountered the same errors using the Docker deployment. At this point, I decided to pivot to a different open source project in the interest of time and feasibility, as my original task was to implement and integrate a new feature for the UI.

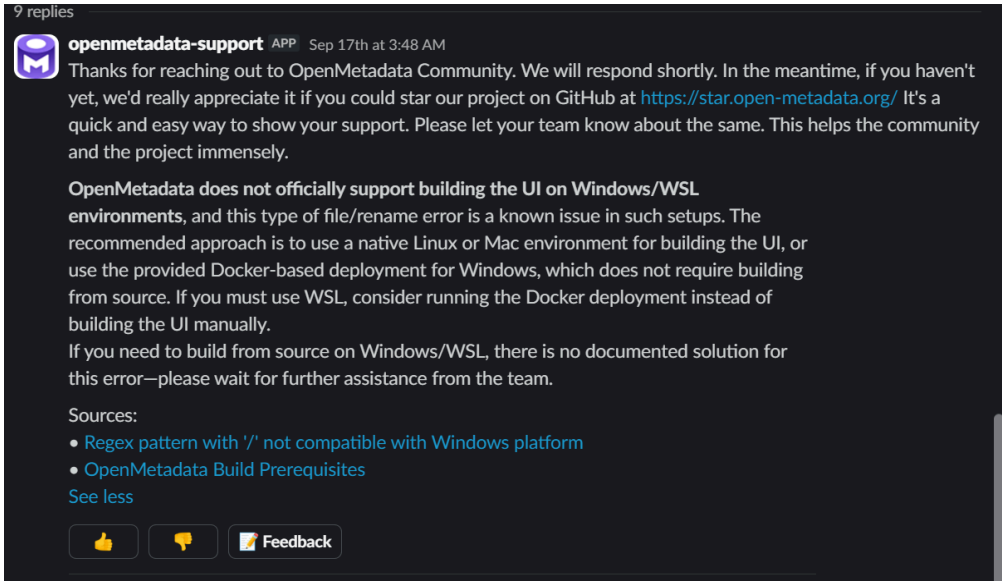

Figure 1. OpenMetaData response

After doing more research into open source projects, I ultimately decided to contribute to the Hydrant project on GitHub. The Hydrant project aims to create and maintain a semester planning app similar to Atlas but for MIT courses. The project has three main components: scraping, parsing, and displaying. These three components work across the front and back end in order to collect course information, modify the format of the information to match the expected format of a human readable schedule, and properly display the courses for students to use and manipulate within a schedule. The Hydrant project is used internally at MIT and is written primarily in TypeScript (78.7%) and Python (19.4%). The project can be found here.

Social Good Indication:

This project does not contribute to social good.

Project Context:

As University of Michigan students with access to applications like Atlas, planning future semesters is easy, convenient, and painless. On the other hand, for students at MIT, who were lacking a schedule building application, planning future semesters was often stressful, inconvenient, and painful. In 2019, MIT student CJ Quines (‘23) set out to resolve this issue and created the Hydrant project, an open source schedule builder designed specifically for MIT. The Hydrant application scrapes the MIT course guide, parses all the information gathered to build an internal database of courses, and then allows students to interact with the database through the Hydrant website. The website displays a calendar that can be populated with courses which can be selected from a scrolling list of classes. The list of classes can be sorted by class number, rating, credit hours, and name. Additionally, the website provides students with the option to export the calendar and pre-register for classes. Hydrant is open to all users, regardless of MIT enrollment, but the options to export calendars and pre-register for classes require an MIT student login to use. Hydrant has no competition and doesn’t have a “business” model in the traditional sense. Hydrant was created by and is maintained by members of the MIT Student Information Processing Board, an organization that consists of student volunteers dedicated to working in service of the broader MIT community. In recent years, the project has been opened to the GitHub open source community to allow anyone interested in computing to contribute to the project. The Hydrant website can be found here.

Project Governance:

The Hydrant project governance is overseen by members of the MIT Student Information Processing Board, and all governance is conducted exclusively within the GitHub repository. At the beginning of this project, I got involved with Hydrant by simply commenting on an open issue and asking to work on it. Almost immediately, one of the project maintainers responded to my comment (Figure 2). His response was very informal, and the communication style was very

reminiscent of casual conversation rather than professional interactions. While working on the issue I claimed, I had no interaction with the project maintainers and I was not required to check in with maintainers at any point. Additionally, the only guidelines (Figure 3) for developers were to not introduce any new technologies to the project, and there was no "contribution guide" like the ones you might find in some repositories. Once I had completed my task, I was free to submit a pull request without doing any kind of quality assurance, although I did do my own quality assurance anyways. After submitting my pull request, one of the maintainers reviewed my code, ran some regression and integration tests, but required nothing from me. Once my code passed the internal tests, it was merged to the repository without any further communication or requirements elicited from me. Overall, the governance and communication for the Hydrant project was very informal, which was expected given that a group of students maintain the project.

Figure 2. Claiming an issue within Hydrant

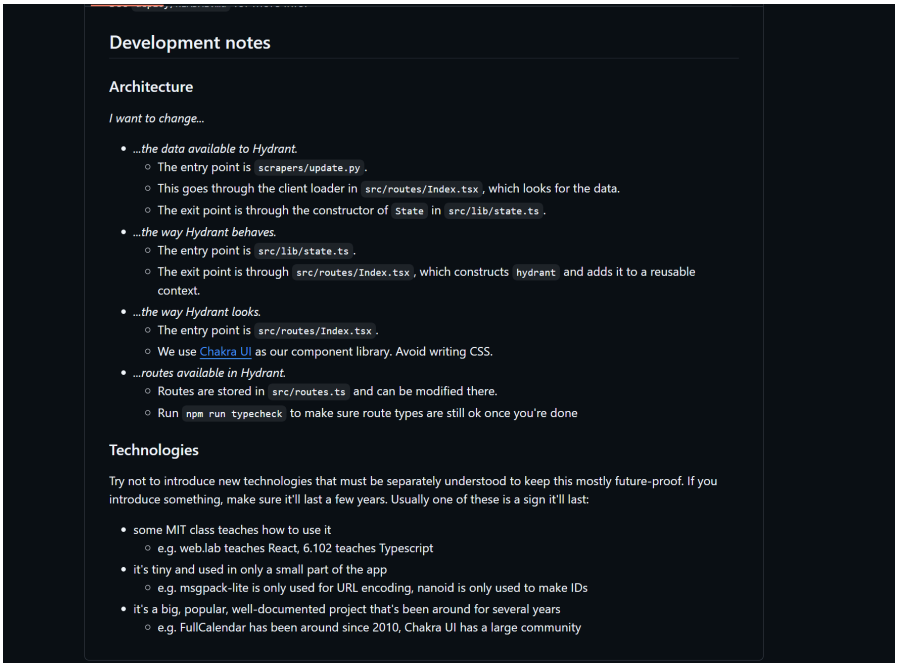

Figure 3. Hydrant contributor guidelines

Alternatively, while working on the OpenMetadata project I found the project governance to be much more formal. Before any issue is claimed, OpenMetadata strongly encourages contributors to join the Slack channel. They then check in with each new member of the Slack channel to encourage them to get involved within the OpenMetadata community (Figure 4). Once you begin to claim issues, you aren't officially working on something until project maintainers assign the task to you. If you begin to work on an issue that is then assigned to someone else, you must get permission from the assignee to contribute to the issue. Additionally, when submitting a pull request, the review process is tedious and takes multiple project maintainers to approve a pull request. Similarly, in order to submit a pull request you must include tests for the changes you made. I found it very interesting to compare this governance to the Hydrant governance, as I feel that it highlights the differences between projects used in industry by professionals and projects used in academia by students. After working with both types of project governance, I found that I preferred the more informal governance as it was a more familiar management style to me. However, I can see the advantages of having a formal governance process in place for larger projects. The formal governance makes it easier to keep track of who is working on what, and allows for OpenMetadata maintainers to have specific points of contact for each issue, rather than having to track down multiple uncoordinated contributors.

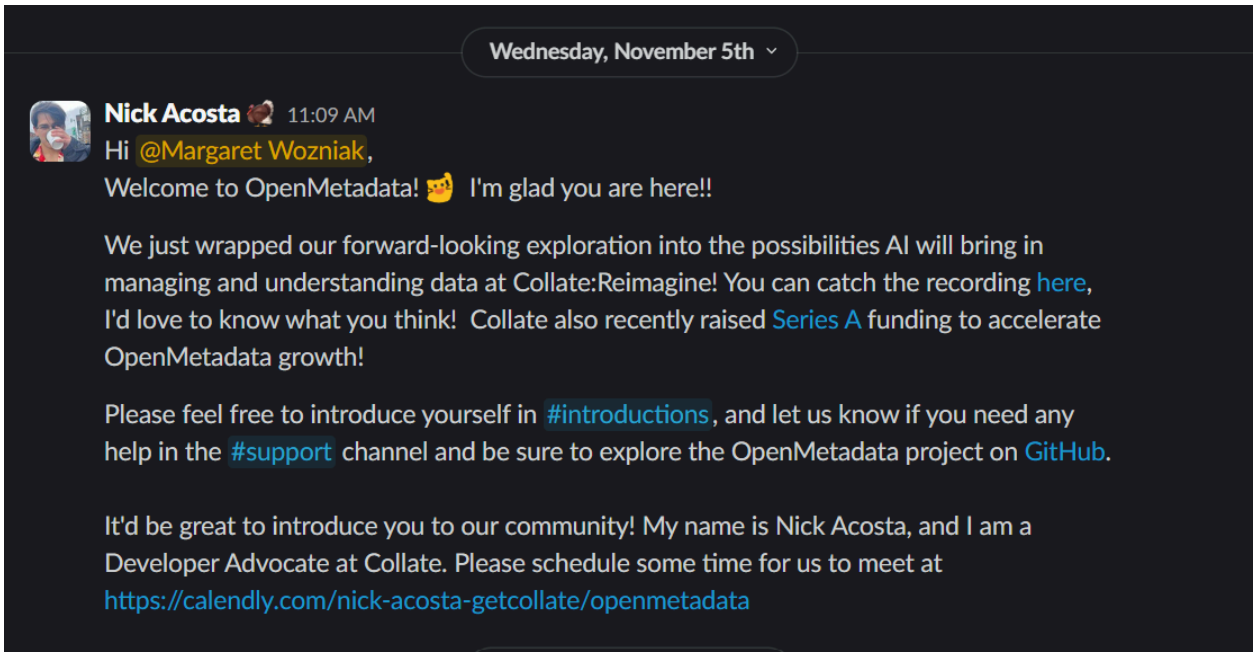

Figure 4. OpenMetadata invite to get involved in the community

Task Description:

As previously mentioned, I was unfortunate enough to have to switch projects between HW6a and HW6b. Included with this was switching tasks between assignment parts. My new task after switching to the Hydrant project was to fix a parsing error as indicated by issue #254 within the

GitHub repository. This issue pertains to a class time format used exclusively by the Sloan School of Management in the upcoming semester that can't be properly parsed using the current parsing functions. My task was to debug this parsing error and implement the necessary changes within the parsing functionality such that these classes could be properly parsed and added to the database. Issue #254 where this buggy behavior is reported can be found here.

After being assigned to the issue, I began my task by building a baseline comprehension for how the scraping and parsing functions worked. In order to do this, I first went through the files included in the scrapers folder of the repository and read their contents and any comments left regarding the functionality. After reading through all the files, I then collected courses from the MIT course guide (Figure 5) for testing purposes. While collecting courses, I placed a specific emphasis on collecting courses that differed in offering type (Lecture, Recitation, Lab, Discussion), days offered (MTWRF), and time slots offered (AM-AM, AM-PM, PM-PM). Once I had collected sixteen courses, including the known buggy course from Issue #254, I placed breakpoints within the main parsing file, fireroad.py, and used the VSCode debugger to manually step through the parsing process for several non-buggy courses. The goal of manually stepping through the parsing process was to understand how each parsing function received and modified course information in non-buggy courses in order to diagnose the issue within the buggy courses. Once I felt I had a solid understanding of the parsing process, I then set the breakpoints to only trigger for the buggy course indicated in Issue #254 and manually stepped through the code until I hit the ValueError raised in Issue #254. Once I hit this error, I noticed that the error wasn't caught and instead propagated out to the main function, causing a valid course to be removed from the database on the grounds that it was reported by the ValuError as incomplete. Based on this behavior, I decided that error handling was faulty, and upon further inspection of the error handling and the error raising behavior, realized that the code was raising a ValueError and was looking for a KeyError. I changed the error handling to look for a ValueError instead. Please see the Quality Assurance section of this paper for details regarding the quality assurance steps taken after making this change. Below is the program output when the full program was run before my change (Figure 6) and after (Figure 7).

MIT REGISTRAR'S OFFICE

Registrar Home | Registrar Search:

MIT Course Picker | Hydrant

Home | Subject Search | Help | Symbols Help | Pre-Reg Help | Final Exam Schedule | My Selections

Course 1: Civil and Environmental Engineering
IAP/Spring 2026

Course 1 Home CI-M Subjects for Undergraduate Majors IAP only Evaluations (Certificates Required)

| 1.00-1.149 | 1.150-1.499 | 1.50-1.999 plus UROP and Thesis |

Fundamentals

1.00 Engineering Computation and Data Science

(Subject meets with 1.001)
Prereq: Calculus I (GIR) and ((6.100A and 6.100B) or (6.100L and 16.C20))
Units: 3-2-7
Lecture: MW9.30-11 (1-390) Lab: F9-11 (1-390)

Presents engineering problems in a computational setting with emphasis on data science and problem abstraction. Covers exploratory data analysis and visualization, filtering, regression. Building basic machine learning models (classifiers, decision trees, clustering) for smart city applications. Labs and programming projects focused on analytics problems faced by cities, infrastructure, and environment. Students taking graduate version complete additional assignments and project work.
J. Williams
No textbook information available

1.000 Introduction to Computer Programming and Numerical Methods for Engineering Applications

Prereq: None. *Coreq: 18.03*
Units: 3-2-7

Presents the fundamentals of computing and computer programming (procedural and object-oriented programming) in an engineering context. Introduces logical operations, floating-point arithmetic, data structures, induction, iteration, and recursion. Computational methods for interpolation, regression, root finding, sorting, searching, and the solution of linear systems of equations and ordinary differential equations. Control of sensors and visualization of scientific data. Draws examples from engineering and scientific applications. Students use the Python programming environment to complete weekly assignments.
R. Juanes

Figure 5. MIT course catalog

```
margwoz@MSI:~/hydrant$ python3 -m scrapers
=== Update fireroad data (pre-semester) ===
Can't parse schedule 15.089: ValueError('Invalid timeslot W, 11, True')
Got 873 courses
Skipped 6066 courses that are not offered in the IAP term
=== Update fireroad data (semester) ===
Can't parse schedule 15.089: ValueError('Invalid timeslot W, 11, True')
Got 4226 courses
Skipped 2713 courses that are not offered in the spring term
=== Update catalog data ===
```

Figure 6. Program output pre-fix

```
margwoz@MSI:~/hydrant$ python3 -m scrapers
=== Update fireroad data (pre-semester) ===
Got 873 courses
Skipped 6066 courses that are not offered in the IAP term
=== Update fireroad data (semester) ===
Got 4226 courses
Skipped 2713 courses that are not offered in the spring term
=== Update catalog data ===
Scraping page: m1a.html
Scraping page: m1b.html
Scraping page: m1c.html
```

Figure 7. Program output post-fix

Submitted Artifacts:

For my task, I submitted the following artifacts: the change made to the code (Figure 8), the unit tests I created for quality assurance purposes (Figure 9), and the pull request submitted to the Hydrant repository (Figure 10). The pull request, which includes both artifacts, can also be found here.

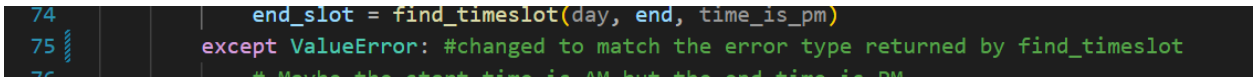

Figure 8. Codebase change made

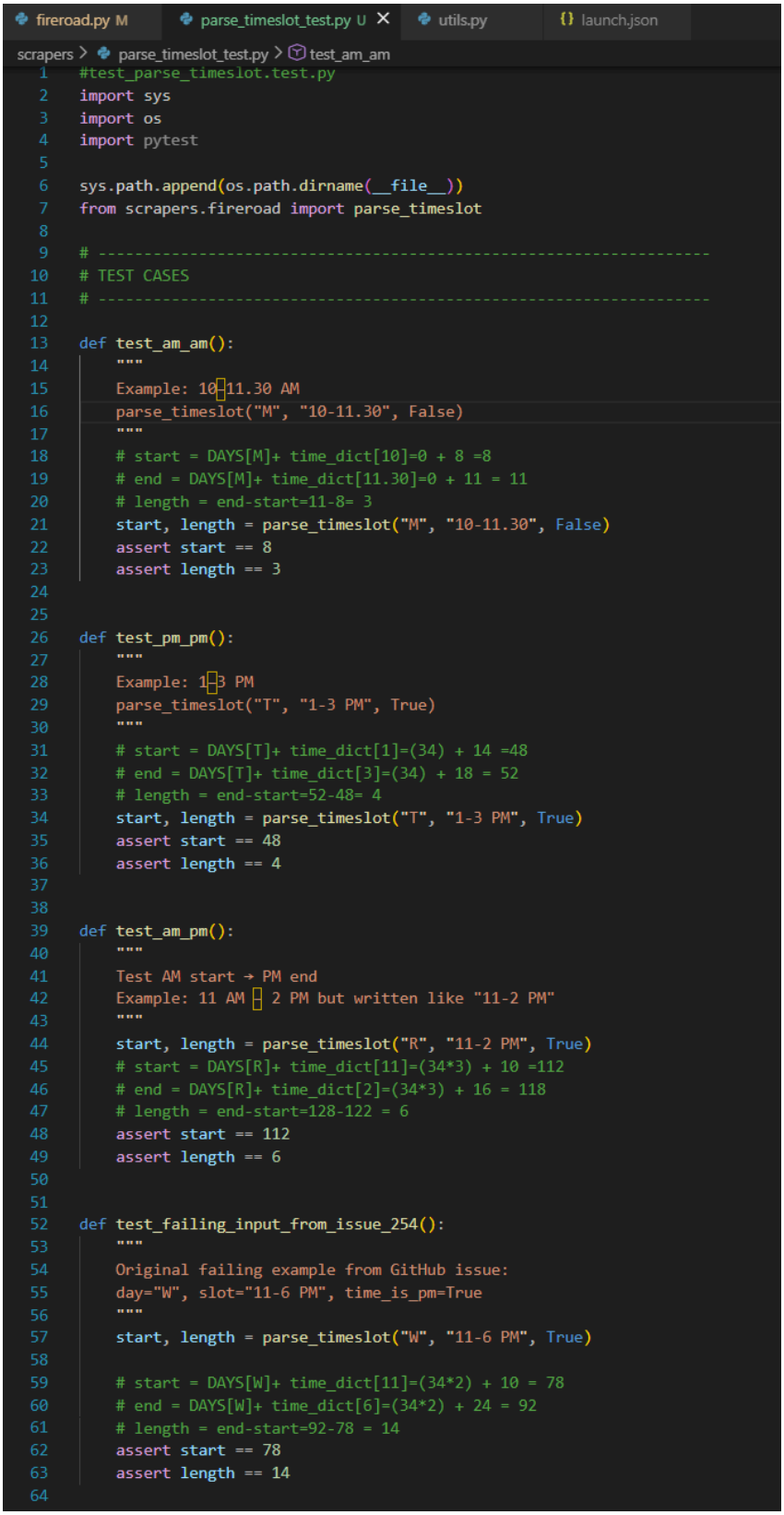

Figure 9. Unit tests written

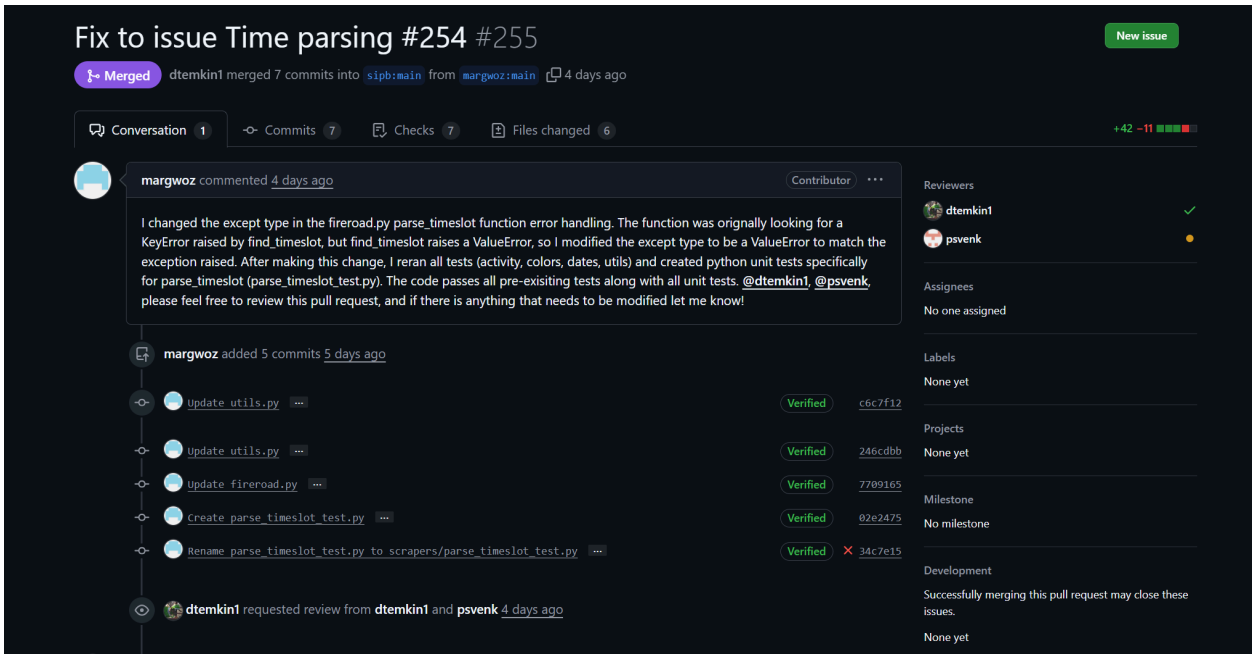

Figure 10. Pull request submitted to Hydrant

Quality Assurance Strategy:

For my task, I chose to use regression testing and unit tests for quality assurance. While completing my task, I changed the error handling to catch a ValueError instead of a KeyError. Due to the nature of this change, I felt that I needed to ensure that all previous tests still passed and that I hadn't inadvertently caused more errors by removing the logic that was previously in place to catch them. This led to the decision to use regression testing as one of my quality assurance activities, as running regression testing could show me any previously passing tests that were now failing due to the changed error handling. Along with regression testing, I decided to use unit tests to ensure that all expected use cases were handled properly. Before I had worked through the codebase to diagnose the issue, I originally believed that Issue #254 was the result of a mishandled edge case for a class that started in the morning and ended in the afternoon. While I was fairly certain that by fixing the mismatched exception types the edge case would be handled correctly by the code run when the exception was caught, I wanted to be able to definitively say that it was. In order to do this, I wrote unit tests for the three time slot cases, AM-AM, AM-PM, and PM-PM. The main motivation for unit testing was that it would allow me to isolate the three time slots and the parse_timeslot function in order to ensure that the previously buggy parse_timeslot function was behaving as intended for all use cases. The unit tests were constructed by creating a realistic time slot for each use case, calling parse_timeslot, and asserting that the parse_timeslot returned the correct timeslot for each case. Once I had these tests written, I ran both the regression tests and unit tests, and only submitted a pull request once I had confirmed that I was passing all regression and unit tests.

Quality Assurance Evidence:

Regression Testing

Before I made any changes to the Hydrant codebase, I ran all existing tests (stored under the tests folder of the repository) in order to get a baseline for future regression testing. Note that in the original run (Figures 11,12,13), eight tests relating to the front end failed. This was expected as I didn't build the front end for a purely backend task. After making my change to the codebase, these same eight tests failed, but all other tests passed (Figures 14,15,16). As such, the tasks passing before the change still passed, and the tests pertaining to something I did not build still failed. Additionally, after submitting my pull request, my code was run through regression tests by the Hydrant maintainers and passed all tests (Figure 17), including the front end tests.

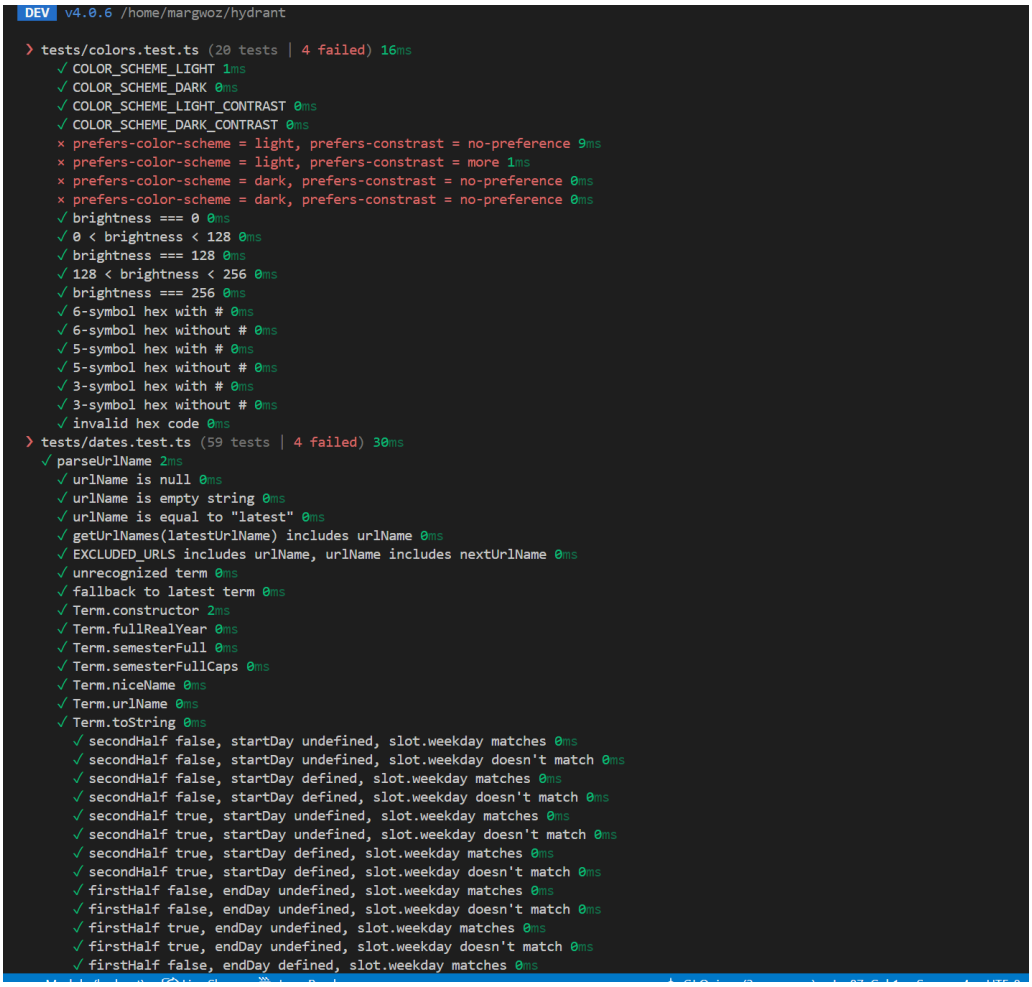

Figure 11. Pre-fix failing tests

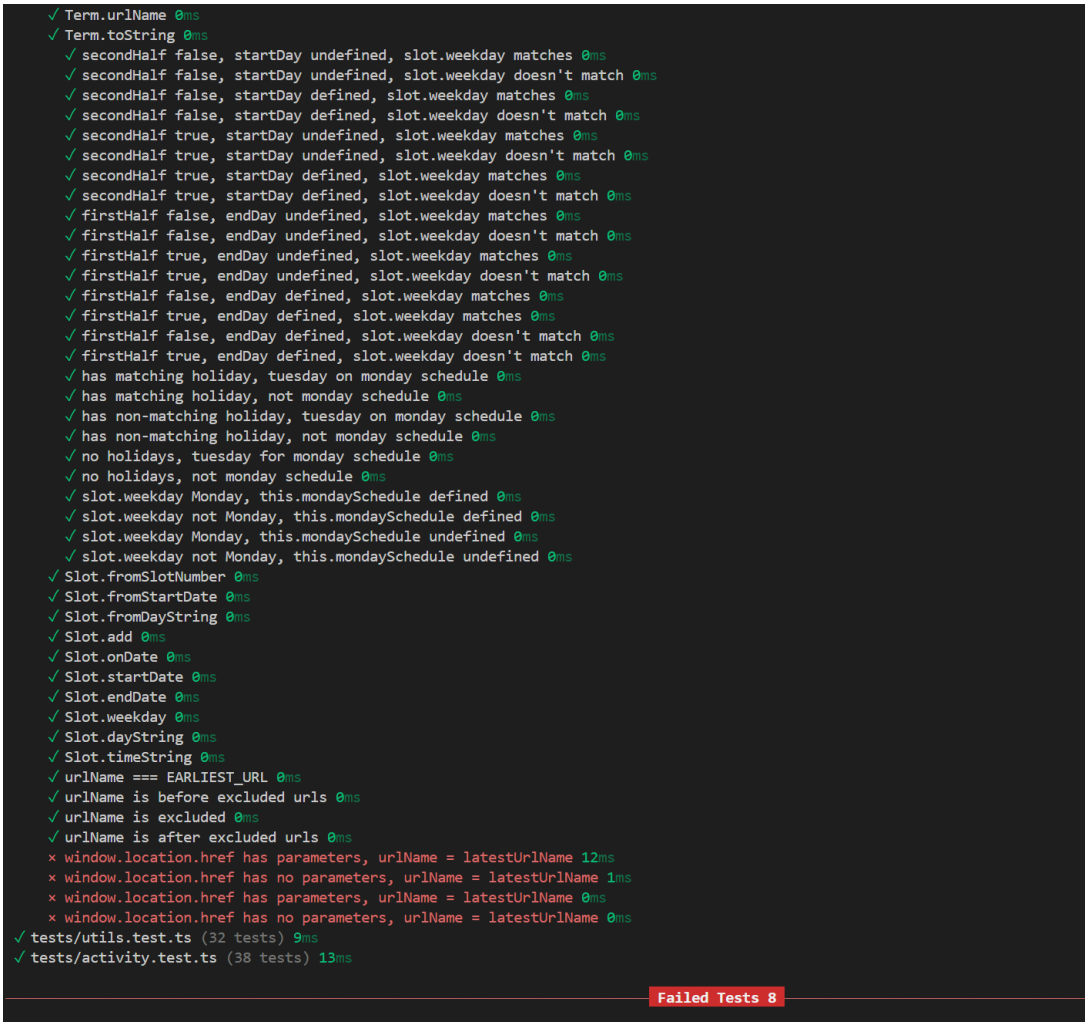

Figure 12. Pre-fix failing tests

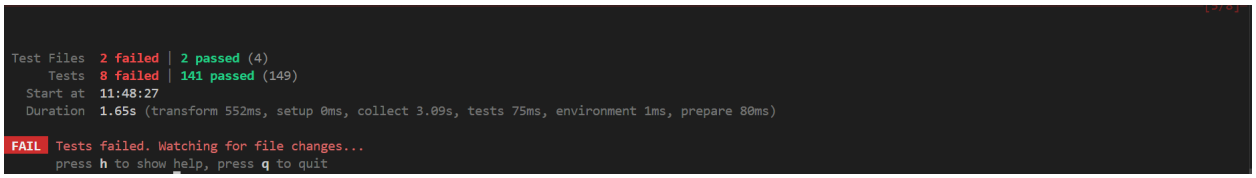

Figure 13. Pre-fix failing tests summary

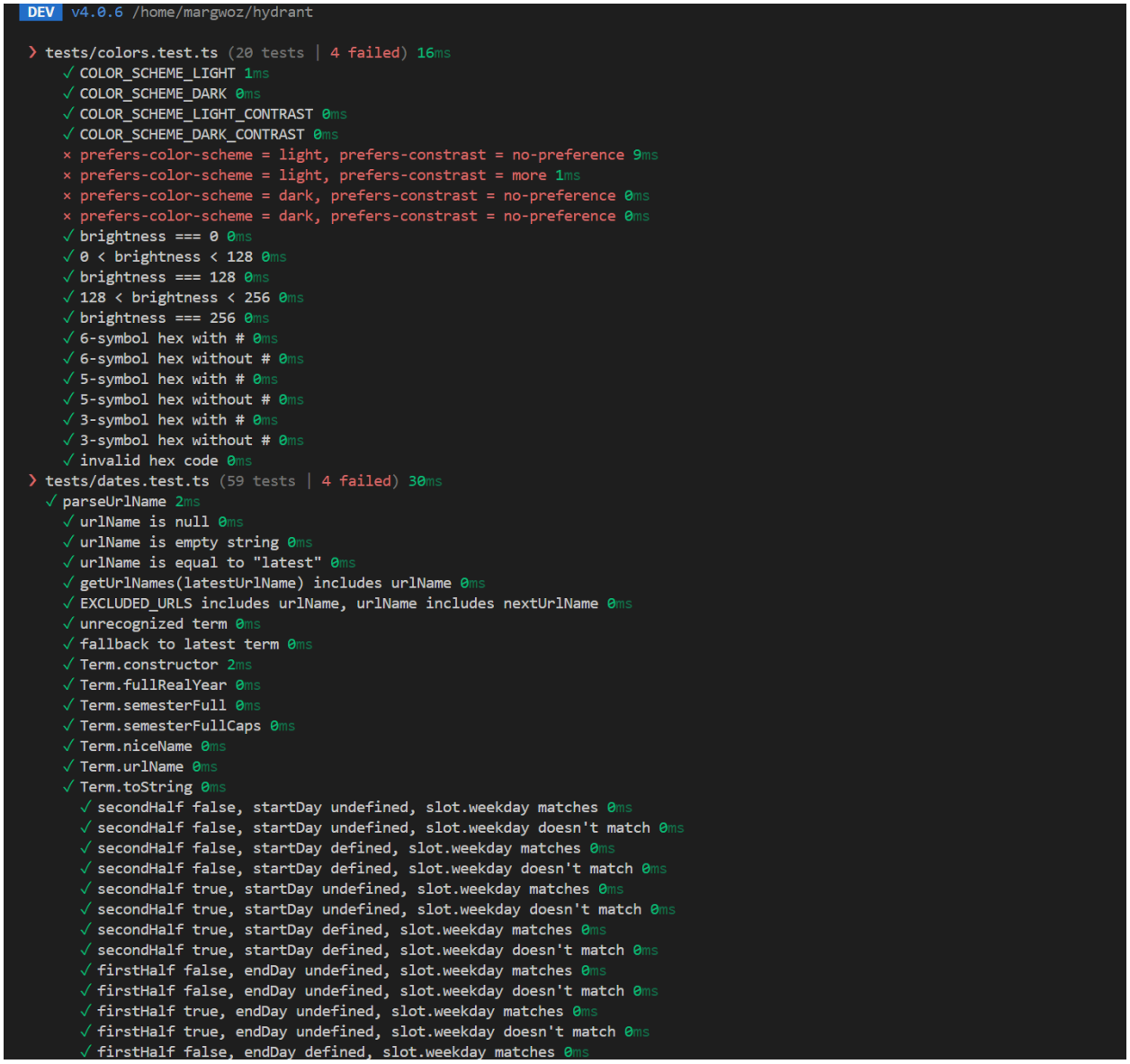

Figure 14. Post-fix failing tests

```
    ✓ Term.urlName 0ms
    ✓ Term.toString 0ms
      ✓ secondHalf false, startDay undefined, slot.weekday matches 0ms
      ✓ secondHalf false, startDay undefined, slot.weekday doesn't match 0ms
      ✓ secondHalf false, startDay defined, slot.weekday matches 0ms
      ✓ secondHalf false, startDay defined, slot.weekday doesn't match 0ms
      ✓ secondHalf true, startDay undefined, slot.weekday matches 0ms
      ✓ secondHalf true, startDay undefined, slot.weekday doesn't match 0ms
      ✓ secondHalf true, startDay defined, slot.weekday matches 0ms
      ✓ secondHalf true, startDay defined, slot.weekday doesn't match 0ms
      ✓ firstHalf false, endDay undefined, slot.weekday matches 0ms
      ✓ firstHalf false, endDay undefined, slot.weekday doesn't match 0ms
      ✓ firstHalf true, endDay undefined, slot.weekday matches 0ms
      ✓ firstHalf true, endDay undefined, slot.weekday doesn't match 0ms
      ✓ firstHalf false, endDay defined, slot.weekday matches 0ms
      ✓ firstHalf true, endDay defined, slot.weekday matches 0ms
      ✓ firstHalf false, endDay defined, slot.weekday doesn't match 0ms
      ✓ firstHalf true, endDay defined, slot.weekday doesn't match 0ms
      ✓ has matching holiday, tuesday on monday schedule 0ms
      ✓ has matching holiday, not monday schedule 0ms
      ✓ has non-matching holiday, tuesday on monday schedule 0ms
      ✓ has non-matching holiday, not monday schedule 0ms
      ✓ no holidays, tuesday for monday schedule 0ms
      ✓ no holidays, not monday schedule 0ms
      ✓ slot.weekday Monday, this.mondaySchedule defined 0ms
      ✓ slot.weekday not Monday, this.mondaySchedule defined 0ms
      ✓ slot.weekday Monday, this.mondaySchedule undefined 0ms
      ✓ slot.weekday not Monday, this.mondaySchedule undefined 0ms
    ✓ Slot.fromSlotNumber 0ms
    ✓ Slot.fromStartDate 0ms
    ✓ Slot.fromDayString 0ms
    ✓ Slot.add 0ms
    ✓ Slot.onDate 0ms
    ✓ Slot.startDate 0ms
    ✓ Slot.endDate 0ms
    ✓ Slot.weekday 0ms
    ✓ Slot.dayString 0ms
    ✓ Slot.timeString 0ms
    ✓ urlName === EARLIEST_URL 0ms
    ✓ urlName is before excluded urls 0ms
    ✓ urlName is excluded 0ms
    ✓ urlName is after excluded urls 0ms
    × window.location.href has parameters, urlName = latestUrlName 12ms
    × window.location.href has no parameters, urlName = latestUrlName 1ms
    × window.location.href has parameters, urlName = latestUrlName 0ms
    × window.location.href has no parameters, urlName = latestUrlName 0ms
  ✓ tests/utils.test.ts (32 tests) 9ms
  ✓ tests/activity.test.ts (38 tests) 13ms

                                                    Failed Tests 8
```

Figure 15. Post-fix failing tests

```
 Test Files  2 failed | 2 passed (4)
      Tests  8 failed | 141 passed (149)
   Start at  11:53:18
   Duration  1.48s (transform 521ms, setup 0ms, collect 2.79s, tests 68ms, environment 1ms, prepare 54ms)

 FAIL  Tests failed. Watching for file changes...
       press h to show help, press q to quit
```

Figure 16. Post-fix failing tests summary

Figure 17. All checks passing within Hydrant repository

Unit Testing

After making my changes to the Hydrant codebase, I wrote four unit tests, one for each of the following cases: AM-AM, AM-PM, PM-PM, and one for the known failing input. I then ran these unit tests using pytest and passed all four (Figure 18).

```
margwoz@MSI:~/hydrant$ pytest -q
....
4 passed in 0.03s
margwoz@MSI:~/hydrant$
```

Figure 18. Unit tests passing

Plan Updates:

Over the course of this assignment, there were an unexpected number of plan changes. Almost all of these changes stemmed from the fact that my original task exploded, and I had to switch both projects and tasks. This added a large amount of extra time to my plan, as I had to repeat the process of selecting a task, building codebase comprehension, along with actually completing my new task. Overall, I spent 39.2 hours on this assignment compared to my original estimate of 28.25 hours. As such, the final "what actually happened schedule" is vastly different from the original planned schedule and is as follows:

11/3 - 11/7
Description: Over the course of this week I selected a project (4 hrs), built the project locally (2 hrs), and wrote the HW 6a report (2hrs).
Total time: 8 hours

11/10- 11/14

Description: Over the course of this week I worked on comprehending the OpenMetadata codebase and learning TypeScript.
Total time: 8 hrs

11/17 - 11/21
Description: Over the course of this week I began working on the OpenMetadata feature frontend. Halfway through the week, my project exploded and I argued with the maintainers about my lost work and the build no longer compiling (1 hr), tried to get the build re-running (3hrs), gave up on the OpenMetadata project (0 hrs)
Total time: 9 hrs

11/24 - 11/28
Description: Over the course of this week I found a new project (2 hrs), conducted codebase comprehension for Hydrant (3 hrs), debugged the issue (1.5 hrs), implemented the fix (0.25 hrs), wrote unit tests (0.25 hrs), ran unit and regression tests (0.1 hrs), and submitted a pull request (0.1 hrs).
Total time: 7.2 hrs

12/1 - 12/5
Description: Over the course of this week I wrote my HW6b report.
Total time: 7 hrs

Experience and Recommendations:

Original Task Selection - OpenMetadata

In order to complete this assignment, I first had to select a task. This was incredibly daunting, as I had never contributed to or even explored an open source GitHub project before this assignment. Additionally, I was very intimidated by the sheer size of the search space as there are thousands of open source GitHub repositories out there. In an attempt to get some advice and reassurance, I made a Piazza post (Figure 19), to which Professor Weimer responded (Figure 20) with some very helpful insights. Along with following his advice, I also found websites such as Good First Issues and Up for Grabs helpful to narrow the search space for projects. After spending a decent amount of time looking for projects, I ended up settling on the OpenMetadata project. I picked this project for several reasons: the strong contributor community, the large amounts of documentation, and the recent creation of an issue that seemed within the scope of my abilities. I selected my original task of implementing a new notification icon based on this issue. While the HW6 spec advises against purely UI tasks due to difficulties to conduct quality assurance on them, I felt that this task, while UI centered, was integrated enough with the backend that there were ample opportunities to conduct quality assurance.

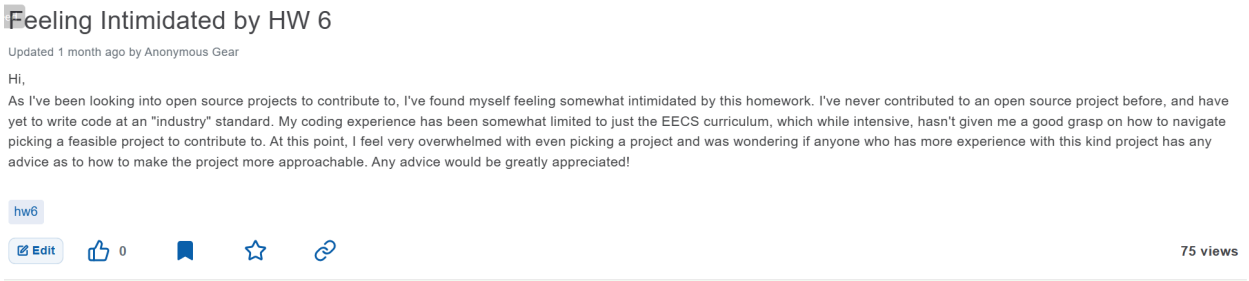

Figure 19. Piazza Post

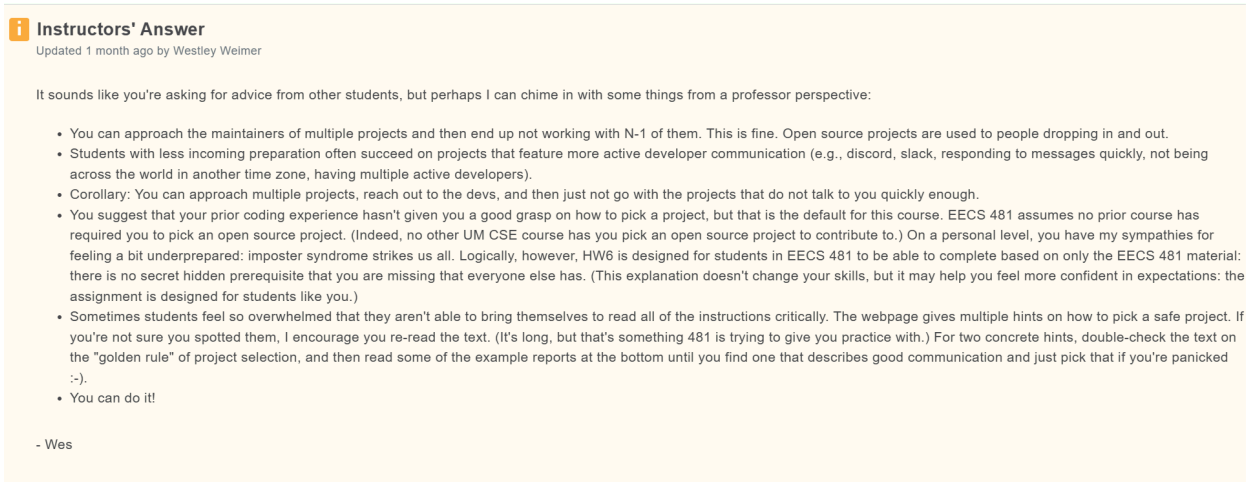

Figure 20. Professor Weimer's response

Completing the Task - OpenMetadata

After selecting the feature implementation task, I reached out to the project maintainers and formally asked to be assigned to the issue. After being assigned, I then began the process of getting my device and myself ready to contribute. This involved joining the OpenMetadata Slack channel in order to stay in touch with maintainers and other contributors, getting the build set up locally, and beginning to go through the codebase comprehension process. After joining the Slack channel, I was directed to the OpenMetadata contributor set up guide, found here, in order to get a local build running. After following the guide, I was able to get the backend and UI running locally in developer mode. Once I had everything running, I began the codebase comprehension process. OpenMetadata is a huge project, so I had to be very selective with which files I chose to examine and read through. I ran a search for the word notification in order to narrow down the codebase to files that made references to the existing notification management system and UI components. This still left about 100 files, so I narrowed it down even further to files that only existed in the UI components folder of the repository and contained the word notification in the title. This left about 25 files, which I then started to read through. As I read through the files, I had to do a lot of research about specific syntax, as most of the code was

written in TypeScript and React, both of which I'd never used before. Additionally, I had to spend a lot of time reading documentation in order to understand what each specific function was doing for the UI. Overall, it took me roughly eight hours to build a baseline understanding of the notification UI code, TypeScript, and React.

After building this baseline, I then began to work on modifications to the code in order to complete the feature request. This implementation process required a lot of research into notification infrastructure within React and different ways to combine style components in order to get the desired behavior within the OpenMetadata notification icon. Once I had done my research, I began actually modifying the code to implement the feature. However, about halfway through my implementation, my laptop force restarted, which in turn force quit my VSCode instance. When this forced quit occurred, I lost all work in the file I had been working in, as I hadn't yet pushed my changes to my forked GitHub. Additionally, when the quit occurred I lost the ability to locally build the UI part of the OpenMetadata project, which was incredibly problematic as my entire task revolved around the UI. I reached out to the OpenMetadata maintainers asking for help getting the build running again, and was told that there was nothing they could do as the UI build wasn't officially supported on Windows/WSL architecture (Figure 1). They suggested that I try rebuilding using Docker, but I was unsuccessful in getting this to work. I spent roughly three hours trying in vain to repeat the steps that let me build the UI the first time, but was ultimately unsuccessful. At this point, I made the executive decision to pivot to a different project, as the OpenMetadata project seemed like a lost cause. I debated switching tasks rather than projects, but ultimately decided against it due to the fact that I had lost trust in the OpenMetadata maintainers as they had failed to disclose that the UI build wasn't supported on certain systems. I felt that that information should have been included in the build guide or documentation, and at the very least should have been communicated to me when I was assigned a UI task to complete.

Starting Over - Hydrant Project and Task Selection

After the failure of the OpenMetadata project, which occurred exactly two weeks and three days before the HW6b report was due, I was left scrambling for a new project. It again seemed incredibly daunting to select a new project and task, but for different reasons this time. This time around, I felt confident in my ability to find a project and task, but was more concerned with the feasibility of actually being able to complete a task, do quality assurance, and write the report before the suggested report deadline of 12/4. Additionally, I was worried about being able to balance this project with two other final projects, a final research paper, and two exams. Based on these fears and the remaining time left on the project, I decided to modify the project search space to only include smaller projects that used TypeScript, Python, and C++. Using this search criteria, I came across the Hydrant project. The Hydrant project stood out to me as it was small, highly active, written in TypeScript and Python, and had an issue that was opened three hours before I came across the project. I decided almost immediately that this would be my project, as

it was in languages I was familiar with and had an issue that seemed to be within my ability to solve in two weeks. I placed a comment asking to be assigned the issue, and two hours after that, received a greenlight response from the maintainers. This cemented my switch, and I fully pivoted from OpenMetadata to Hydrant.

Completing the Task - Hydrant

Once I received the green light from maintainers, I immediately started working on my task. To begin, I pulled the codebase into a local VSCode instance and started reading through files to get a sense of what the code was actually doing. This experience was very different from the codebase comprehension I did for OpenMetadata for several reasons. To begin with, Hydrant has virtually no documentation and no way to quickly and reliably reach maintainers. This made understanding the code much more difficult, as I had no way to get any kind of explanation for what the code was doing. Additionally, the parsing functionality within Hydrant is done very non-intuitively and uses base 34 rather than base 24 for all time slot conversions. I had to figure this out on my own, which was much harder than figuring out the functionality for various OpenMetadata functions. However, one strength of Hydrant is that the code base was small enough that it was feasible for me to place break points and manually step through examples to see how the parsing functionality worked. Additionally, I had a lot of courses I knew were parsed correctly, and I was able to access and use those courses as my example courses for my manual walk-throughs. This allowed me to build a deeper understanding of how the code worked, as I was able to see at each step of the process how information was being processed. I tend to be a visual learner, so I found it easier to build my own interpretation and intuition by watching variables change as they progressed through the program as opposed to reading documentation written by someone else.

Once I had an understanding of the codebase, actually fixing the issue was rather trivial. In order to isolate the issue I used my breakpoints and set them to trigger only for the course causing the ValueError seen in Issue #254. Once the breakpoints triggered, I then walked through the execution of the buggy input until I reached the buggy behavior. Once I reached this behavior, I was able to isolate the issue to a pair of functions, one of which called the other and then used the returned result. I found the exact error being raised in the called function, and then looked at the error handling in the calling function. In doing so, I found a mismatch between the error type being raised and the error type being checked for. I then switched the error type being checked for to match the one being raised and reran the original buggy course, which was now parsed correctly.

After making the switch in the error handling, I had some suspicions that by changing the error type being caught, I would have inadvertently caused errors previously being caught to be missed. This suspicion informed my quality assurance process, and led me to use regression and unit testing in order to verify my change. As previously discussed in the quality assurance

section of this report, these two metrics were chosen as they allowed me to ensure that the change made to one function, i.e. one unit, did not reintroduce old bugs and that the unit functioned as expected in all possible use cases. Once I had passed all regression and unit tests, I then created a pull request in which I explained the changes I made, why I made them, and the quality assurance processes I used.

Failure, Flexibility, and Risk in Software Engineering

Over the course of this homework, I learned many valuable lessons about software engineering in the real world, the most important of which pertained to failure, flexibility, and risk. Going into this project, I expected things to be difficult, and I expected there to be setbacks; what I did not expect was complete and total failure. Up until this point in my engineering career, I have always been able to execute a plan, albeit with some variation, to get to the intended and original end result. This was the first time that I had to not only deviate from the plan, but abandon the end goal entirely. This was something that I found very difficult to accept and deal with as I am someone who does not take kindly to perceived failures. However, I think that from this "failure", which technically isn't even a failure, I learned a very valuable lesson about software engineering in the "real" world: failure is only a matter of perspective.

When reading the instructions for this assignment, it is never stated anywhere that I had to complete my task. In fact, it was mentioned numerous times that a very likely outcome was that I would not complete my task, and that that was entirely okay. Success, as defined by the instructions, is to simply engage with and document the software engineering process and write a report on it. However, success as defined by myself, was to complete the original task that I selected. Depending on which definition of success you use, I either failed or succeeded on this assignment. Importantly, the only definition that matters is the one in which I succeeded. I think that engineers have a tendency to internally define success such that success for them is entirely different from success as defined by the requirements. We have a tendency to want things to go the way we plan them, and any variation between the imagined result and the actual is perceived as a failure, even if it objectively isn't one. Additionally, I think that we tend to struggle with abstract notions of success. We like numbers, we want to get 100% coverage or an optimized time complexity, and we forget that, in most cases, these numbers are an upper bound on success, not a lower. This assignment, with its abstract notion of success, taught me that most software engineering will be about meeting a set of requirements and that my personal definition of success, and with it failure, does not matter so long as I meet the requirements set out for me.

Additionally, this taught me that in order to succeed at meeting these requirements, I must let go of my own internal notions of success and embrace flexibility. This assignment has shown me that flexibility is imperative in order to succeed in software engineering, as the plan always changes, primarily as a direct result of risk. When writing the HW6a report, we were asked to write about risks we might encounter in this project and how we would manage them. In my

report, I accounted for several risks, but failed to account for the only risk I ended up actually encountering. In my description of nuanced risks, I forgot the most obvious – that the project itself would no longer be feasible. This risk seemed benign and improbable, and I assumed that as a competent engineer I had fully accounted for all risks and had effectively come up with strategies to manage them. Obviously, I did not account for all risks, and as such, had to scramble and panic midway through the assignment. This oversight taught me that in order to truly have a successful plan, you have to account for all possible risks, not just the ones that you find probable. Similarly, it taught me that even when you do plan for all risks, an unexpected risk can still occur, meaning that in order to be a successful engineer, you need to be able to be flexible and make decisions on the fly. Overall, it taught me that I should plan on not planning for everything, and keep an open mindset about potential paths forward through a project. It also reinforced the importance of risk in the software engineering process.

Recommendations:

After completing this assignment, I have several recommendations for future students. First and foremost, I would recommend that students start this project early. I started this project with a little more than a month to complete it, and still ended up feeling panicked about being able to get it done on time. You will absolutely need more time than you allocated, regardless of your task exploding or not. Starting early gives you the opportunity to make mistakes and change your plan, and helps you avoid a situation in which you don't finish the assignment due to some unexpected issue. Second, I would recommend that students select projects that are well documented and have reliable means of communication with the maintainers. Documentation and communication make the codebase comprehension process exponentially easier, which is the central part of this assignment. Third, I would recommend that students pick tasks that require them to use skills they already possess. My original OpenMetadata task was in a language I didn't know, doing UI work which I had never done. This made my life much harder than it needed to be, and I had to spend a lot of extra time building a skill set to be able to complete my task. The Hydrant task went much smoother because it was in a language I knew and pertained to parsing functions which I had experience with. Lastly, I would recommend that students embrace the process while letting go of the result. Most EECS classes exclusively place emphasis on the final result. They teach students that the only thing that matters is how well your code runs; that success is measured by how many test cases you pass, the space and time complexity of your code, and how many bugs your test cases catch. This is how software engineering works in academia, not how it works in the real world. This assignment aims to introduce you to the reality of software engineering by placing its emphasis on the process, not the end result. Lean into the process, as understanding and enjoying it will make you a better software engineer.

Advice for Future Students:

My advice to future students would be this: Do the readings, go to class, and appreciate the process.

I give my permission to course staff to use my materials for future semesters.

Extra Credit - Pull Request Accepted:

My change was merged into the Hydrant repository on November 25th, 2025. The closed pull request can be found here.

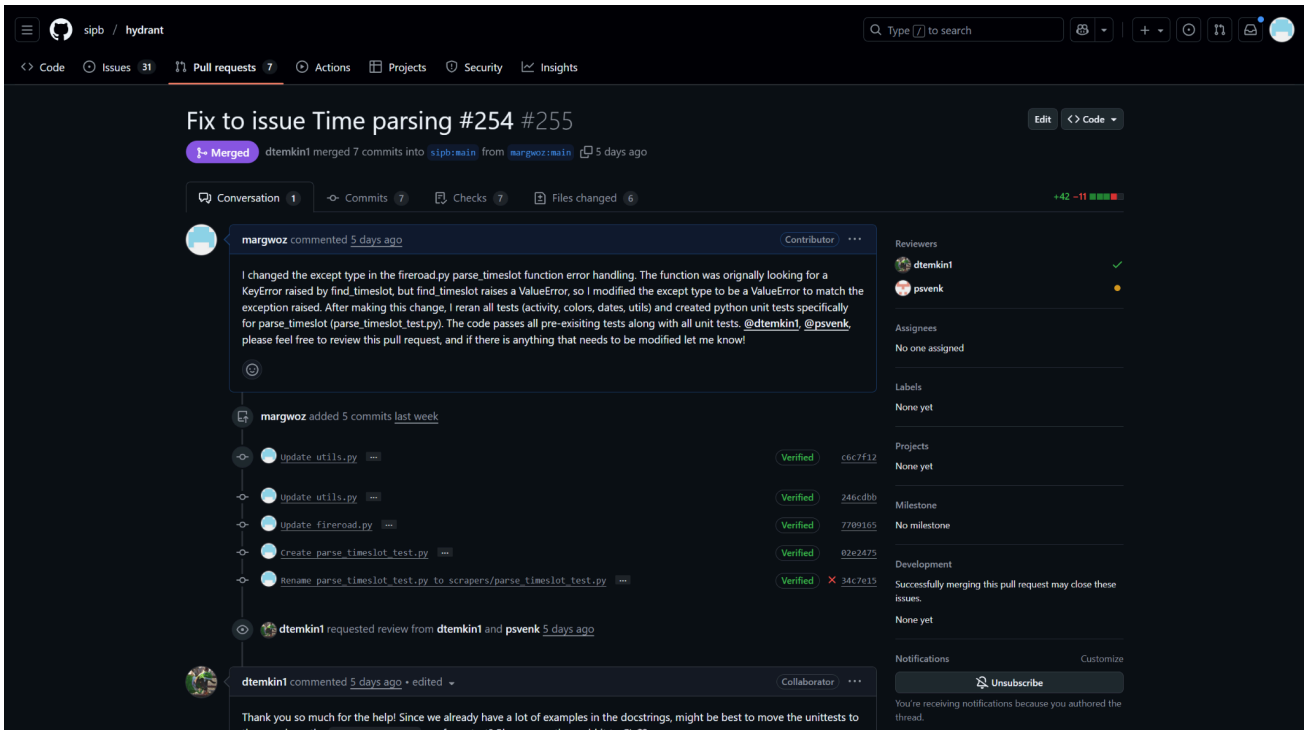

Figure 21. Merged pull request